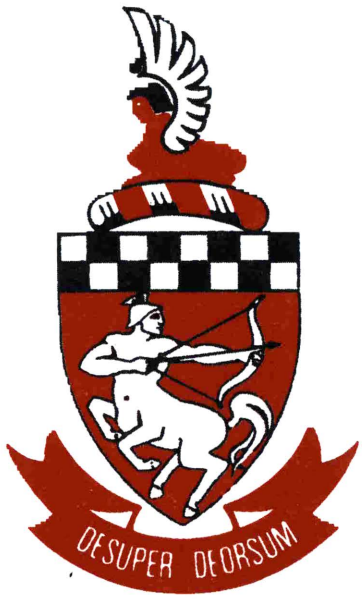
South African Computer Journal

Number 25
August 2000

Suid-Afrikaanse Rekenaar-Tydskrif

Nommer 25
Augustus 2000



The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarvereniging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

World-Wide Web: `http://www.cs.up.ac.za/sacj/`

Subscriptions

	Annual	Single copy
Southern Africa	R80.00	R40.00
Elsewhere	US$40.00	US$20.00

An additional US$15 per year is charged for airmail outside Southern Africa

to be sent to:

Computer Society of South Africa
Box 1714, Halfway House, 1685
Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

In Memoriam:
Stef Postma

South Africa has lost one of her most colourful and eminent computer scientists. Professor Stef Postma passed away peacefully in his sleep on May 5, 2000 after a short illness. He will be remembered for his forthright views and total integrity. Never a man to shy from controversy, he always debated his position with vigour, displaying his extensive vocabulary at every opportunity.

Those who knew him mourn the loss of a very good friend.

Stef was born on August 10, 1938 in Graaff-Reinet and matriculated from Höerskool Linden in Johannesburg. He majored in geology and mathematics at the University of the Witwatersrand and graduated with honours in mathematics from that university. Stef devoted much of his life to promoting computer science as a science and to this end spent a lot of energy and time defining syllabi for undergraduate and honours courses at our universities. He was the prime mover in creating the South African Institute of Computer Scientists and Information Technologists (SAICSIT) in 1982, providing a professional body to represent the interests of local computer scientists. He was also instrumental in establishing Quaestiones Informaticae (now the South African Computer Journal) which afforded South African computer scientists the opportunity to publish papers locally in a refereed journal.

-Doug Laing

Links2Go "Computer Science Journals" Award

The SACJ production editing team received the following unsolicited e-mail:

The page at <http://www.cs.up.ac.za/sacj/>, was selected as a Links2Go "Key Resource" in the Computer Science Journals topic, at <http://www.links2go.com/topic/Computer_Science_Journals>.

How your page was selected:

Each quarter, Links2Go samples millions of web pages to determine which pages are most heavily cited by web pages authors, such as yourself. The most popular pages are downloaded and automatically categorized by topic. At most 50 of the pages related to a topic are selected as "Key Resources." Out of 50 pages selected as Key Resources for the Computer Science Journals topic, your page ranked 19th. For topics like Music, where there are a large number of interested authors and related pages, it is harder to achieve selection as a Key Resource than for a special-interest topic, such as Quantum Physics.

The Links2Go Key Resource award differs from other awards in two important ways. First, it is objective. Most awards rely on hand selection by one or more "experts," many of whom have only looked at tens or hundreds of thousands of pages in bestowing their awards. Selection for these awards means no more than that one person, somewhere, noticed your page and liked it enough to select it. The Key Resource award, on the other hand, is based on an analysis of millions of web pages. Any group or organization who conducts a similar analysis will arrive at similar conclusions. When Links2Go says your page is a Key Resource, we mean that your page is one of the most relevant pages related to a particular topic on the web today, using an objective statistical measure applied to an extremely large data set.

Second, the Key Resource award is exclusive. We get literally hundreds of people requesting that their page be added to one or more topics per week. All of these requests are denied. The only way to get listed as a Key Resource is to achieve enough popularity for our analysis to select your pages automatically. We do not accept fees, offers of link exchanges, free advertising, or bartered livestock as inducements to add new sites to our lists. Fewer than one page in one thousand will ever be selected as a Key Resource.

Once again, congratulations on your award! Links2Go Awards awards@links2go.com

Orthogonal Axial Line Placement in Chains and Trees of Orthogonal Rectangles

ID Sanders DC Watts AD Hall

Department of Computer Science, University of the Witwatersrand

Abstract

Previous research has shown that the orthogonal axial line placement problem for orthogonal rectangles is NP-complete in general but also that there are restrictions of the problem for which polynomial time solutions can be obtained. This article presents algorithms which solve two restricted versions of the orthogonal axial line placement problem – chains and trees of orthogonal rectangles. These restricted versions are slightly more general than the polynomial time versions previously presented.

Keywords: *Computational Geometry, NP-Complete Problems, Covering, Guarding, Orthogonal Rectangles, Axial Lines*
Computing Review Categories: *F.2, F.2.2 (Also G.2.1 and G.2.2)*

1 Introduction

The general axial line placement problem has its origin in the area of town planning and urban design – the idea of *Space Syntax Analysis* [3]. Space Syntax Analysis gives a globalising perspective of town design by determining how easy it is to traverse the town. This analysis is accomplished by the positioning of axial lines on a town plan – the fewest such lines are required.

The axial line placement problem in general is:

> Given a number of adjacent convex polygons find the fewest axial lines (line segments), contained wholly inside the convex polygons, required to cross all of the boundaries shared between adjacent convex polygons. An additional requirement is that each axial line should cross as many of the shared boundaries as possible — a *maximal* axial line.

Depending on how the problem is considered there are two similar but distinct problems which can be addressed — adjacencies can be crossed more than once but every adjacency must be crossed at least once; and any adjacency can only be crossed once.

Sanders *et al.* [6] considered a restriction of the general problem

> Given a number of adjacent orthogonally-aligned rectangles, find the fewest orthogonal axial lines, contained wholly inside the rectangles, required to cross all of the boundaries shared between adjacent rectangles. An additional requirement is that each axial should cross as many of the shared boundaries as possible — a *maximal* axial line.

