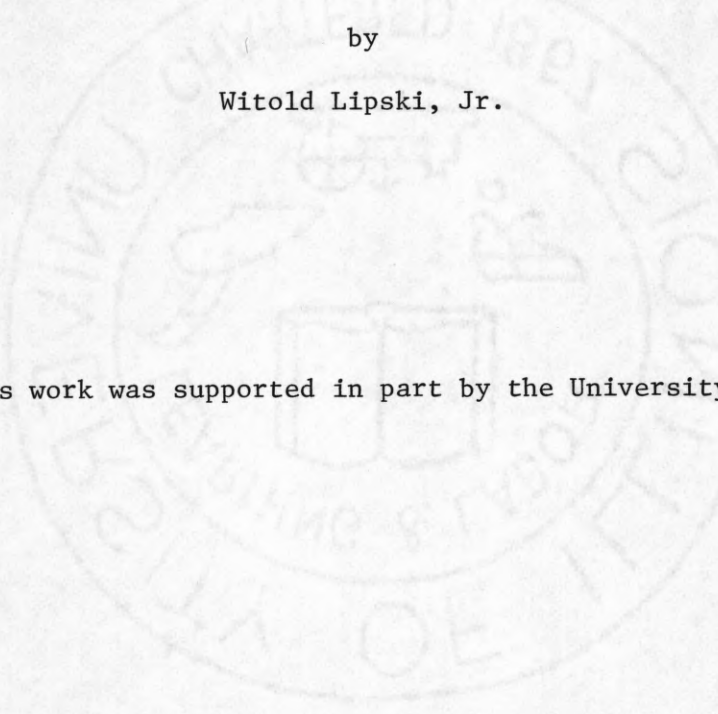


FINDING A MANHATTAN PATH AND RELATED PROBLEMS

by

Witold Lipski, Jr.

This work was supported in part by the University of  
Illinois.

A faint, circular seal of the University of Illinois is visible in the background. It features a central shield with a book and a torch, surrounded by the text "UNIVERSITY OF ILLINOIS" and "FOUNDED 1808".

# FINDING A MANHATTAN PATH AND RELATED PROBLEMS

Witold Lipski, Jr.\*

Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

Urbana, IL 61801, USA

Abstract. Let  $S$  be a set of  $n$  horizontal and vertical segments on the plane, and let  $s, t \in S$ . A Manhattan path (of length  $k$ ) from  $s$  to  $t$  is an alternating sequence of horizontal and vertical segments  $s = r_0, r_1, \dots, r_k = t$  where  $r_i$  intersects  $r_{i+1}$ ,  $0 \leq i < k$ . We give an  $O(n \log^2 n)$  algorithm to find, for a given  $t$ , a tree of shortest Manhattan paths from all  $s \in S$  to  $t$ . We also determine a maximum set of crossings (intersections of segments) with no two on the same segment, as well as a maximum set of nonintersecting segments, both in  $O(n^{3/2} \log^2 n)$  time. The latter algorithm is applied to decomposing, in  $O(n^{3/2} \log^2 n)$  time, a hole-free union of  $n$  rectangles with sides parallel to the coordinate axes into the minimal number of disjoint rectangles. All the algorithms require  $O(n \log n)$  space, and for all of them the factor  $\log^2 n$  can be improved to  $\log n \log \log n$ , at the cost of some complication of the basic data structure used.

Keywords and phrases: computational geometry, horizontal and vertical segments, segment tree, Manhattan path, minimal decomposition into disjoint rectangles.

---

\* On leave from the Institute of Computer Science, Polish Academy of Sciences, P. O. Box 22, 00-901 Warsaw PKiN, Poland.

This work was supported in part by the University of Illinois.

The diagram shows a stepped profile composed of several horizontal and vertical segments. The leftmost vertical segment is labeled 's'. The rightmost vertical segment is labeled 't'. The profile is defined by a series of horizontal lines of varying lengths and vertical lines connecting them. A faint circular watermark is visible in the background.

In Section 3 we give an  $O(n \log^2 n)$  time algorithm to find, for a given  $t$ , a tree of shortest Manhattan paths from all (reachable from  $t$ ) segments  $s$  to  $t$  (i.e., a tree with root  $t$  where the path from any node  $s$  to the root is the shortest Manhattan path between these segments).



In Section 4, we present algorithms to determine a maximum set of crossings (intersections of segments) with no two on the same segment, as well as a maximum set of nonintersecting segments. Both algorithms run in time  $O(n^{3/2} \log^2 n)$ .

