Quaestiones Informaticae

Vol. 2 No. 1 November, 1982

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.

NOTE FROM THE EDITOR

After an absence of two years we are happy to announce that we are now in a position to continue the publication of Quaestiones Informaticae. The first Volume of QI consists of three numbers, and appeared during the period June 1979 till March 1980 under the editorship of Prof Howard Williams. Because Prof Williams took up a post at the Herriot-Watt University in Edinburgh, he had to relinquish his position as editor. The Computer Society of South Africa, which sponsors the publication of QI, appointed me as editor, whereas Mr Peter Pirow took over the administration of the Journal. The editorial board functions under the auspices of the Publications Committee of the CSSA.

The current issue is Number 1 of Volume 2. It is planned to publish altogether three issues in the Volume, with most of the papers coming from the Second South African Computer Symposium on Research in Theory, Software and Hardware. This Symposium was held on 28th and 29th October, 1981. At present it appears that most of the material published in this Journal comes from papers read at conferences. We invite possible contributors to submit their work to QI, since only the vigorous support of researchers in the field of Computer Science and Information Systems will keep this publication alive.

G WIECHERS

November, 1983

An Efficient Implementation of an Algorithm for Min-Max Tree Partitioning

Ronald I. Becker
Department of Mathematics, University of Cape Town

Yehoshua Perl[1]
Department of Mathematics & Computer Science, Bar-Ilan University

Stephen R. Schach[2]
Department of Computer Science, University of Cape Town

Abstract
An implementation of an algorithm for finding a min-max partition of a weighted tree T with n vertices into q subtrees by means of $k = q-1$ cuts is presented. The implementation is shown to have asymptotic complexity $O(k^3 rd(T) + kn)$, where rd(T) is the number of edges in the radius of T.

1. Introduction

We consider the problem of partitioning a weighted tree into connected components in such a way that the heaviest component is as light as possible.

More formally, let $T = (V,E)$ be an (unrooted) tree with n edges. We associated a non-negative weight w(v) with every vertex $v \in V$. A *q-partition* of T into q connected components $T_1, T_2, \ldots, T_q$ is obtained by deleting $k = q\text{-}1$ edges of T, $1 \leq k \leq n$. The weight $W(T_i)$ of a component T_i is then the sum of the weights of its vertices.

The *min-max q-partition* problem is then: Find a q-partition of T minimizing $\max_{i \leq i \leq q} W(T_i)$.

Applications of this problem arise in paging and overlaying techniques.

In [1] we presented an algorithm for the above problem, and proved its correctness. Here we describe an efficient implementation of our algorithm, and derive its complexity.

Whereas the algorithm itself is straightforward to state, in order to achieve an implementation of low complexity, a careful choice of data structures is necessary. In our implementation we initialize and update five different data structures.

The algorithm involves shifts of two types, down-shifts which improve the partition, and side-shifts which make corrections. The complexity analysis consists of finding bounds on the number of operations required to down-shift a cut, the number of operations to side-shift a cut, and the total numbers of shifts of each kind. Special care must be taken in bounding the number of operations required for down-shifting, in order to obtain low asymptotic complexity.

For the reader's convenience the definitions of [1] are reproduced in Section 2. In addition, a figure is given which illustrates these definitions. The algorithm itself is stated in Section 3; an example showing its operation is also given. The implementation of the algorithm and the complexity analysis are presented in Section 4.

For NP-completness results for the min-max q-partition problem for a general graph (as well as for three related partitioning problems), the reader is referred to [5].

2. Definitions

Transform the given tree into a rooted directed tree by choosing an arbitrary terminal vertex as root, and imposing a top-down direction on the edges. In this paper we use the usual terminology of Graph Theory. If e is a directed edge *incident from* v_1 and *incident to* v_2, denoted by $(v_1 \rightarrow v_2)$, then we will refer to v_1 and *tail*(e) and to v_2 as *head*(e). Edge e is said to be the *father* of edge e_1 if head(e) = tail(e_1), and in this case, e_1 is said to be the *son* of edge e. Edges e_1 abd e_2 are said to be *brothers* if tail(e_1) = tail(e_2). For convenience, if a cut c is assigned to an edge $e = (v_1 \rightarrow v_2)$ then we shall use head(c), tail(c) for head(e), tail(e) respectively. We shall also refer to e as a *son-edge* of v_1 and to the cut c as *incident from* v_1.