In the work of Sanders *et al.* [6], adjacencies were allowed to be crossed more than once but every adjacency had to be crossed at least once. Sanders *et al.* [6] showed this version of the problem to be NP-Complete. (Note that at that time the problem was termed "ray guarding" but this was later considered to be misleading nomenclature [4].) In the remainder of this article the problem studied by Sanders *et al.* [6] is called the *orthogonal axial line placement problem for orthogonal rectangles.*

Sanders *et al.* [6] also showed that there are special cases of their problem which can be solved in polynomial time. Specifically they showed that if the union of the adjacent rectangles is itself a rectangle then the axial line placement problem can be solved in linear time by mapping to the problem of finding the minimum cover of an interval graph [2]. This involves projecting all of the adjacencies onto a vertical line segment L to obtain the set of intervals on L. This line L and the overlapping intervals can then be represented as an interval graph and the vertex cover problem for interval graphs can be solved in linear time [2]. The mapping from the axial line placement problem to vertex cover for interval graphs is also possible for other configurations of adjacent rectangles provided that any vertical adjacencies which produce overlapping intervals when projected onto L can be crossed by a horizontal line which does not leave the union of the rectangles [6].

The present article considers algorithms to solve two restricted cases of the orthogonal axial line placement problem for orthogonal rectangles – chains and trees of rectangles (see Section 2 for definitions of these). These problems cannot be solved by a mapping to an interval graph because the layout of the rectangles could lead to adjacencies which cannot be crossed by a single axial line which remains inside the union of the rectangles being mapped to the same interval. These problems can, in fact, be solved using the algorithm developed by Sanders *et al.* [6] (discussed in Section 3) but this algorithm does a lot of unnecessary work in these cases so better ways of solv-

ing these problem are required. In this article an $O(n)$ algorithm for orthogonal axial line placement in chains of rectangles (Section 4) is given and an $O(n^2)$ algorithm for trees of rectangles (Section 5) is presented.

This work could have application in VLSI design – the rectangles as the components to be connected and the axial lines as the connections.

2 Terminology

In this article all rectangles are assumed to be orthogonal – aligned with the cartesian axes. In addition, this article is only concerned with placing horizontal lines segments through shared vertical edges of rectangles. The problem of vertical lines and horizontal adjacencies can clearly be solved in the same way.

In this research an axial line is defined by a range of y-values through which a line segment parallel to the x-axis can be drawn. An axial line thus actually defines a set of possible line segments. An axial line crossing a given adjacency would be defined by the y-value range of that adjacency and the two rectangles involved. An axial line crossing the adjacencies between a number of rectangles would be given by the common y-value range of the adjacencies between the rectangles and a list of the rectangles concerned.

A chain of rectangles is defined as any collection of rectangles where every rectangle is horizontally (vertically) adjacent to at most one other rectangle at each end. An example of a chain is shown in Figure 1.

A tree is defined as a number of adjacent orthogonally aligned rectangles, where each rectangle is joined on the left (right) end at most one rectangle and on the right (left) by zero or more rectangles. A tree is thus a generalisation of a chain – each branch of the tree can be considered as a chain of rectangles. An example of a tree of rectangles is shown in Figure 2.

3 The naive algorithm

Sanders *et al.* [6] presented an $O(n^2)$ algorithm to return a non-redundant set of maximal orthogonal axial lines for the problem of placing orthogonal axial lines to cross the adjacencies between orthogonal rectangles. This algorithm has four phases (after determining which rectangles are adjacent). It generates all the possible straight lines which cross the adjacencies between rectangles; it determines which lines are essential (i.e. are the only lines which cross a particular adjacency); it removes any lines which only cross adjacencies crossed by the essential lines (redundant lines); and then it resolves the choice conflict. The resolving of the choice is done by repeatedly choosing the choice line which crosses the highest number of previously uncrossed adjacencies. Each phase is $O(n^2)$ in the worst case.

This algorithm can be applied directly to place the minimal number of axial lines in chains and trees of orthogonal rectangles but it does more work than is necessary – it generates lines which are redundant and then have to be removed to get a minimal set of maximal lines. This is shown in Figure 3 where 5 lines are generated in the first part of the algorithm but only 2 lines are actually needed – the lines *a–b–c–d–e–f* and *e–f–g–h–i–j*. The reason that these extra lines are generated is that the algorithm starts from the left and tries to extend any existing line into any rectangle adjacent to the current one. If this can be done it is but if it cannot be done then a new line is created crossing from the current rectangle into the next rectangle and this new line is then extended as far as it can be to the left before the algorithm continues working towards the right. In Figure 3 the line *a–b–c–d–e–f* cannot be extended into *g* so a new line *f–g* is created and this is extended back as far as possible giving the line *b–c–d–e–f–g*. Similarly for the other lines shown. Phase 3 of the algorithm would identify the two lines which have to be in the solution – the line *a–b–c–d–e–f* is the only line to cross the adjacency between rectangles *a* and *b* and the line *e–f–g–h–i–j* is the only line to cross the adjacency between *i* and *j*. These lines are thus essential. The unnecessary lines would then be removed from the final set of lines in phase 3 of the algorithm. Phase 4 of the algorithm would be unnecessary for both chains and trees of rectangles as no choice lines will ever be generated.

The remainder of this article looks at algorithms which are more efficient for solving this problem in the case of chains and trees of orthogonal rectangles. Section 4 gives an $O(n)$ algorithm for placing orthogonal axial lines in chains of orthogonal rectangles and Section 5 gives an $O(n^2)$ algorithm for placing orthogonal axial lines in trees of orthogonal rectangles.

4 Orthogonal axial line placement in chains of rectangles

In this section of the article an algorithm is presented to solve the orthogonal axial line placement problem in a chain of orthogonal rectangles. The algorithm is shown in Figures 4 and 5.