Finally, in Section 5, the efficient determination of a maximum set of nonintersecting segments is applied to produce an  $O(n^{3/2} \log^2 n)$  algorithm to solve the following problem (see [10]): Given  $n$  rectangles,  $R_1, \dots, R_n$  with sides parallel to the coordinate axes, with the union  $F = R_1 \cup \dots \cup R_n$  without holes, find a decomposition of  $F$  into the minimal possible number of disjoint rectangles.

All the algorithms use the same basic data structure for storing and manipulating the collection  $S$  of segments. This data structure is essentially Bentley's segment tree (see [2,3]). We show how this basic data structure can be refined, by using techniques developed by van Emde Boas [4,5], so that for all our algorithms the factor  $\log^2 n$  is improved to  $\log \log \log n$ . However, this modification considerably complicates the data structure. Under both implementations all our algorithms require  $O(n \log n)$  space.

Notice that a Manhattan path corresponds to the usual path in the intersection graph of  $S$ , i.e., the graph with vertices corresponding to segments  $s \in S$ , two vertices joined by an edge iff the corresponding segments intersect each other. However, the naive approach based on transforming the problem of finding a Manhattan path into the corresponding problem for the intersection graph requires in general  $\Omega(n^2)$  time for merely constructing the intersection graph, which may have up to  $n^2/4$  edges.

Before we describe the basic data structure in more detail in Section 2, let us introduce some notation. We denote the collection of streets (horizontal segments) and avenues (vertical segments) by  $H$  and  $V$ , respectively, so that  $S = H \cup V$ ,  $|H| + |V| = n$ . We may normalize the coordinates of the segments by replacing every abscissa and ordinate of an endpoint of a segment by its rank in the set of all different abscissae and ordinates, respectively. Denote by  $M$  the number of distinct abscissae and by  $N$  the number of distinct ordinates (clearly,  $M, N \leq 2n$ ). Every segment  $s$  may be represented by three integers  $Y[s]$ ,  $L[s]$ ,  $R[s]$  interpreted as its ordinate, left abscissa and right abscissa, respectively, if  $s$  is horizontal; for a vertical segment they are interpreted as its abscissa, bottom ordinate and top ordinate, respectively. We identify the segments by integers  $1, \dots, n$  in such a way that segments  $1, \dots, |H|$  are horizontal, segments  $|H|+1, \dots, n$  are vertical, and, moreover,  $Y[1] \leq \dots \leq Y[|H|]$  and  $Y[|H|+1] \leq \dots \leq Y[n]$ . It is clear that the above normalization process can be carried out in  $O(n \log n)$  time. Notice that from now on all sorting operations are doable in  $O(n)$  time by standard bucket sorting (see e.g. [1]).

## 2. SEGMENT TREES

Let  $a, b$  be two integers with  $a \leq b$ . The segment tree  $T(a,b)$  is defined in the following recursive way. It consists of the root  $v$  with  $B[v] = a$ ,  $E[v] = b$ , and, if  $a < b$ , of a left subtree  $T(a, \lfloor (a+b)/2 \rfloor)$  and right subtree  $T(\lfloor (a+b)/2 \rfloor + 1, b)$ . The roots of these subtrees are given by  $LSON[v]$  and  $RSON[v]$ , respectively. If  $b = a$  then  $LSON[v] = RSON[v] = \Lambda$ . Every node  $u$  of  $T(a,b)$  corresponds to the interval  $[B[u], E[u]]$  of (integer) values of abscissa; in particular,  $B[u] = E[u]$  if  $u$  is a leaf, so that the leaves are in a one-to-one correspondence with the integers  $a, a+1, \dots, b$ .

Define  $TH = T(1, N)$ . We say that a street covers node  $v$  if  $[L[s], R[s]] \supseteq [B[v], E[v]]$  and this inclusion is not true for the father of  $v$ . Every street  $s \in H$  will be stored at each of the nodes it covers - it is easily seen that there are at most  $2^{\lceil \log N \rceil - 4}$  such nodes (for  $N > 4$ ). The collection of streets covering a node  $v$  will be maintained as a usual balanced binary search tree corresponding to the ordering of the streets by the increasing ordinate. Street  $s$  can be inserted into and deleted from this tree by  $\text{insert}(s, v)$  and  $\text{delete}(s, v)$ , respectively in time logarithmic in the number of streets currently stored at node  $v$ .