A cut is said to be *down-shifted* if it is moved from its present edge to a vacant son-edge. It is said to be *side-shifted at vertex* v if it is moved from its present edge e_1 to a vacant brother edge e_2, and v = tail(e_1)(= tail(e_2)).

We further require the notions of *partial* and *complete rooted subtrees:* a subtree T′ of T is a partial (complete) subtree of T rooted at a vertex v if v is the root of T′, and T′ contains one (every) son of v together with all the latter's descendents.

Let A be an arbitrary assignment of the k cuts to the edges of T. We define a *cut tree* C = C(T,A) to be a rooted tree with k+1 vertices representing r, the root of T, and the k cuts of A. A cut c_1 is the son of r (of a cut c_2) if there exists a (unique) path from r (from head(c_2)) to tail(c_1) containing no cuts.

The *down-component of a vertex* v is obtained from the complete subtree of T rooted at v by deleting the complete subtrees rooted at the heads of all cuts of T immediately below v, if any. The *down-component of a cut* c is the down-component of head(c), and c is called the *top cut* of that component. The *down-component of an edge* e is the down-component of head(e). The *root-component* of T is the component obtained by deleting the complete subtrees rooted at the heads of the sons of r in C. The *up-component* of a cut is the down-component of its father in the cut tree if its father is not the root, else it is the root component. A *bottom cut* of a component is a son of the top cut of the component in the cut tree, if the component has a top cut, else it is a son of the root of the cut tree. The *root of a component* is head(c) for the top cut c, if the component has a top cut, else it is the root of the tree. A component T_i is *lighter* than another component T_j (or T_j is *heavier* than T_i) if $W(T_i) < W(T_j)$.

We illustrate these definitions in Figure 1. Referring to the tree T shown in Figure 1(a), edge($v_2 \rightarrow v_3$) is the father of ($v_3 \rightarrow v_5$), and the brother of edge($v_2 \rightarrow v_8$). Cut c_1 can be downshifted to edge($v_3 \rightarrow v_5$), and side-shifted to ($v_2 \rightarrow v_8$). It is incident from vertex v_2. The subtree comprising vertices {v_3, v_4, v_5, v_6, v_7} is a partial subtree of vertex v_2. The cut tree C is shown in Figure 1(b). Turning now to Figure 1(c), the component B = {v_3, v_5} may be described as

1. Supported in part by the National Council for Research and Development of Israel
2. Supported in part by the Council for Scientific and Industrial Research of South Africa.