The algorithm takes as input a set of rectangles which are known to represent a chain. Each rectangle is defined by the coordinates of its bottom left and top right corner. The data structure used is an array of records `Rect[]`. Here each record has fields for `left`, `right`, `top` and `bottom` to represent the rectangle's coordinates (minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, maximum y-coordinate) and `adjtop` and `adjbottom` to represent lowest and highest y-values of the range of y-values which defines the adjacency with the next rectangle in the chain. The algorithm sorts these rectangles according to the x-coordinate of the left edge of each rectangle. This first stage is dominated by the sorting and thus has a complexity of $O(n \lg n)$. Clearly this stage is unnecessary if the rectangles are given in sorted order as a chain of rectangles.

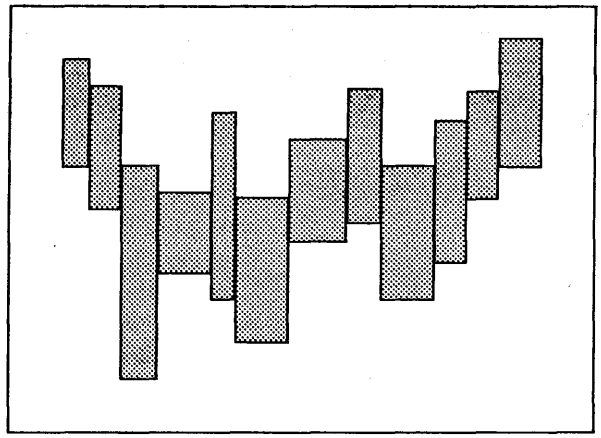


Figure 1: An example of a chain

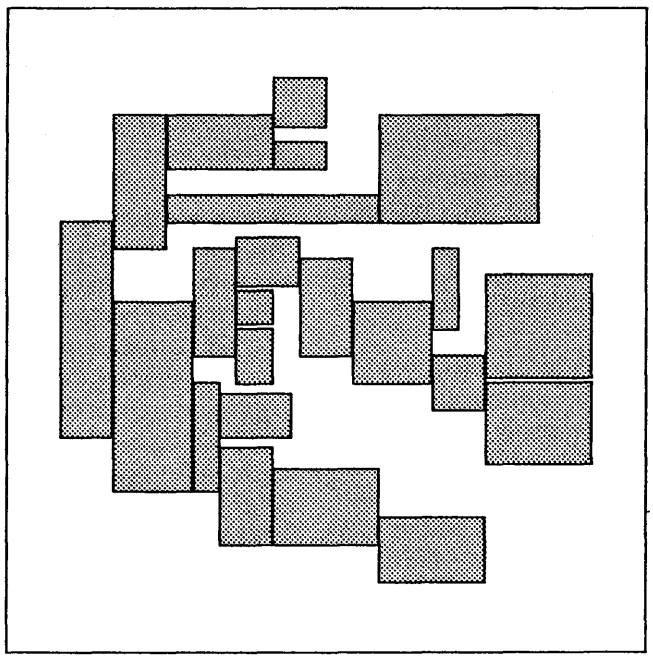


Figure 2: An example of a tree of rectangles

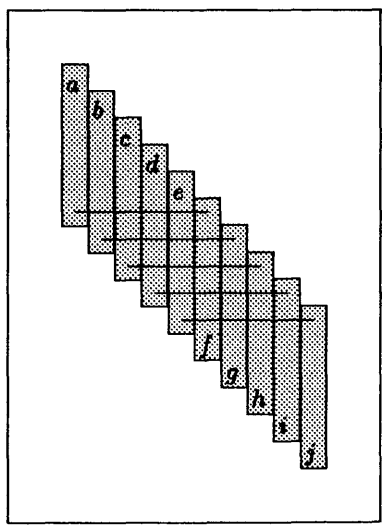


Figure 3: A case where more axial lines than necessary are generated

```
    // Stage 0:
    // --------
    // Get input
0   create array Rect[ ] of all given rectangles
    // n is the number of rectangles
1   for i from 1 to n
2      Input Rect[i].left
3      Input Rect[i].right
4      Input Rect[i].top
5      Input Rect[i].bottom
6      Set Rect[i].adjbottom to be undefined          // will be calculated
7      Set Rect[i].adjtop to be undefined

    // Stage 1:
    // --------
    // Define order of rectangles in the chain
8   sort Rect[ ] according to left value of each rectangle
    //          (i.e. based on Rect[].left)

    // Stage 2:
    // --------
    // Determine the extent of the adjacency between each
    // rectangle and its right neighbour.
    // Note: minimum and maximum are functions which return the smaller
    //       of two numbers and the greater of two numbers respectively
9   for i from 1 to n-1
10     Rect[i].adjtop := minimum(Rect[i].top, Rect[i+1].top)
11     Rect[i].adjbottom := maximum(Rect[i].bottom, Rect[i+1].bottom)

    // Stage 3:
    // --------
    // Determining set of forward lines

    // ForwardLines is a list of the lines which have been found in this stage

    // Starting the forward sweep by initialising the smallest common adjacency
    // This is called CurrentAdj
    // Start off by using the adjacency between rectangle 1 and rectangle 2

    // Currentline is a list of adjacencies which are crossed by the line being worked on

12  currentAdj.top := Rect[1].adjtop
13  currentAdj.bottom := Rect[1].adjbottom
14  Add adjacency 1|2 to the list CurrentLine

15  for i from 2 to n
16     if (Rect[i].adjtop < CurrentAdj.bottom) OR (Rect[i].adjbottom > currAdj.top)
17        then
18           Add the list CurrentLine to the end of list ForwardLines
19           currentAdj.top := Rect[i].adjtop
20           currentAdj.bottom := Rect[i].adjbottom
21           Set CurrentLine to be empty
22        else
23           if (Rect[i].adjtop < currAdj.top) then currAdj.top := Rect[i].adjtop
24           if (Rect[i].adjbottom > currAdj.bottom) then currAdj.bottom := Rect[i].adjbottom
25           Add the adjacency i|i+1 to the end of list CurrentLine
26  If CurrentLine is not empty
27     then
28        Add the list CurrentLine to the end of list ForwardLines
```

Figure 4: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stages 0 to 3

From this sorted list of rectangles the right adjacencies of each rectangle are found – lines 9 to 11 of the algorithm – by comparing the y-coordinate range of any rectangle with the y-coordinate range of its neighbour on the right – the largest bottom y-value and the smallest top y-value define this adjacency. This information is stored in `Rect[i].adjtop` and `Rect[i].adjbottom` for rectangle `i`. This process takes linear time and forms the second stage of the algorithm.