Inserting a street  $s$  into  $TH$  is accomplished by calling  $\text{INSERT}(s, \text{root}(TH))$ , where  $\text{INSERT}$  is the following recursive procedure:

```

procedure INSERT( $s, v$ )
  (*insert segment  $s$  at node  $v$  of segment tree*)
  begin
    if ( $L[s] \leq B[v]$ ) and ( $E[v] \leq R[s]$ ) then  $\text{insert}(s, v)$ 
    else begin if  $L[s] \leq (B[v] + E[v]) / 2$  then  $\text{INSERT}(s, LSON[v])$ 
      if  $(B[v] + E[v]) / 2 < R[s]$  then  $\text{INSERT}(s, RSON[v])$ 
    end
  end

```



Similarly,  $s$  can be deleted from  $TH$  by  $DELETE(s, root(TH))$ , where  $DELETE$  follows the same pattern as  $INSERT$ :

```

procedure  $DELETE(s, v)$ 
  (*delete segment  $s$  at node  $v$  of segment tree*)
  begin
    if  $(L[s] \leq B[v])$  and  $(E[v] \leq R[s])$  then  $delete(s, v)$ 
    else begin if  $L[s] \leq (B[v] + E[v]) / 2$  then  $DELETE(s, LSON[v])$ 
              if  $(B[v] + E[v]) / 2 < R[s]$  then  $DELETE(s, RSON[v])$ 
    end
  end

```

It is easy to see that both inserting and deleting a street involve visiting  $O(\log N) = O(\log n)$  nodes of  $TH$  with  $O(\log n)$  work spent at each node, which amounts to  $O(\log^2 n)$  total work (we assume the number of streets currently represented in  $TH$  does not exceed  $n$ ).

In exactly the same way we store, insert, and delete avenues in a segment tree  $TV = T(1, M)$ . For any segment (street or avenue)  $t$  we can now find all segments in  $S$  intersecting  $t$ , by calling the following procedure:

```

procedure  $LIST(t, QUEUE)$ 
  (*put on  $QUEUE$  all segments intersecting  $t$ *)
  begin
    if  $t \leq |H|$  then  $v := root(TV)$  (* $t$  is horizontal*)
    else  $v := root(TH)$  (* $t$  is vertical*)
    while  $v \neq \Lambda$  do
      begin  $list(t, v, QUEUE)$ 
        if  $Y[t] \leq (B[v] + E[v]) / 2$  then  $v := LSON[v]$ 
        else  $v := RSON[v]$ 
      end
    end
  end

```

Here  $list(t, v, QUEUE)$  is a procedure which puts on  $QUEUE$  all segments  $p$  with  $L[t] \leq Y[p] \leq R[t]$  stored at node  $v$ . To prove the correctness of  $LIST$  assume, without loss of generality, that  $t$  is vertical. Notice that a street  $s$  intersects  $t$  if and only if (a)  $s$  has its ordinate between the

bottom and top ordinates of  $t$ , i.e.  $L[t] \leq Y[p] \leq R[t]$ , and (b)  $s$  covers some node  $v$  with  $[B[v], E[v]]$  containing  $Y[t]$ , the abscissa of  $t$ . Our procedure follows a path from the root to the leaf corresponding to  $Y[t]$ , i.e. visits all nodes  $v$  with  $[B[v], E[v]]$  containing  $Y[t]$ . At each such node  $v$ ,  $\text{list}(t, v, \text{QUEUE})$  selects from all segments covering  $v$  only those satisfying condition (a).

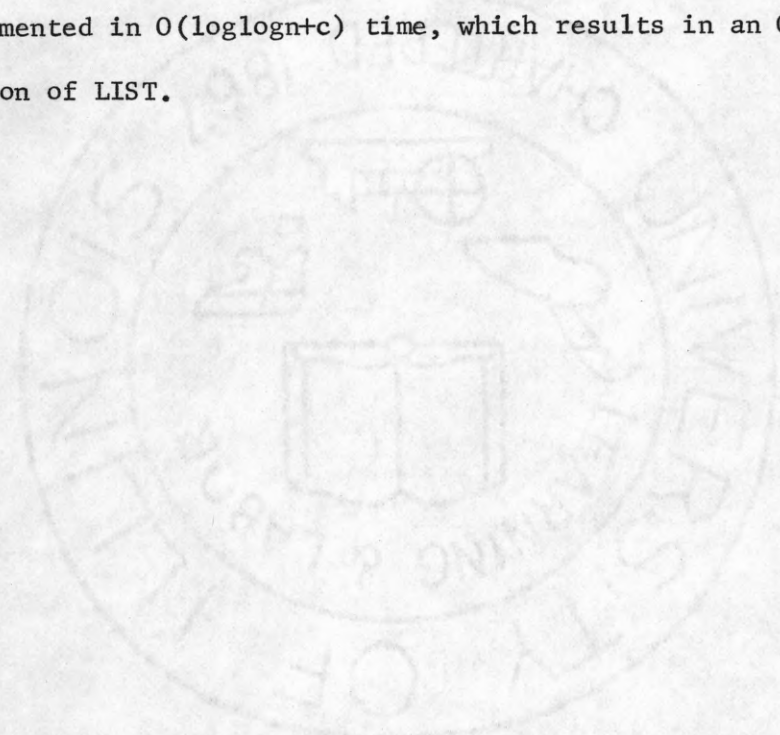
We can implement  $\text{list}(t, v, \text{QUEUE})$  in  $O(\log n + c)$  time, where  $c$  is the number of segments it puts on  $\text{QUEUE}$ . It follows that the total complexity of  $\text{LIST}$  is  $O(\log^2 n + k)$ , where  $k$  is the total number of segments put on  $Q$ .

