Quaestiones Informaticae

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

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Editorial Advisory Board
PROFESSOR D. W. BARRON
Department of Mathematics
The University
Southampton SO9 5NH
England

PROFESSOR K. GREGGOR
Computer Centre
University of Port Elizabeth
Port Elizabeth 6001
South Africa

PROFESSOR K. MACGREGOR
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch 7700
South Africa

PROFESSOR M. H. WILLIAMS
Department of Computer Science
Herriot-Watt University
Edinburgh
Scotland

MR P. P. ROETS
NRIMS
CSIR
P.O. Box 395
Pretoria 0001
South Africa

PROFESSOR S. H. VON SOLMS
Department of Computer Science
Rand Afrikaans University
Auckland Park
Johannesburg 2001
South Africa

DR H. MESSERSCHMIDT
IBM South Africa
P.O. Box 1419
Johannesburg 2000

MR P. C. PIROW
Graduate School of Business Administration
University of the Witwatersrand
P.O. Box 31170
Braamfontein 2017
South Africa

Subscriptions
Annual subscriptions are as follows:

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>US</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individuals</td>
<td>R6</td>
<td>$7</td>
<td>£3,0</td>
</tr>
<tr>
<td>Institutions</td>
<td>R12</td>
<td>$14</td>
<td>£6,0</td>
</tr>
</tbody>
</table>

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
Department of Mathematics & Computer Science, Bar-Ilan University

Stephen R. Schach
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^3rd(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 ∈ V. A q-partition of T into q connected components T_1, T_2, ..., T_q is obtained by deleting k = q-1 edges of T, 1 ≤ k ≤ 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 \( \max_i W(T_i) \).

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-completeness 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_i to v_j, denoted by (v_i → v_j), then we will refer to v_i and tail(e) and to v_j 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 and 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_i → v_j) 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_i and to the cut c as incident from v_i.

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_i to a vacant brother edge e_j, and v = tail(e) = tail(e_j).

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 descendants.

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_i is the son of r (of a cut c_j) if there exists a (unique) path from r (from head(c_j)) to tail(c_i) 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_q is lighter than another component T_j (or T_j is heavier than T_q) if W(T_j) < W(T_q).

We illustrate these definitions in Figure 1. Referring to the tree T shown in Figure 1(a), edge(v_1 → v_2) is the father of (v_3 → v_4), and the brother of edge(v_2 → v_9). Cut c_1 can be down-shifted to edge(v_3 → v_4), and side-shifted to (v_2 → v_9). It is incident from vertex v_4. The subtree comprising vertices \{v_2, v_9, v_3, v_4\} 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_9, v_3\} 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.

27
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 $\leftarrow \infty$, 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:
   a. If the down-component of a cut incident from $w$ is lighter than the down-component of the vacant son-edge $e_5$ of $w$ on the path, then side-shift that cut to edge $e_5$. 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 $\leftarrow$ LARGESTWT, and set BEST_PARTITION_SO_FAR $\leftarrow$ $A$.

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 [11 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 with a 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 $O(k^3 rd(T) + kn)$ where $rd(T)$ is the number of edges in the radius of the tree.

Throughout this section the index $i$, $1 \leq i \leq n$, 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 $t$, $0 \leq t \leq k$ then refers either to a cut $c_t$ or to the root $c_0$. The index $j$ refers specifically to a cut and not the root; thus $1 \leq j \leq k$.

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 $e_j$, for convenience we shall use $Z(c_j)$ for $Z(\text{head}(e_j))$.

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

1. For each vertex (cut) $c_j$ in the cut tree $C$ we maintain a list $L(c_j)$ of the sons of $c_j$ 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) $c_j$ we require the weight $D(c_j)$ of the down-component (root-component) of $c_j$. 

(i) the down-component of cut $c_1$, or
(ii) the down-component of vertex $v_3$, or
(iii) the down-component of edge $e_2 \rightarrow v_2$, 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 $v_8$.