The third stage involves determining the lines that move forward through the chain. This stage of the algorithm proceeds by traversing the chain of rectangles from left to right keeping track of any common range of y-values (`CurrentAdj`) of the adjacencies which have been con-

```
    // Stage 4:
    // --------
    // Determining set of reverse lines to be stored in list ReverseLines

29  Set CurrentLine to be empty
30  currAdj.top := Rect[n-1].adjtop
31  currAdj.bottom := Rect[n-1].adjbottom
32  Add the adjacency n-1|n to the end of list CurrLine

33  for i from n-2 downto 1
34     if (Rect[i].adjtop < currAdj.bottom) OR (Rect[i].adjbottom > currAdj.top)
35        then
36           Add the list CurrLine to the front of list ReverseLines
37           currentAdj.top := Rect[i].adjtop
38           currentAdj.bottom := Rect[i].adjbottom
39           Set CurrLine to be empty
40        else
41           if (Rect[i].adjtop < currAdj.top) then currAdj.top := Rect[i].adjtop
42           if (Rect[i].adjbottom  currAdj.bottom) then currAdj.bottom := Rect[i].adjbottom
43           Add the adjacency i|i+1 to the front of list CurrentLine
44  If CurrentLine is not empty
45     then
46        Add the list CurrentLine to the end of list ForwardLines

    // Stage 5:
    // --------
    // Merge the lines to get the final set of lines
47  Set m to the number of lines in the ForwardLines or ReverseLines (these will be equal)
48  for i from 1 to m
49     set FinalLines[i] to be empty
50     while ForwardLines[i] is not empty AND ReverseLines[i] is not empty
51        If (the first adjacency in ForwardLines[i] is to the left of
             the first adjacency in ReverseLines[i])
52           then
53              remove first adjacency from ForwardLines[i]
54              add this adjacency to end of FinalLines[i]
55           else
56              if the adjacencies are the same
57                  then
58                      remove first adjacency from ForwardLines[i]
59              remove first adjacency from ReverseLines[i]
60              add this adjacency to end of FinalLines[i]
61     if ForwardLines[i] is not empty then add all remaining adjacencies to FinalLines[i]
62     if ReverseLines[i] is not empty then add all remaining adjacencies to FinalLines[i]
```

Figure 5: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stages 4 and 5

sidered so far. The existence of such a range implies that an axial line could be placed to cross the adjacencies which share this range. The algorithm also keeps track of which adjacencies between rectangles could be crossed by such a line by maintaining a list of these adjacencies (`CurrentLine` is the list of adjacencies which share a common rage of y-values and each adjacency is given by the rectangles in it in the form `left|right`). The common range starts out as the range of y-values of the adjacency between the first and second rectangles (see lines 12 and 13 of the algorithm). `CurrentLine` is initialised to `1|2` (line 14) as rectangle 1 is adjacent to rectangle 2 and a line could be drawn crossing the adjacency between them. This range and the current line are then updated when the next adjacency is encountered (see lines 15-25). If there is an overlap in y-values between the common range and the next adjacency then the current line can be extended (lines 23-25). If there is no overlap then a new line must be started (18 to 21). In this case the common range is reset as the range of the adjacency being considered. Lines 16 to 25 are repeated until the end of the chain is reached i.e. until all the adjacencies have been crossed by a line in the set of forward lines. Lines 26 to 28 make sure that the last line is also added to the set of forward lines. Since each rectangle is visited once by a single line, this stage of the algorithm also takes linear time.

The fourth stage is the same the third stage, except one finds the reverse lines by moving from right to left, instead of left to right (see lines 29 to 46 which are very similar to lines 12 to 28). Note that the set of reverse lines are still arranged so that the leftmost lines come first in the ordering. The adjacencies in any line are also arranged in order from left to right. Clearly the complexity for this stage is the same as the previous stage.

Lastly the forward lines and the reverse lines are merged together to obtain the *maximal* lines which cross every adjacency. This merging is accomplish by noting that any forward line crosses each adjacency that it can ex-

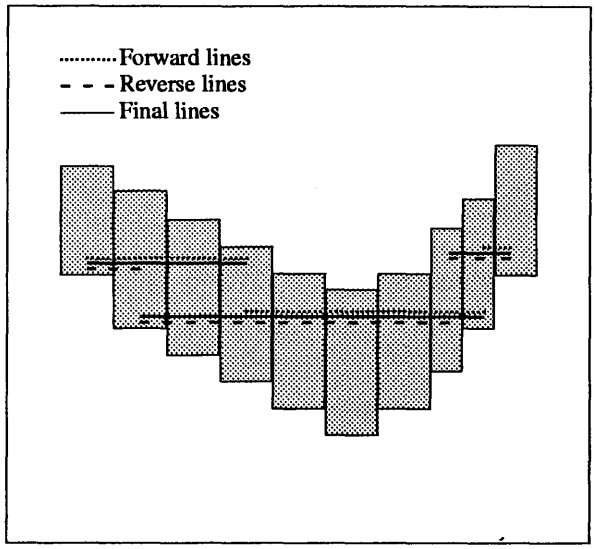


Figure 6: A chain of orthogonal rectangles showing the forward and reverse lines and the final maximal lines.

actly once and extends as far as possible to the right. The process of placing the forward lines thus generates a set containing the minimum number of *non-maximal* lines to cross all of the adjacencies in the chain – any fewer lines and some adjacencies would be left uncrossed, any more lines and one or more adjacencies would have to be crossed more than once. Similarly any reverse line crosses each adjacency that it can exactly once and extends as far as possible to the left. The set of reverse lines is also the minimum number of non-maximal lines required to cross all of the adjacencies. Clearly the two sets must contain the same number of lines. As each forward line extends as far as possible to the right, the forward lines will thus define the rightmost end of some maximal line. Similarly the reverse lines extend as far as possible to the left and define the leftmost extent of the maximal lines. Thus merging any forward and reverse lines which have ranges of y-values which overlap and which cross some common adjacencies will result in a maximal line, through the resulting overlap of y-values, which crosses all the adjacencies crossed by both lines.