The simplest way to construct the segment trees representing collections  $H$  and  $V$  is the following. We first form segment trees  $TH$  and  $TV$  with an empty search tree at each node. This can be done in  $O(n)$  time by a recursive algorithm exactly following the definition of a segment tree. Then we insert all streets into  $TH$  and all avenues into  $TV$ , one by one, by using  $\text{INSERT}$ . Such a method clearly takes  $O(n \log^2 n)$  time. This can be improved by inserting both all streets and all avenues in the increasing order of  $Y[t]$ , and first assembling the segments covering a node of a segment tree as a linked list, ordered by  $Y[t]$ . Then every insertion takes only  $O(\log n)$  time; moreover, at the end of the process each of the lists can be converted into a balanced binary search tree in time linear in its length, so that the total work involved is  $O(n \log n)$ .

Notice also that since every segment is identified by an integer, and since both for streets and for avenues this labelling corresponds to an ordering by nondecreasing value of  $Y$ , we may assume that any node of the



segment trees stores a collection of integers in the range  $[1, n]$ . This enables us to use instead of the usual search tree, a special tree structure developed by van Emde Boas [4,5] and to implement both insert and delete in  $O(\log \log n)$  time, and consequently both INSERT and DELETE in  $O(\log \log \log n)$  time. It is also clear that list can then be implemented in  $O(\log \log n + c)$  time, which results in an  $O(\log \log \log n + k)$  version of LIST.



### 3. FINDING A MANHATTAN PATH

Given a segment  $t \in s$  we shall find a tree of shortest (in the sense of the number of segments) Manhattan paths to  $t$  from all segments  $s$  from which  $t$  is reachable. More exactly, for any such segment  $s$  we find a segment  $\text{NEXT}[s]$  such that the sequence  $s = s_0, s_1, s_2, \dots$ , where  $s_{i+1} = \text{NEXT}[s_i]$ , determines the shortest Manhattan path from  $s$  to  $t$ .

We perform what is essentially a breadth first search in the intersection graph of the segments, so that the tree obtained corresponds exactly to the breadth first search spanning tree of (a connected component of) this graph (see e.g. [8]). We start with segment  $t$ , and any segment  $s$  reached from  $t$  is first deleted from the segment tree containing it and put on a QUEUE; when its turn comes, it is deleted from QUEUE and explored, i.e. all so far unreached segments it intersects are put on QUEUE. The algorithm is summarized below.

Algorithm 1 (Finding a breadth first search spanning tree of shortest Manhattan paths to  $t$ )

```

1 begin
2   for  $s := 1$  to  $n$  do  $\text{NEXT}[s] := \Lambda$       (*initialize*)
3    $\text{QUEUE} := \emptyset$ ,  $Q := \emptyset$ 
4    $\text{QUEUE} \leftarrow t$  (*the breadth first search starts at  $t$ *)
5   while  $\text{QUEUE} \neq \emptyset$  do
6     begin  $s \leftarrow \text{QUEUE}$       (*consider next unexplored segment*)
7       if  $s \leq |H|$  then  $v := \text{root}(TV)$  (* $s$  is horizontal*)
8       else  $v := \text{root}(TH)$  (* $s$  is vertical*)
9        $\text{LIST}(s, Q)$  (*put segments intersecting  $s$  on  $Q$ *)
10      while  $Q \neq \emptyset$  do
11        begin  $p \leftarrow Q$ 
12           $\text{DELETE}(p, v)$ 
13           $\text{QUEUE} \leftarrow p$ 
14           $\text{NEXT}[p] := s$ 
15        end
16      end
17 end

```

An example of a tree of shortest Manhattan paths constructed by Algorithm 1 is given in Fig. 2.