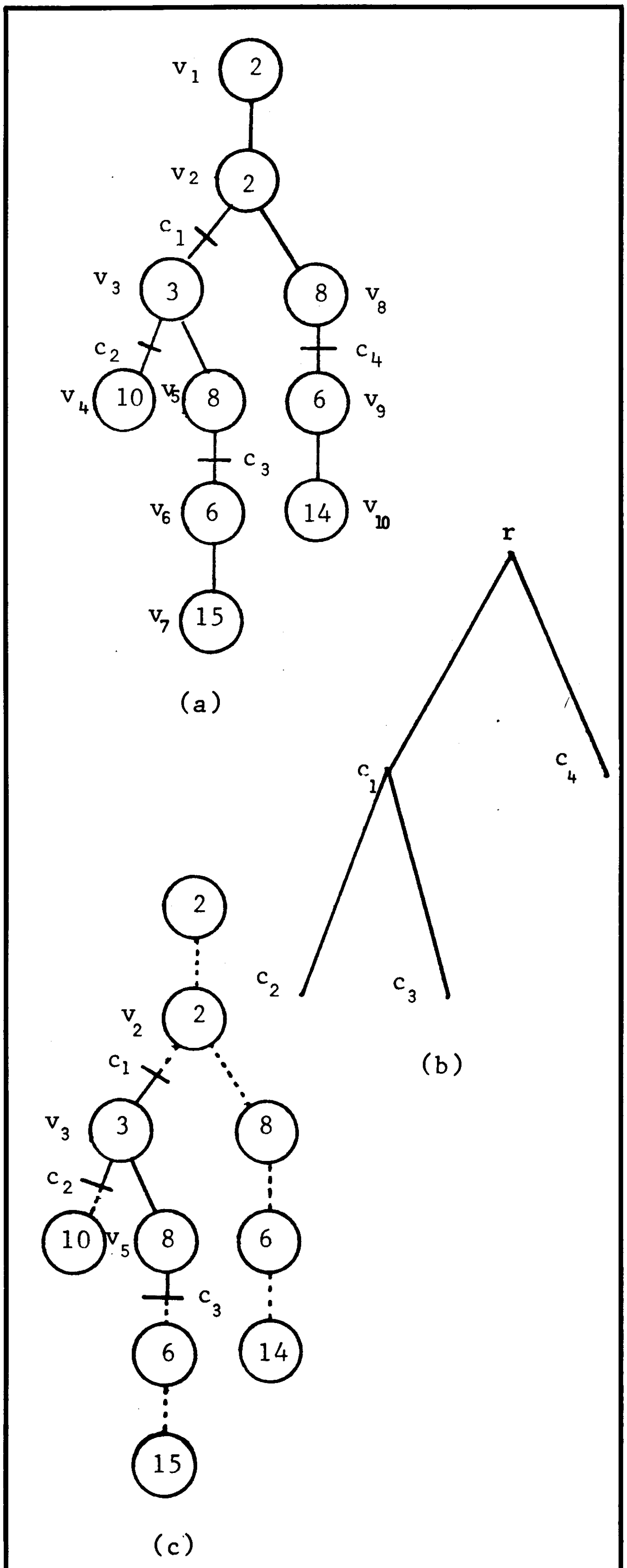

FIGURE 1:
(a) Tree T(k = 4, n = 9), (b) Cut tree C,
(c) Component B = $\{v_3, v_5\}$

(i) the down-component of cut c_1, or
(ii) the down-component of vertex v_3, or
(iii) the down-component of edge($v_2 \rightarrow v_3$), or
(iv) the up-component of cut c_2 (or of cut c_3).

Further, c_1 is the top cut of component B, while c_2 and c_3 are its bottom cuts. The root of B is vertex v_3. The root-component contains v_1, v_2 and v_8.

3. Statement of the Algorithm
The Shifting Algorithm: Min-max q-partition of a tree

1. Place all k cuts on the edge incident with the root. Set BEST_MINMAX_SO_FAR ← ∞, and set BEST_PARTITION_SO_FAR equal to the starting configuration.
2. While the root component is not a heaviest component, perform steps 3, 4 and 5.
3. Find a cut with a heaviest down-component, and down-shift it from its current edge e to a vacant son-edge having heaviest down-component. If no such vacant edge exists then halt.
4. Traverse the path from tail(e) to the root in the bottom-up direction until a vertex v, which is the head of a cut, is encountered. For each vertex w on that path having a cut incident from w, perform the following:
   If the down-component of a cut incident from w is lighter than the down-component of the vacant son-edge e_s of w on the path, then side-shift that cut to edge e_s. If more than one cut incident from w can be side-shifted, choose a cut with a lightest down-component.
5. Set LARGESTWT equal to the weight of the largest component in the current partition A. If LARGESTWT < BEST_MINMAX_SO_FAR then set BEST_MINMAX_SO_FAR ← LARGESTWT, and set BEST_PARTITION_SO_FAR ← A.

We define a *terminating position* to be a partition at which the algorithm terminates. A final value of BEST_PARTITION_SO_FAR is called a *resulting partition* of the algorithm. (The terminating partition is different from the resulting partition if some previous partition had lighter heaviest component than the termination partition.) Figure 2 illustrates the operation of the algorithm.