The sets of forward and reverse lines are arranged from leftmost to rightmost starting x-value. The merging begins by considering the first (leftmost) lines in each set and generating a new line which crosses all of the adjacencies crossed by those two lines. No other lines need to be considered as no other lines will have an overlap of y-values and share some common adjacencies. Lines 47 to 62 show the detail of how this is accomplished. The merging begins by considering the first adjacency in `ForwardLines[1]` and comparing it with the first adjacency in `ReverseLines[i]` (line 51). The leftmost of these adjacencies (found by looking at the x-coordinates of the rectangles concerned) gives the leftmost adjacency in the new final lines. This adjacency is removed from the list that it occurred in (both if this was the case) and added to final line which is being created (lines 53 and 54 or lines 56 to 60). The process is the repeated with the first adjacencies of the two new lists. If either list becomes empty then the remaining adjacencies in the non-empty list are copied to the final line which is being built up (lines 61 and 62). This new line is then the first line in the final set of lines – it is maximal because it extends as far as it can to the left and to the right. The other lines in the forward and reverse sets are handled in a similar fashion to produce final lines. An example of a chain of rectangles with forward lines, reverse lines and the resulting final lines is shown in Figure 6. This final stage of merging the forward and reverse lines takes $O(n)$ time to complete – each adjacency can appear once in a forward line and once in a reverse line so at most $2n$ adjacencies will be considered for addition into one of the final lines.

The complexity of the entire algorithm is thus $O(n)$.

Empirical tests were done on some different configurations of chains of rectangles [8]. The data for the running time of the algorithm, excluding the sorting done in the first stage, verified the theoretical analysis – that is, the data suggested that the last four stages of the algorithm are indeed linear.

This restricted instance of the problem is, in fact, a special case of the problem to be considered in the next section of the article but it was presented as a separate problem in order to make the algorithm presented in the next section easier to understand.

5 Orthogonal axial line placement in trees of rectangles

5.1 The Algorithm

The algorithm to find the minimal set of orthogonal axial lines to cross the adjacencies in a tree of orthogonal rectangles is presented in Figures 7, 8 and 9. The algorithm takes as input a list of rectangles which represents a tree of

orthogonal rectangles and produces a minimal set of maximal orthogonal axial lines.

The algorithm is split into six stages: inputting the rectangle data, finding the adjacencies between the rectangles; defining the order in which the rectangles will be visited; finding the forward lines; finding the leaf lines; and merging the forward lines and leaf lines into the final lines.

The main data structure in the algorithm (as given in Figure 7) is an array `Rect` of records to represent the rectangles. Each rectangle is represented by a record with 8 fields – `left`, `right`, `top` and `bottom` to define the rectangle, `parent` to keep track of the rectangle to the left of the current rectangle, `numadj` to keep track of the number of rectangles adjacent to the right end of any rectangle, `adjlist` which is a list of these rectangles and finally `LeafLineNo` which is used in stages 4 and 5 to keep track of which leaf line crosses the rectangle.

The adjacencies between the rectangles are found using the algorithm presented by Sanders *et al.* [5]. In this stage of the algorithm `parent`, `numadj` and `adjlist` are given values.

In stage 2 of the algorithm an additional array, `RightList[]`, is created which stores a list of the indices of the array `Rect[]` arranged according to the x-coordinate of the right end of the rectangles. This list defines the order in which the rectangles will be visited in stage 3 – finding the forward lines. Stages 0 to 3 are essentially preprocessing to define a tree of orthogonal rectangles.

Stage 3 of the algorithm (as given in Figure 8) finds the *forward lines*. These are the lines found by starting at the root rectangle (in this case the leftmost rectangle – with the smallest x-coordinate of its right edge) and working towards the leaf rectangles of the tree (these are rectangles with no right neighbours) considering each rectangle in turn. An interval tree [1] is used to maintain the y-value ranges which can be considered at any stage. An interval tree is a red-black tree where each vertex x contains an (open or closed) interval defined by its low and high endpoints and where the *key* of the vertex is the low endpoint. Insertions into the tree are based on the low endpoint of the interval to be inserted. Thus an inorder traversal of an interval tree would return the intervals in sorted order by low endpoint. In the algorithm above every vertex in the interval tree stores an interval defined by the variables `low` to `high` and, in addition, a candidate line (`line`) represented by a list of the rectangles such a line could cross. Each vertex in the interval tree thus represents a line which might need to be considered when attempting to extend lines from the root to the leaf rectangles. Lines 12 to 14 initialise the root vertex of the interval tree to contain the interval represented by the y-value range of the root rectangle and a line which crosses only that rectangle but could be extended into the root rectangle's right neighbours.

After the initialisation each rectangle is considered in turn (lines 15 to 36 of Figure 8). Each such rectangle can have at most one rectangle adjacent to it on the left and could have a number of other rectangles adjacent to it on the right. If it has no rectangles adjacent to it on the right then it is a leaf rectangle and its number is added to a list of such leaf rectangles (line 17 to 19). If it has adjacent rectangles on the right then the forward line coming into the rectangle could potentially be extended into these adjacent rectangles (lines 21 to 36). The algorithm searches the interval tree for the line which comes into the rectangle from the left – it will have an interval which has some overlap with the y-value range of the rectangle and the last rectangle through which the line passes will be the current rectangle (line 21). The algorithm then considers each right neighbour of the current rectangle in turn (lines 22 to 35) and determines which adjacencies can be crossed by extensions of this line. There could be more than one possible extension of the line depending on the y-coordinates of the adjacencies being considered and thus the one line coming into a rectangle from the left could become more than one line (see lines 26 to 30) – one line going into each adjacent rectangle where there is an overlap of the interval covered by the line and bottom and top y-values of the adjacent rectangle. Each of these new lines would result in a new vertex being inserted into the interval tree to store the new interval and detail which rectangles are crossed by the new line. The algorithm also determines for which adjacencies (if any) new forward lines have to be started (lines 32 to 35), i.e. the existing line cannot be extended to cross the new adjacencies (there is no overlap of y-coordinates). Again new vertices are created in the interval tree.