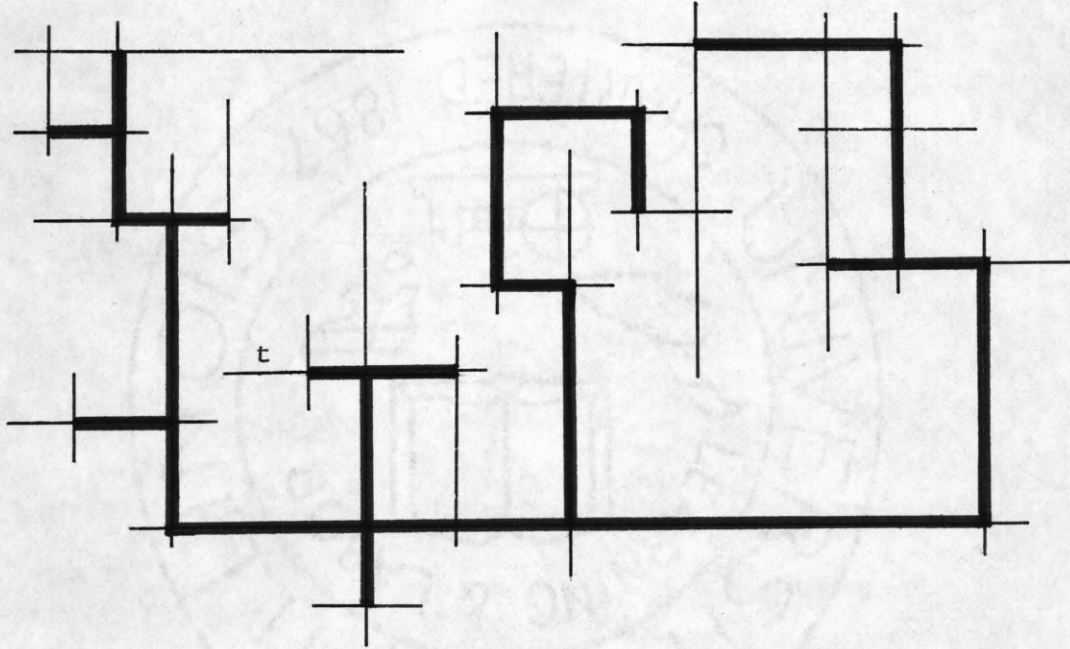


Figure 2. A breadth first search tree of shortest Manhattan paths to  $t$ .

In order to evaluate the time complexity of Algorithm 1 notice that each segment is listed by LIST and deleted by DELETE at most once. Let us denote by  $k_i$  the number of segments put on  $Q$  by the  $i$ -th call of LIST. The total complexity of the algorithm is

$$O(n \log^2 n + \sum_i (\log^2 n + k_i)) = O(n \log^2 n).$$

The implementation using the van Emde Boas trees improves this to

$$O(n \log n \log \log n + \sum_i (\log n \log \log n + k_i)) = O(n \log n \log \log n).$$



Notice that if we replace  $\text{NEXT}[p] := s$  in line 14 of Algorithm 1 by  $\text{ROOT}[p] := t$ , and if we perform the breadth first search over the whole collection of segments (by choosing a new, not yet considered segment  $t$  as a starting point each time when  $\text{QUEUE} = \emptyset$ ), then we obtain a structure capable of answering in constant time questions of the form "is there connection between  $s$  and  $t$ ?" (there is a connection iff  $\text{ROOT}[s] = \text{ROOT}[t]$ ). Of course the complexity of such a modified algorithm remains  $O(n \log^2 n)$  or  $O(n \log n \log \log n)$ , depending on the implementation.

#### 4. FINDING MAXIMUM INDEPENDENT SET OF CROSSINGS AND MAXIMUM SET OF NON-INTERSECTING SEGMENTS

A crossing is the intersection point of a street and an avenue. A set  $C$  of crossings is called independent if no two crossings in  $C$  lie on the same segment  $s \in S$ . The word "maximum" means, as usual, "with the maximal possible number of elements."

Let  $G$  be the intersection graph of our collection  $S$  of segments. Clearly,  $G$  is bipartite, every edge of  $G$  corresponds to a crossing, any maximum independent set of crossings (MISC) corresponds to a maximum matching in  $G$ , while any maximum set of non-intersecting segments (MSNIS) corresponds to a maximum independent set of vertices in  $G$ . This is shown in Fig. 3.