Figure 1:
(a) Tree $T(k=4, n=9)$, (b) Cut tree $C$, (c) Component $B = \{v_1, v_3\}$

FIGURE 1:
4. In order to determine the heaviest component, we maintain the \(|D(c)|\) 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 \(k \times n\), such that if \(v_i\) lies on the path to cut \(c_j\), then \(P(i,j)\) contains the vertex following \(v_i\) 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.

<table>
<thead>
<tr>
<th>(c_0)</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(c_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>([c_1, c_4])</td>
<td>([c_2, c_3])</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>F</td>
<td>---</td>
<td>(c_0)</td>
<td>(c_1)</td>
<td>(c_1)</td>
</tr>
<tr>
<td>D</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>21</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(v_1)</th>
<th>(v_2)</th>
<th>(v_3)</th>
<th>(v_4)</th>
<th>(v_5)</th>
<th>(v_6)</th>
<th>(v_7)</th>
<th>(v_8)</th>
<th>(v_9)</th>
<th>(v_{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_1)</td>
<td>(v_2)</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>(c_2)</td>
<td>(v_2)</td>
<td>(v_3)</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>(c_3)</td>
<td>(v_2)</td>
<td>(v_3)</td>
<td>(v_3)</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>(c_4)</td>
<td>(v_2)</td>
<td>(v_3)</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

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

**Initialization**

The \(k\) cuts \(c_i\) 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 \(D(c_0) = w(r)\), \(D(c_i) = w(T) - w(r)\), and \(D(c_j) = 0\), \(1 \leq u \leq k-1\). The number of operations required for this initialization is clearly \(O(\text{kn})\) since all entries of Table \(P\) are initially set equal to "undefined".

**Down-Shifting**

Step 3 of the algorithm requires finding the largest component. A cut \(c_m\) for which \(D(c_m)\) is maximum can be found in \(O(\text{lg} \, k)\) operations by using the priority queue \(PQ\). If \(c_m = c_{\text{opt}}\), 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 \( e_i \) 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 \( e_i \) we calculate the weight \( R(\text{head}(e_i)) \) of the resulting down component for shifting \( c_m \) to \( e_i \) by subtracting from \( Z(\text{head}(e_i)) \) the sum \( \Sigma Z(\text{head}(c_m)) \) over all sons \( c_s \) of \( c_m \) in the cut tree \( C' \) obtained from \( C \) by shifting \( c_m \) to \( e_i \). In fact, we can simultaneously compute all the weights \( R(\text{head}(e_i)) \), \( 1 \leq i \leq \text{siz}(\text{tail}(e_i)) \) as follows. Initially assign \( R(\text{head}(e_i)) \leftarrow Z(\text{head}(e_i)) \), requiring 0(\text{d(tail}(e_i)) \) operations. Now scan the list \( L(c_s) \) of the sons \( c_s \) in \( C \), and for each cut \( c_t \) in this list identify the first edge \( e_i \) on the path from \( \text{head}(c_t) \) to \( c_t \) using the table \( P \), and subtract from \( R \) accordingly. Thus for each cut \( c_t \notin L(c_m) \), set \( R(P(c_t,\text{head}(c_m))) \leftarrow R(\text{P}(c_t,\text{head}(c_m))) - Z(\text{head}(c_t)) \).

This requires 0(k) operations. 0(\text{d(tail}(c_t))) \) operations are required for selecting the edge \( e_i \) to which \( c_m \) should be shifted. Hence the total number of operations required for one down-shift is 0(k + d(\text{tail}(e_i))).

### 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 \( c_t \).

- For each cut \( c_t \notin L(c_m) \) which is no longer a son of \( c_m \), the weight \( D(c_t) \) is in fact computed while determining the resulting down-components of \( c_m \). The weight \( D(c_t) \) is increased by the difference between the old and new values of \( D(c_m) \).