After all the right neighbours have been considered then the vertex in the interval tree representing the line coming into the current rectangle is deleted if the line has been extended (it is no longer needed in this case). If the line has not been extended then it is one of the forward lines and is thus not deleted. The last step in this stage of the algorithm (lines 37 to 42) is to traverse the interval tree and produce a list of the forward lines, represented by a range of y values and a list of rectangles crossed, to cross all of the adjacencies in the tree of rectangles.

Figure 10 shows the forward lines generated for part of the tree of Figure 2. Note that the line which crosses the adjacencies between rectangles a and b and b and c can be extended in d but not into e. A new forward line is created to cross the adjacency between rectangles c and e. This line can be extended into f but not into g.

Stage 4 of the algorithm (as given in Figure 9) works from the leaves of the tree to the root rectangle, hence finding the *leaf lines*. This case is easier than that of finding the forward lines. It is only necessary to check if one of the lines coming into any rectangle from the right can be extended to cross the one adjacency on the left or whether a new leaf line must be created. The algorithm considers each leaf rectangle (from line 19 in stage 3) in turn (lines 43 to 70). The leaf line for any leaf rectangle is first initialised (lines 44 to 48) – this involves finding the interval to be considered, finding the parent of that rectangle and setting the first rectangle in the currentline. The algorithm then works up the rectangle tree until the root rectangle is reached (lines 48 to 68). For each rectangle on the path

```
    // Stage 0:
    // --------
    // Get input
0   create array Rect[ ] of all given rectangles
    // n is the number of rectangles
1   for i from 1 to n
2      Input Rect[i].left
3      Input Rect[i].right
4      Input Rect[i].top
5      Input Rect[i].bottom
7      Set Rect[i].parent to be undefined       // will be calculated
8      Set Rect[i].numadj to be undefined       // will be calculated
9      Set Rect[i].adjlist to be Nil            // will be calculated
10     Set Rect[i].LeafLineNo to be undefined   // used in stage 4 and 5

    // Stage 1:
    // --------
    // Find adjacencies
    // This is done using the algorithm published by Sanders et al.
    // Rect[i].parent, Rect[i].numadj and Rect[i].adjlist are calculated here

    // Stage 2:
    // --------
    // Define order of rectangles in which the rectangles will be visited
11  create array RightList[ ] according to right value of each rectangle
    //         (i.e. based on Rect[].right)
    //This will be an array of the numbers of the rectangles in the
    order which they will be visited.
```

Figure 7: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stages 0 to 2

from leaf rectangle to root rectangle it determines if the leaf line coming into a rectangle can be extended into the current rectangle's parent (lines 53 to 57) or if the current leaf line must be terminated and a new leaf line started (lines 59 to 67). This stage of the algorithm also records the number of the last leaf line which crosses from the current rectangle into its parent rectangle (`LeafLineNo` – line 68) – this could be a new leaf line (starts in that rectangle) or one which crosses the rectangle from right to left. This information is used in stage 5 of the algorithm which is described below. Figure 10 shows the leaf lines generated for part of the tree of Figure 2. Note that the leaf line from rectangle g cannot be extended from e into c and a new "leaf" line has to be created here.

The final stage of the algorithm (stage 5 as given in Figure 9) merges the forward and leaf lines to produce the set of maximal *final lines* which cross all of the adjacencies in the tree of rectangles. Section 4 explains why these lines need to be merged in order to obtain maximal axial lines. The merging here is necessarily more complicated than when working with chains of rectangles because any rectangle could be crossed by more than one forward line and more than one leaf line. Here the algorithm considers each forward line in turn (lines 71 to 84 in Figure 9). A new final line is created for each forward line (lines 73 to 75) and the algorithm then considers each rectangle in the forward line in turn starting from the rightmost end of the line – as before forward lines define the rightmost extent of any final line (lines 76 to 86). The leaf line which corresponds to the forward line is found by considering the leaf line (determined in stage 4, line 68) associated with each of the rectangles which make up the forward line in turn, until a leaf line which has an overlapping range of y-values is found. This leaf line then defines the leftmost extent (leaf lines are always extended as far as possible to the left). The final line is created by using the adjacencies from the forward line and its associated leaf line – the adjacencies from the forward line are used (lines 77 and 78) to define the right end of the final line and when the associated leaf line is found its adjacencies are used to define the left extent of the line (lines 81-84). Figure 10 shows the final lines (and forward and leaf lines) generated for part of the tree of Figure 2. The forward line c–e–f and the leaf line b–c–e which together result in the final line $b-c-e-f$ are an example of how the merging works. Rectangle f is considered first, the leaf line through f does not have an overlapping y-value range so rectangle e is considered. Here the leaf line which was stored in stage 4 of the algorithm is the line that starts in e and crosses into c. This line does have an overlap so the final line can be generated using the forward line and this leaf line.

This example also illustrates why it is sufficient in stage 4 (line 68) of the algorithm to record only the last leaf line which either starts in or crosses across a given rectangle. Leaf lines define the left extent of any final line. If two or more leaf lines start in or cross a given rectangle then the associating of a leaf line and a forward line will not be done when that rectangle is considered. In Figure 2, rectangle c has 2 leaf lines which cross it – the line a–b–c–d and the line b–c–e. These leaf lines are paired with

```
    // Stage 3:
    // --------
    // Find forward lines

    // interval(x,y) defines a range of y values between the two arguments
    // overlap(interval1, interval2) returns true if the two intervals have an
    //         overlapping range of y-values

    // note that || is used to denote concatenation

    // Set values in root vertex of Interval tree
12  high := Rect[RightList[1]].top
13  low := Rect[RightList[1]].bottom
14  line := RightList[1]
15  for p := 2 to n do                          // traverse rectangles in order of right edge
16     k := RightList[p]
17     if Rect[k].numadj = 0
18        then
19           add k to LeafList
20        else
21           find vertex z in interval tree such that
                  overlap(interval(z.low, z.high), interval(Rect[k].bottom, Rect[k].top)) = true
                  AND last rectangle in z.line = k
22           for each rectangle r in Rect[k].adjlist do
23              set extended to be false
24              if overlap(interval(z.low, z.high), interval(Rect[r].bottom, Rect[r].top)) = true
25                 then
    //                Extend an existing line
26                    Insert a new vertex in interval tree with
27                       high := top of overlap
28                       low := bottom of overlap
29                       line := z.line || r
30                    set extended to be true
31                 else
    //                Start a new line
32                    Insert a new vertex in interval tree with
33                       high := Rect[r].top
34                       low := Rect[r].bottom
35                       line := r
36           if extended = true then delete vertex z
    // traverse interval tree to produce a list of forward lines
37  i := 0
38  for each vertex z in the interval tree
39     i := i + 1
40     ForwardLines[i].rects := z.line
41     ForwardLines[i].top := z.high
42     ForwardLines[i].bottom := z.low
```