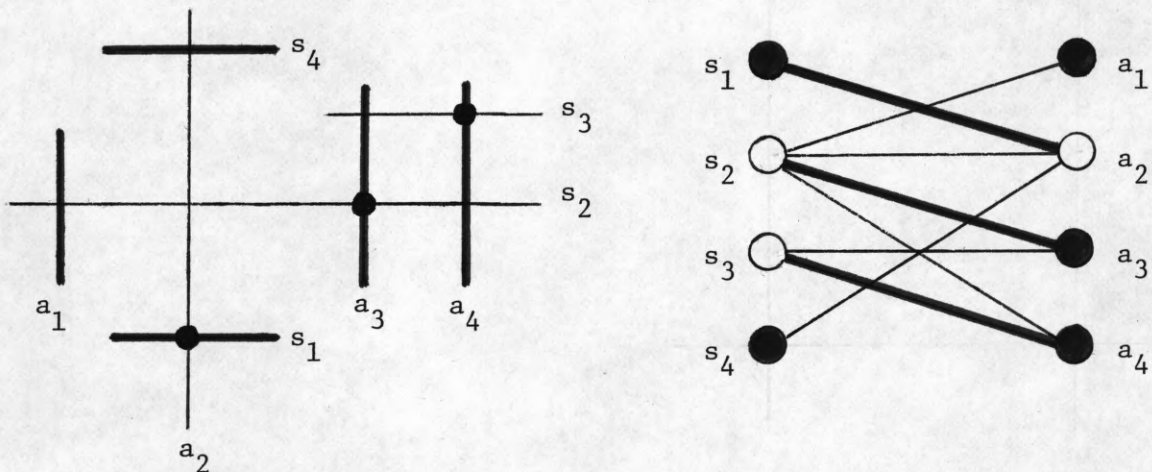


Figure 3. A maximum independent set of crossings, a maximum set of non-intersecting segments; and the corresponding maximum matching and maximum independent set in the intersection graph.

Our method of finding an MISC follows closely the Hopcroft and Karp maximum matching algorithm [7], as applied to  $G$ ; however, we shall not explicitly construct  $G$ , as it would require  $\Omega(n^2)$  time in worst case. We assume the reader is familiar with [7], and with the basic notions related to matchings, such as that of an augmenting path.

The current matching constructed by the algorithm is represented by an array  $\text{MATCH}[1:|H|]$ , where  $\text{MATCH}[s]$  is the avenue currently matched to street  $s$ ;  $\text{MATCH}[s] = \Lambda$  if  $s$  is free (unmatched). All streets are represented in a segment tree  $TH$ . Assume that every edge of  $G$  is directed from a street to an avenue if this edge is in the current matching, and from an avenue to a street otherwise. Then any augmenting path corresponds to a Manhattan path  $a_1, s_1, a_2, s_2, \dots, a_k, s_k$ , where  $k \geq 1$ ,  $a_1$  and  $s_k$  are free, and  $\text{MATCH}[s_i] = a_{i+1}$ ,  $1 \leq i < k$ . Below we describe the steps of our algorithm, closely corresponding to steps of the original Hopcroft and Karp algorithm.

Algorithm 2. (Finding a maximum set of independent crossings.)

Step 1. Initialize the empty matching by putting  $\text{MATCH}[s] = \Lambda$ ,  $1 \leq s \leq |H|$ .

Step 2. Starting with the set of currently free avenues breadth first search our directed graph  $G$ . Let  $\ell$  be the length of a shortest path from a free avenue to a free street (the algorithm halts if no such path exists). Put into segment tree  $T$  all streets reachable from free avenues by paths of length not exceeding  $\ell$ . This step can easily be implemented by a modification of Algorithm 1. Notice that it is not necessary to store the avenues in a segment tree, since the only avenue intersecting a given street  $s$  we consider is  $\text{MATCH}[s]$ .