4. Complexity Analysis of the Algorithm

We start with a general description of the complexity analysis. In addition to the initialization, the algorithm uses two basic operations: down-shifts and side-shifts. We show in [1] Lemma 1 that the total number of down-shifts and side-shifts during the operation of the algorithm is bounded by kh′(T) and k^2h′(T), respectively.

For each down-shift we look for a cut of heaviest down-component, then decide to which edge to shift and finally update the data structures.

Between any two down-shifts at most k-1 side-shifts are considered. For each possible side-shift the down-component of the cut and of a brother edge are compared. Updating the data structures is required whenever a cut is side-shifted.

The complexity of the algorithm will be shown to be $O(k^3rd(T) + kn)$ where rd(T) is the number of edges in the radius of the tree.

Throughout this section the index i, $1 \le i \le n$, labels an edge e_i in the tree T. With regard to the cut tree C, for simplicity we denote the root r by c_0; index t, $0 \le t \le k$ then refers either to a cut c_j or to the root c_o. The index j refers specifically to a cut and not the root; thus $1 \le j \le k$.

Let d(v) denote the out-degree of vertex v in the tree T.

For each vertex v in tree T we compute the weight Z(v) of the complete subtree of T rooted at v, that is to say, the sum of the weights of v and of all its descendents. Z(v) may be computed once and for all by scanning the tree in endorder [3]. As before, if a cut c_j is assigned to an edge e_i, for convenience we shall use $Z(c_j)$ for $Z(head(e_i))$.

During the execution of the algorithm we update the following data structures:

1. For each vertex (cut) c_t in the cut tree C we maintain a list $L(c_t)$ of the sons of c_t in C.
2. We also require a pointer F from each c_j to its father in C.
3. During the execution of the algorithm, for each cut (root) c_t we require the weight $D(c_t)$ of the down-component (root-component) of c_t.