Figure 8: The algorithm for placing orthogonal axial lines in trees of orthogonal rectangles – Stage 3

their forward lines when rectangle d of forward line a–b–c–d and rectangle e of forward line c–e–f respectively are being considered. Similar arguments apply for other combinations of more than one leaf line either crossing or starting in a given rectangle.

Sanders *et al.* [7] show another example of the applying the algorithm to a specific tree.

5.2 The algorithm complexity

Stage 1 and 2: Find adjacencies and define order The algorithm used to find the adjacencies of the rectangles in stage 1 of the algorithm was developed by Sanders *et al.* [5]. The algorithm also determines the array `RightList`. This algorithm is dominated by sorting and thus has a complexity of $O(n \lg n)$ where n represents the number of rectangles in the tree.

Stage 3: Find forward lines In essence this phase of the algorithm determines which adjacencies overlap with intervals which have already been inserted into the interval tree – an interval in the interval tree means that there is a forward line which can be considered. There are $n - 1$ adjacencies in a tree of n rectangles and each adjacency only needs to be considered once – only one forward line can cross any adjacency. Also each interval tree operation (adding or deleting here) takes $O(\lg n)$ time [1]. Therefore stage 3 of the algorithm has a complexity of $O(n \lg n)$.

This stage of the algorithm could be done more efficiently as regards time by noting that each rectangle can

```
    // Stage 4:
    // --------
    // Find leaf lines

43  for each leaf do
44     k := RectNo of current leaf
45     CurrentT := Rect[k].top
46     CurrentB := Rect[k].bottom
47     p := Rect[k].parent
48     currentline.rects := k
49     i := 1
50     while p is still defined
51        if overlap(interval(Rect[p].top , Rect[p].bottom),
                     interval(CurrentT, CurrentB)) = true
52           then
    //        extend an existing leaf line
53             currentline.rects := p || currentline.rects
54             currentline.top := top of overlap
55             currentline.bottom := bottom of overlap
56             CurrentT := top of overlap
57             CurrentB := bottom of overlap
58           else
    //        add current line to set of leaf lines
59             LeafLines[i].rects := currentline.rects
60             LeafLines[i].top := currentline.top
61             LeafLines[i].bottom := currentline.bottom
    //        start a new leaf line
62             i := i + 1
63             currentline.rects := p
64             currentline.top := Rect[p].top
65             currentline.bottom := Rect[p].bottom
66             CurrentT := Rect[p].top
67             CurrentB := Rect[p].bottom
68        Rect[k].LeafLineNo := i
69        k := p
70        p := Rect[k].parent

    // Stage 5:
    // --------
    //  Merge forward lines and leaf lines to produce final lines

71  Set m to the number of lines in ForwardLines
72  for i from 1 to m
73     FinalLines[i].top := ForwardLines[i].top
74     FinalLines[i].bottom := ForwardLines[i].bottom
75     set FinalLines[i].line to be empty
76     for each rectangle j in ForwardLines[i].line from right end to left end
77        FinalLines[i].line := j || FinalLines[i].line
78        k := Rect[j].LeafLineNo
79        if overlap(interval(ForwardLines[i].top, ForwardLines[i].bottom),
                     interval(LeafLines[k].top,LeafLines[k].bottom)) = true
80           then
81             FinalLines[i].top := top of overlap
82              FinalLines[i].bottom := bottom of overlap
83              Add all rectangles in LeafLines[k].line to FinalLines[i].line
84              Break out of for loop
```

Figure 9: The algorithm for placing orthogonal axial lines in chains of orthogonal rectangles – Stages 4 and 5

only have one line coming into it from the left and using a matrix of size $n-1$ to store the y intervals of these lines. This has the advantage of direct lookup but the disadvantage of always requiring $O(n)$ space – the best case for the interval tree could be much less.

Stage 4: Find leaf lines In this phase of the algorithm the lines starting in the leaves of the tree are extended back towards the root of the tree. Here testing whether the line can be extended is $O(1)$ (a simple test for overlap) but it is possible that there are $O(n)$ leaves and that for each of these leaves it is necessary to extend the line through $O(n)$ rectangles (a tree with approximately $n/2$ leaves and of height also approximately $n/2$ – see Figure 11 for a simple example of this case). So the complexity for this part of the algorithm is $O(n^2)$ in the worst case.

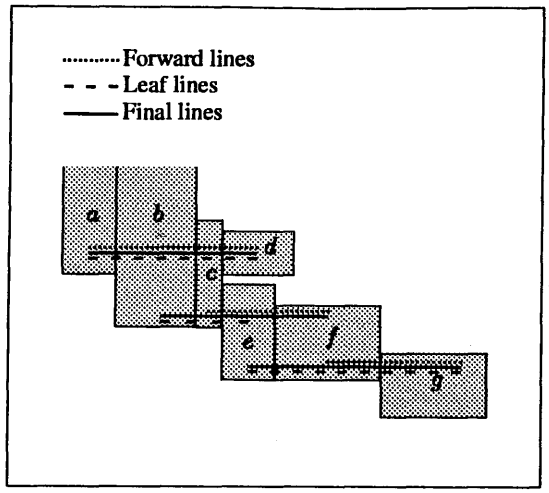


Figure 10: An example of placing orthogonal axial lines in a tree of orthogonal rectangles