Step 3. Find a maximal set of disjoint augmenting paths using the streets stored in  $T$ , and augment the matching along these paths. (Note 1: A set is "maximal" with a given property if it is not properly contained in any other set with this property; Note 2: The paths are disjoint in the sense that no two of them use the same vertex of the intersection graph - but segments corresponding to vertices of two different paths may of course intersect). This is done as in [7], by a depth first search of  $G$  starting in the set of free avenues and using only streets in  $T$ . Each time we reach a free street, we obviously found an augmenting path. In such a case we augment the matching along this path, and we continue the search starting at a new free avenue. The depth first search is implemented in the usual way, by using a STACK (instead of the QUEUE in the breadth first search). Instead of LIST, we use a procedure which finds, for any avenue  $a$ , some street  $CROSS(a)$  in  $T$  intersecting  $a$ ;  $CROSS(a) = \Lambda$  if no such street exists. The key to the efficiency of our search is, similarly as in the case of Algorithm 1, the fact that every street reached by the search is immediately deleted from  $T$  (see line 13 of the procedure AUGMENT below), so that  $T$  always contains only so far unreached streets. The search is summarized below by a procedure AUGMENT. FREEAVENUES is a queue containing all free avenues, and STACK always contains an alternating sequence of avenues and streets corresponding to an initial part of an augmenting path which the search tries to construct.

```

1  procedure AUGMENT
   (*augment current matching along maximal collection of disjoint
   augmenting paths*)
2  begin
3    while FREEAVENUES  $\neq \emptyset$  do
4      begin
5        a  $\leftarrow$  FREEAVENUES (*a is a current free avenue*)
6        STACK  $\leftarrow$  a (*STACK assembles the segments of an augmenting path*)
        (*depth first search starting at a*)
7        while STACK  $\neq \emptyset$  do
8          begin
9            a := top (STACK)
10           s := CROSS(a)
11           if s  $\neq \Lambda$  then (*s is a street intersecting a*)
12             begin STACK  $\leftarrow$  s
13               DELETE(s, root(T))
14               if MATCH[s] =  $\Lambda$  then (*s is a free street*)
15                 while STACK  $\neq \emptyset$  do (*augment the matching*)
16                   begin sm  $\leftarrow$  STACK
17                     am  $\leftarrow$  STACK
18                     MATCH[sm] := am
19                   end
20                 else (*MATCH[s]  $\neq \Lambda$ , i.e. s is not free*)
21                   begin a := MATCH[s]
22                     STACK  $\leftarrow$  a
23                   end
24                 end
25               else (*s =  $\Lambda$ , i.e. sequence on STACK cannot be extended*)
26                 begin a  $\leftarrow$  STACK
27                   if STACK  $\neq \emptyset$  then s  $\leftarrow$  STACK
28                 end
29               end
30             end
31           end

```

After the completion of Step 3 we return to Step 2, and the loop consisting of Step 2 and Step 3 is iterated until we arrive at the situation where no augmenting path is found in Step 2; the execution of the algorithm is then completed.

It follows from the analysis given in [7] that the main loop of the algorithm, i.e. Steps 2 and 3 are executed  $O(\sqrt{n})$  times. The complexity of Step 2, including the construction of the segment tree T, is clearly

$O(n \log^2 n)$ . Since procedure CROSS can easily be implemented in  $O(\log^2 n)$  time, and since in procedure AUGMENT every segment is reached by CROSS, pushed onto, and popped from the STACK at most once, it follows that Step 3 also runs in  $O(n \log^2 n)$  time. Consequently, the total complexity of our algorithm is  $O(n^{3/2} \log^2 n)$ . As before, this can be improved to  $O(n^{3/2} \log n \log \log n)$ .

There is a standard method for obtaining a maximum independent set of vertices from a maximum matching in a bipartite graph (see e.g. [6,10]). As applied to the construction of an MSNIS from an MISC it can be described as follows. We find the sets  $H_0$  and  $S_0$  of streets and avenues, respectively, reachable by alternating paths originating in the set of free avenues. An MSNIS is then obtained as  $S^* = S_0 \cup (H \setminus H_0)$ . The sets  $S_0$  and  $H_0$  can easily be obtained in  $O(n \log^2 n)$  time by the usual breadth first search (in fact, they are actually found in the last execution of Step 2 in Algorithm 2). We conclude that an MSNIS can be obtained in  $O(n^{3/2} \log^2 n)$  (or  $O(n^{3/2} \log n \log \log n)$ ) time.