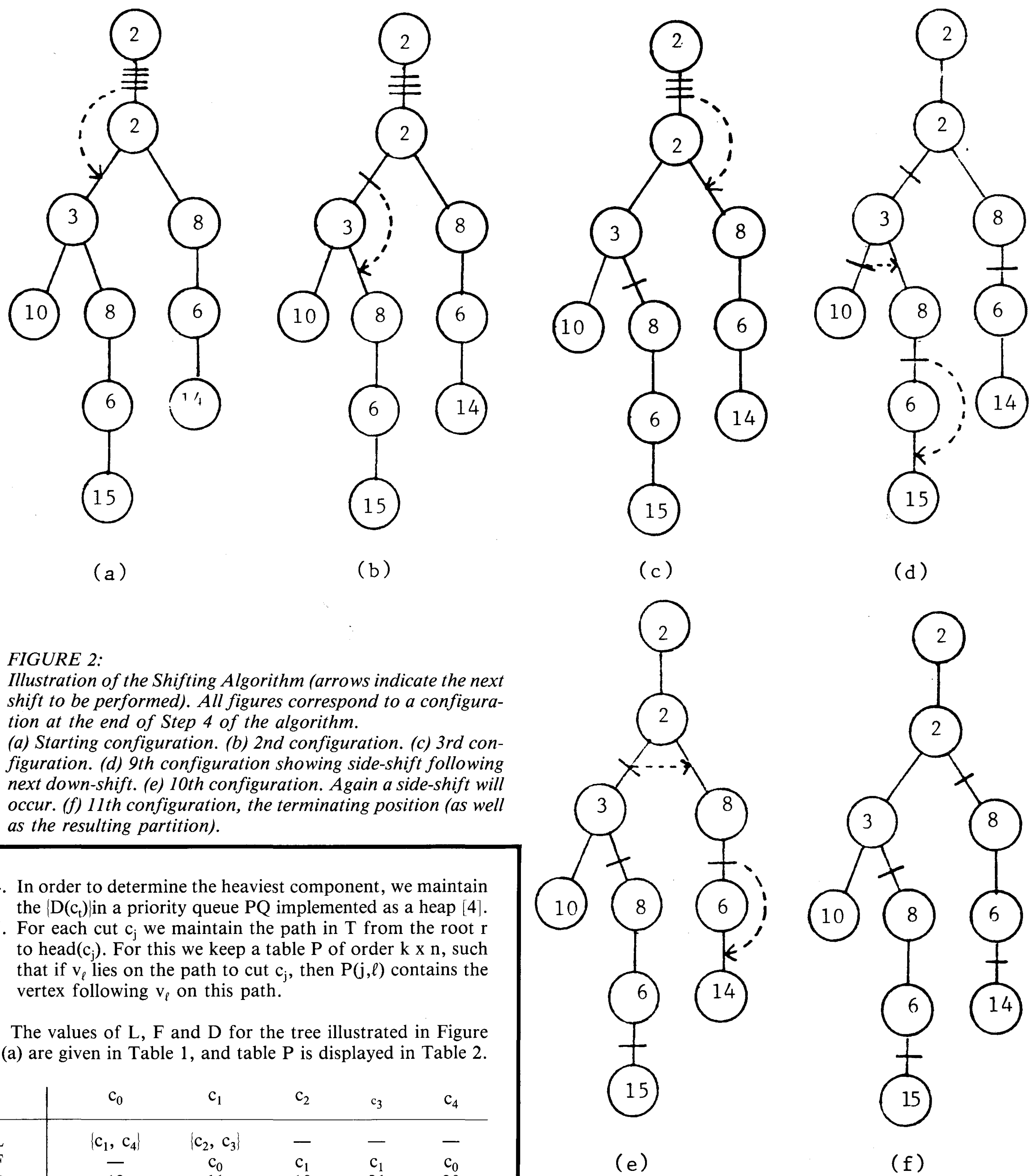

FIGURE 2:
Illustration of the Shifting Algorithm (arrows indicate the next shift to be performed). All figures correspond to a configuration at the end of Step 4 of the algorithm.
(a) Starting configuration. (b) 2nd configuration. (c) 3rd configuration. (d) 9th configuration showing side-shift following next down-shift. (e) 10th configuration. Again a side-shift will occur. (f) 11th configuration, the terminating position (as well as the resulting partition).

4. In order to determine the heaviest component, we maintain the $\{D(c_t)\}$ in a priority queue PQ implemented as a heap [4].
5. For each cut c_j we maintain the path in T from the root r to head(c_j). For this we keep a table P of order k x n, such that if v_ℓ lies on the path to cut c_j, then $P(j,\ell)$ contains the vertex following v_ℓ on this path.

The values of L, F and D for the tree illustrated in Figure 1(a) are given in Table 1, and table P is displayed in Table 2.

	c_0	c_1	c_2	c_3	c_4
L	$\{c_1, c_4\}$	$\{c_2, c_3\}$	—	—	—
F	—	c_0	c_1	c_1	c_0
D	12	11	10	21	20

TABLE 1:
Values of data structures L, F and D for the tree in Figure 1(a).

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
c_1	v_2	—								
c_2	v_2	v_3	—							
c_3	v_2	v_3	v_5		—					
c_4	v_2	v_8						—		

TABLE 2:
Entries of data structure P for the tree in Figure 1(a).

Initialization

The k cuts $\{c_j\}$ are initially assigned to the edge incident with the root c_0; the corresponding cut tree C is therefore a directed chain. The initialization of data structures $\{L\}$ and $\{F\}$ is thus straightforward. With regard to $\{D\}$, initially we set $D(c_0) \leftarrow w(r)$, $D(c_k) \leftarrow W(T) - w(r)$, and $D(c_u) \leftarrow 0$, $1 \le u \le k-1$. The number of operations required for this initialization is clearly $0(kn)$ since all entries of Table P are initially set equal to "undefined".