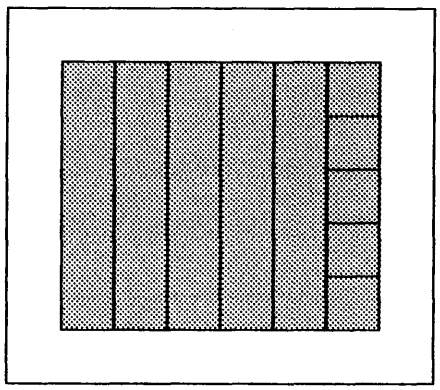


Figure 11: A tree with $n/2$ leaves and height also $n/2$

Stage 5: Merge forward lines and leaf lines For a forward line to exist, it must cross at least one adjacency between two rectangles in the tree, and each adjacency is crossed exactly once by the definition of the axial line placement problem. Therefore since there are n rectangles, there are exactly $n-1$ of these adjacencies and there can be a maximum of $O(n)$ forward lines.

The merging phase of the algorithm considers each forward line in turn and in the worst case works from the rightmost rectangle of the line until a leaf line with overlap is encountered. The algorithm then merges these two lines. Effectively this is merging two lines in a chain of rectangles which is $O(n)$ (as shown in Section 4).

So since there are a maximum of $O(n)$ forward lines and merging a forward line with its associated leaf line is $O(n)$, this part of the algorithm is $O(n^2)$ in the worst case.

Overall The complexity of the entire algorithm is thus $O(n^2)$ in the worst case but better performance can be expected from some configurations of input rectangles.

5.3 Empirical Analysis

The algorithm consists of four parts: finding the adjacencies, finding the forward lines, finding the leaf lines, and merging the forward lines and leaf lines. The algorithm was implemented and tested on various configurations of trees. The results of this empirical analysis confirm the theoretical analysis and also demonstrate that different shapes of the trees of rectangles affect the complexity quite dramatically. In some cases the sorting and finding the forward lines dominate and in other cases finding the leaf lines and merging the forward and leaf lines are the more costly operations. Sanders *et al.* [7] present detailed results of the empirical analysis.

6 Future Work

Sanders *et al.* [6] presented some variations of the orthogonal axial line placement problem which can be solved in polynomial time. In this article polynomial time algorithms for the orthogonal axial line placement problem for chains of orthogonal rectangles and trees of orthogonal rectangles have been presented. Future research can be focussed on finding other arrangements of rectangles which

are more general than a tree of rectangles which can be solved in polynomial time.

An interesting, but not directly related, area of further research is in the generation of test data. Generating non-trivial trees of adjacent but non-overlapping rectangles proved to be relatively complex in this research.

7 Conclusion

The axial line placement problem in general has applications in various fields such as VLSI design, computer graphics, databases, image processing and town planning. However, the orthogonal axial line placement problem in orthogonal rectangles is NP-complete in general [6]. This article presents some restrictions of the orthogonal axial line placement problem for which polynomial time solutions can be obtained – chains of orthogonal rectangles ($O(n)$) and trees of orthogonal rectangles ($O(n^2)$). It is likely that other restrictions exist which are solvable in polynomial time and future research will investigate this area.

References

[1] T.H. Cormen, C.E. Leierson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1996.

[2] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[3] B. Hillier, J. Hanson, J. Peponis, J. Hudson, and R. Burdett. Space syntax, a different urban perspective. *Architect's Journal*, 179:47–63, 1983.

[4] J. O'Rourke. Personal communication, 1999.

[5] I. D. Sanders, D. Lubinsky, and M. Sears. Ray guarding configurations of adjacent rectangles. In *Proceedings of The 1997 National Research and Development Conference (SAICSIT 97)*, pages 221–238. Potchefstroom University for Higher Christian Education, October 1997.

[6] I. D. Sanders, D. Lubinsky, M. Sears, and D. Kourie. Orthogonal Ray Guarding of Adjacencies between Orthogonal Rectangles. *South African Computer Journal*, (23):18–29, July 1999.

[7] I. D. Sanders, D. C. Watts, and A. D. Hall. Orthogonal Axial Line Placement in Chains and Trees of Rectangles. Technical Report TR-Wits-CS-2000-8, Department of Computer Science, University of the Witwatersrand, June 2000.

[8] D. C. Watts and I. D. Sanders. Ray Guarding Chains of Rectangles. Technical Report TR-Wits-CS-1997-02, Department of Computer Science, University of the Witwatersrand, 1997.

South African Computer Journal

Number 25, August 2000
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 25, Augustus 2000
ISSN 1015-7999

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for *review* should be prepared according to the following guidelines:

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
- the title (as brief as possible)
- the author's initials and surname
- the author's affiliation and address
- an abstract of less than 200 words
- an appropriate keyword list
- a list of relevant Computing Review Categories
- Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text according to the Harvard. References should also be according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page,

`http://www.cs.up.ac.za/sacj`

which gives a number of examples.

SACJ is produced using the LaTeX document preparation system, in particular LaTeX 2_ε. Previous versions were produced using a style file for a much older version of LaTeX, which is no longer supported. Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 25, August 2000
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 25, August 2000
ISSN 1015-7999

Contents

Editorial
Stef Postma Memorial 1
Links2Go "Computer Science Journals" Award 2

Research Articles

A New Approach for Program Integration
Z-E Bouras, T Khammaci, S Ghoul 3

Technological Experience and Technophobia in South African University Students
MC Clarke 12

Image coding with Fractal Vector Quantization
E Cloete, LM Venter 18

A Declarative Framework for Temporal Discrete Simulation
H Abdulrab, M Ngomo, A Drissi-Talbi 23

Multilingual Training of Acoustic Models in Automatic Speech Recognition
C Nieuwoudt, EC Botha 32

Syntactic Description of Neighbourhood in Quadtree
JR Tapamo 38

Object Oriented Programs and a Stack Based Virtual Machine
JT Waldron 45

Technical Reports

Orthogonal Axial Line Placement in Chains and Trees of Orthogonal Rectangles
ID Sanders, DC Watts, AD Hall 56

Scalability of the RAMpage Memory Hierarchy
P Machanick 68