## 5. AN APPLICATION TO DECOMPOSING INTO THE MINIMUM NUMBER OF RECTANGLES

Finding a maximum set of non-intersecting segments is a crucial step in solving the following problem (see [10]): Given a collection of  $n$  rectangles  $R_1, \dots, R_n$  with sides parallel to the coordinate axes, find the decomposition of the union  $F = R_1 \cup \dots \cup R_n$  into the minimal possible number of disjoint rectangles. We shall consider here only the case where  $F$  does not contain holes (our algorithm works correctly in the general case, without however being superior over the  $O(n^3)$  method given in [10]). In such a case the boundary of  $F$  consists of at most  $8n-4$  edges, and it can be constructed in  $O(n \log n)$  time (see [9]). The main idea of the decomposition algorithm, as described in [10], is the following. We cut  $F$  along a (vertical or horizontal) segment  $s$  joining a concave vertex of the contour of  $F$  with some other point on the contour, and contained entirely within  $F$ . Then the decomposition procedure is called recursively for the two resulting figures. Since this cutting process is repeated until there are no concave vertices left, and since the number of rectangles in the decomposition is equal to the number of cuts plus one (for a connected  $F$ ) it is clear that we should try to make each of the cuts along a segment joining two concave vertices, which decreases the total number of concave vertices by two, instead of just one. More precisely, it is shown in [10] that the optimal decomposition is obtained by first cutting  $F$  along a maximum set of non-intersecting segments joining pairs of concave vertices.

Clearly, the collection  $S$  of segments involved in the above problem is the collection of all vertical and horizontal segments joining pairs of concave vertices of  $F$  and contained entirely in  $F$ . We now show how to find the subcollection  $V$  of vertical segments (finding  $H$  is analogous). We store the collection of horizontal edges of the contour in a segment tree  $TH$ , and we scan the sequence of vertical edges of the contour sorted primarily by the value of abscissa and secondarily by the bottom ordinate. For any two consecutive edges with the same ordinate, the segment joining the upper endpoint of the first one to the lower endpoint of the second one is included to  $V$  provided (i) both its endpoints are concave vertices of  $F$ , and (ii) no horizontal edge intersects it. Since condition (ii) can easily be tested in  $O(\log^2 n)$  time, it is clear that  $S$  can be found in  $O(n \log^2 n)$  time. Then we find an MSNIS  $S^*$ , as described in the previous section, and we cut  $F$  first along the segments in  $S^*$ , and then along some other allowable lines. Leaving the details of an efficient implementation of the cutting process to the reader, we conclude that the whole decomposition algorithm runs in  $O(n^{3/2} \log^2 n)$  time (again, an improvement to  $O(n^{3/2} \log \log \log n)$  is possible).

### References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass. 1974.
2. J. L. Bentley, Solutions to Klee's rectangle problems. Unpublished notes, Carnegie-Mellon University, 1977.
3. J. L. Bentley and D. Wood, An optimal worst-case algorithm for reporting intersections of rectangles. Carnegie-Mellon University, 1979.
4. P. van Emde Boas, Preserving order in a forest in less than logarithmic time. Proc. 16th Annual Symp. on Foundations of Comp. Sci., Univ. of California, Berkeley, Oct. 1975, pp. 75-84.
5. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space. Information Proc. Lett. 6 (1977), pp. 80-82.
6. E. Gavril, Testing for equality between maximum matching and minimum node covering. Information Proc. Lett. 6 (1977), pp. 199-202.
7. J. E. Hopcroft and R. M. Karp, An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. 2 (1973), pp. 225-231.
8. E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms. Computer Science Press, Inc., Potomac, MD, 1978.
9. W. Lipski and F. P. Preparata, Finding the contour of a union of iso-oriented rectangles. Coordinated Science Lab., Univ. of Illinois at Urbana-Champaign, July 1979.
10. W. Lipski, E. Lodi, F. Luccio, C. Mugnai and L. Pagli, On two dimensional data organization II. Tech. Rep. S-77-43, Inst. of Computer Sci., Univ. of Pisa, Italy, December 1977. To appear in Fundamenta Informaticae.



In Section 4, we present algorithms to determine a maximum set of crossings (intersections of segments) with no two on the same segment, as well as a maximum set of nonintersecting segments. Both algorithms run in time  $O(n^{3/2} \log^2 n)$ .

Finally, in Section 5, the efficient determination of a maximum set of nonintersecting segments is applied to produce an  $O(n^{3/2} \log^2 n)$  algorithm to solve the following problem (see [10]): Given  $n$  rectangles,  $R_1, \dots, R_n$  with sides parallel to the coordinate axes, with the union  $F = R_1 \cup \dots \cup R_n$  without holes, find a decomposition of  $F$  into the minimal possible number of disjoint rectangles.

All the algorithms use the same basic data structure for storing and manipulating the collection  $S$  of segments. This data structure is essentially Bentley's segment tree (see [2,3]). We show how this basic data structure can be refined, by using techniques developed by van Emde Boas [4,5], so that for all our algorithms the factor  $\log^2 n$  is improved to  $\log \log \log n$ . However, this modification considerably complicates the data structure. Under both implementations all our algorithms require  $O(n \log n)$  space.