Down-Shifting

Step 3 of the algorithm requires finding a largest component. A cut c_m for which $D(c_m)$ is maximum can be found in $0(\lg k)$ operations by using the priority queue PQ. If $c_m = c_o$, then the current partition is the terminating partition. Otherwise, we have to consider every possible shift of c_m to each of the vacant edges e_s incident with head (c_m), and then perform that down-shift resulting in a maximum down-component for c_m.

For each such vacant edge e_s we calculate the weight $R(head(e_s))$ of the resulting down component for shifting c_m to e_s by subtracting from $Z(head(e_s))$ the sum $\Sigma Z(head(c_u))$ over all sons c_u of c_m in the cut tree C' obtained from C by shifting c_m to e_s. In fact, we can simultaneously compute all the weights $R(head(e_s))$, $1 \leq s \leq d(tail(e_s))$ as follows. Initially assign $R(head(e_s)) \leftarrow Z(head(e_s))$, requiring $0(d(tail(e_s)))$ operations. Now scan the list $L(c_m)$ of the sons of c_m in C, and for each cut c_u in this list identify the first edge e_s on the path from $head(c_m)$ to c_u using the table P, and subtract from R accordingly. Thus for each cut $c_u \in L(c_m)$, set $R(P(c_u,head(c_m))) \leftarrow R(P(c_u,head(c_m))) - Z(head(c_u))$.

This requires $0(k)$ operations. $O(d(tail(e_s)))$operations are required for selecting the edge e_s to which cut c_m should be shifted. Hence the total number of operations required for one down-shift is $0(k+d(tail(e_s)))$.

Updating after a down-shift

Consider now what updates are required to our data structures following a down-shift of c_m from an edge e to an edge e_s.

For each cut $c_u \in L(c_m)$ which is no longer a son of c_m in the cut tree C', i.e. satisfying $P(c_u,tail(e_s)) \neq head(e_s)$, delete c_u from $L(c_m)$ and add it to $L(F(c_m))$; set $F(c_u) \leftarrow F(c_m)$. The new weight $D(c_m)$ is in fact computed while determining the resulting down-components of c_m. The weight $D(F(c_m))$ is increased by the difference between the old and new values of $D(c_m)$.

Updating the priority queue PQ with the changes in the values of $D(c_m)$ and $D(F(c_m))$ requires at most $0(\lg k)$ operations.

Set $P(c_m,tail(e_s)) \leftarrow head\ (e_s)$.

Hence, at most $0(k)$ operations are required for updating after a down-shift.

Complexity of down-shifting operations

We have shown above that at most $0(k) + O(d(tail(e_s)))$ operations are required to perform a down-shift of a cut to an edge e_s followed by the necessary updates to our data structure.

Now let $f_1\ f_2,\ldots,f_y$ denote the edges through which a cut c was down-shifted during the execution of the algorithm. While performing these shifts of c, at most $0(\Sigma(d(tail(f_q)) + k))$ operations were required. But $\Sigma\ d(tail(f_q))$ is bounded by $h(T)$. Further, the number of edges y in the path is bounded by $h(T) - 1 = h'(T)$ (where $h(T)$ deonotes the height of the tree T). Thus at most $0(n + kh'(T))$ operations are required for the down-shifts of a given cut c, and hence the number of operations required for the down-shifts of all k cuts is at most $0(k^2h'(T) + kn)$.

Side-shifting

After performing a down-shift of a cut c_m we have to consider possible side-shifts along the path R from c_m to the root r (but only up to the first cut, if any, on this path). First we identify which cuts are candidates for side-shifting, then we sort them into bottom-up order, from which the side-shift with smallest down-component at each level can be determined. Travelling up the path R from $tail(c_m)$, we check at each vertex whether the chosen side-shift does, in fact, satisfy step 4 of the shifting algorithm, and if so, perform the side-shift.