- For each cut \( c_t \notin L(c_m) \), set \( P(c_t,\text{tail}(c_m)) \leftarrow \text{head}(e_i) \).

- 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 + 0(d(\text{tail}(e_i)))) operations are required to perform a down-shift of a cut to an edge \( e_i \) 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(\text{tail}(f_j)) + k) \) operations were required. But shifting \( c \) is bounded by \( h(T) \).

Further, the number of edges \( y \) in the path is bounded by \( h(T) - 1 = h'(T) \) (where \( h(T) \) denotes the height of the tree \( T \)). Thus at most 0(n + \( k h'(T) \)) \) 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(kh'(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-shifting are performed. Consider the cuts of \( L(F_m) \) for which \( P(c_t,\text{tail}(c)) \) is defined is a candidate for a side-shift on to the path from \( F_m \) to \( c_m \). Assuming that the vertices of \( T \) are assigned indicies of the form \( f, p \), where \( v_p \) is the \( p \)th vertex at level \( f \) in \( T \), we may sort the cuts \( c_t \) into a bottom-up order according to the indicies of the vertices (tail\( (c_t) \). 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 \( c_m \), say, selected above is in fact valid in terms of step 4 of our algorithm. We must compare \( D(c_m) \) with the weight of the down-component resulting from side-shifting \( c_m \) to the edge (tail\( (c_m), P(c_m,\text{tail}(c_m)) \)). The latter weight is computed by subtracting from \( Z(\text{tail}(c_m)) \) the value of \( Z(\text{head}(c_m)) \) for every cut \( c_t \notin L(c_m) \) for which \( P(c_t,\text{tail}(c_m)) \) 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 \( c_m \) (subsequent to a down-shift of cut \( c_m \) and the attendant updating).

Each cut \( c_t \notin L(c_m) \) is transferred from \( L(c_m) \) to \( L(F_m) \); set \( F(c_m) \leftarrow F(c_m) \neq F_m \). Now move from \( L(F_m) \) to \( L(c_m) \) each cut \( c_m \notin L(F_m) \) for which \( c_m \) is now its father, i.e. for which \( P(c_m,\text{tail}(c_m)) \) = head\( (c_m) \), and assign \( F(c_m) \leftarrow c_m \). The new weight \( D(c_m) \) was in fact computed while examining the validity of side-shifting \( c_m \). The weight \( D(c_m) \) is decreased by the difference between the new and old values of \( D(c_m) \).

Updating the priority queue \( PQ \) according to the changes in the values of \( D(c_m) \) and \( D(F(c_m)) \) requires at most 0(lg k) operations. Set \( P(c_m,\text{tail}(c_m)) \leftarrow P(c_m,\text{tail}(c_m)) \).

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(kh'(T) + kn).

### Complexity of the Shifting Algorithm

Combining the above results for the complexities of down-shifts and side-shifts yields the complexity 0(kh'(T) + kh'(T) + kn) = 0(kh'(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 \( T \) (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 \( 1 + \text{rd}(T) \) where \( \text{rd}(T) \) denotes the radius of \( T \) (see [4]) and \( h'(T) = \text{rd}(T) \).

The resulting algorithm is then of complexity 0(kh'(T) + kn).

### References

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:


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

The Design Objectives of Quadlisp* .......................................................... 3
S W Postma

A CSP Description of some Parallel Sorting Algorithms* .................. 7
M H Linck

The Design and Microprogrammed Implementation of a Structured
Language Machine.................................................................................... 13
G R Finnie

Micro-Code Implementation of Language Interpreters*......................... 19
P P Roets

An Interactive Graphical Array Trace* ..................................................... 23
S R Schach

An Efficient Implementation of an Algorithm for Min-Max
Tree Partitioning....................................................................................... 27
Ronald I Becker, Yehoshua Perl, Stephen R Schach

The Relative Merits of Two Organisational Behaviour Models for Structuring a Management Information System .............................................. 31
Peter Pirow

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