Let F_m denote the father of c_m before any side-shifts are performed. Consider the cuts of $L(F_m)$. Every cut $c \in L(F_)$ for which $P(c_m,tail(c))$ is defined is a candidate for a side-shift onto the path from F_m to c_m. Assuming that the vertices of T are assigned indicies of the form $\ell.p$, where $v_{\ell.p}$ is the p'th vertex at level ℓ in T, we may sort the cuts $\{c_u\}$ into a bottom-up order according to the indices of the vertices $\{tail(c_u)\}$. At most $0(k)$ operations are required for identifying which cuts are candidates for side-shifting, and at most $0(k \lg k)$ to sort them into a bottom-up order and then select a cut with smallest down-component at each level.

Now starting at $tail(c_m)$, traverse the path R in the bottom-up direction. At each vertex we check whether the side-shift of cut c_u, say, selected above is in fact valid in terms of step 4 of our algorithm. We must compare $D(c_u)$ with the weight of the down-component resulting from side-shifting c_u to the edge $(tail(c_u),P(c_m,tail(c_u)))$. The latter weight is computed by subtracting from $Z(P(c_m,tail(c_u)))$ the value of $Z(head(c_w))$ for every cut $c_w \in L(F_m)$ for which $P(c_w,P(c_m,tail(c_u)))$ is defined. Thus at most $0(k)$ operations are required for checking the validity of a side-shift at each level.

Updating after a side-shift

We consider now what updates to our data structures are needed after side-shifting a cut c_u (subsequent to a down-shift of cut c_m and the attendent updating).

Each cut $c_w \in L(C_u)$ is transferred from $L(c_u)$ to $L(F_m)$; set $F(c_w) \leftarrow F(c_u) \equiv F_m$. Now move from $L(F_m)$ to $L(c_u)$ each cut $c_v \in L(F_m)$ for which c_u is now its father, i.e. for which $P(c_v,tail(c_u)) = head(c_u)$, and assign $F(c_v) \leftarrow c_u$.

The new weight $D(c_u)$ was in fact computed while examining the validity of side-shifting c_u. The weight $D(F(c_u))$ is decreased by the difference between the new and old values of $D(c_u)$.

Updating the priority queue PQ according to the changes in the values of $D(c_u)$ and $D(F(c_u))$ requires at most $0(\lg k)$ operations.

Set $P(c_u,tail(c_u)) \leftarrow P(c_m,tail(c_u))$.

Hence at most $0(k)$ operations are required to update the data structures after a side-shift has been performed.

Complexity of side-shifting operations

There are at most (k-1) side-shifts possible following on a down-shift. Thus identifying, sorting and selecting candidates for side-shifting followed by the necessary updating requires all told at most $0(k^2)$ operations for all possible side-shifts following any one given down-shift. Since, as shown in [1] Lemma 1, the number of down-shifts is bounded by $kh'(T)$, the side-shifts of the algorithm as a whole require at most $0(k^3h'(T))$ operations.

Complexity of the Shifting Algorithm

Combining the above results for the complexities of down-shifts and side-shifts yields the complexity $0(k^3h'(T) + k^2h'(T) + kn) = 0(k^3h'(T) + kn)$ for the complete algorithm.

This complexity can be improved by a factor of at most 2 by adding to the undirected tree an additional vertex of weight zero incident with the centre of the tree [2] (which can be found in $0(n)$ operations) and using this vertex as the root. For the rooted tree thus obtained the height $h(T)$ of the tree is $1 + rd(T)$ where $rd(T)$ denotes the *radius* of T (see [4]) and $h'(T) = rd\ (T)$.

The resulting algorithm is then of complexity $0(k^3rd((T) + kn)$.

References

[1] R.I. Becker, Y. Perl and S.R. Schach, A shifting algorithm for min-max tree partitioning. J. ACM., vol. 28 (1981), pp 5-15.

[2] F. Harary, *Graph Theory,* (Addison-Wesley, Reading, Mass, 1969).

[3] D.M. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* (Addison-Wesley, Reading, Mass., 1969).

[4] D.M. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* (Addison-Wesley, Reading, Mass., 1969).

[5] Y. Perl and S.R. Schach, Max-min tree partitioning. J. ACM., vol. 29 (1982), pp 58-67.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

*Presented at the second South African Computer Symposium held on 28th and 29th October, 1981.