EFFICIENT ALGORITHMS FOR GEOMETRIC
GRAPH SEARCH PROBLEMS*

HIROSHI IMAI† AND TAKAO ASANO††

Abstract. In this paper, we show that many graph search problems can be solved quite efficiently for a geometric intersection graph of horizontal and vertical line segments. We first extract several basic operations for depth first search and breadth first search on a graph. Then we present data structures for the intersection graph in terms of which those operations can be implemented in an efficient manner. The data structures enable us to solve various graph search problems besides depth first search and breadth first search. Specifying the results obtained in this paper for an intersection graph of \( n \) horizontal and vertical segments with \( m \) pairs of intersecting segments, we obtain algorithms with the following complexity, where \( N = \min \{ m, n \log n \} \).

(i) Depth first search and breadth first search can be executed in \( O(n \log n) \) time and \( O(N) \) space.
(ii) The biconnected components can be found in \( O(n \log n) \) time and \( O(N) \) space.
(iii) A maximum matching and a maximum independent set can be found in \( O(\sqrt{n \cdot N}) \) time and \( O(N) \) space when no two horizontal (vertical) segments intersect.
(iv) The connectivity \( k_G \) can be found in \( O(k_G n^{3/2} \cdot N) \) time and \( O(N) \) space.

Our algorithms can be applied to various practical problems such as the problem of finding a minimum dissection of a rectilinear region, which arises in the manipulation of VLSI artwork data, and the problem of determining whether there is a Manhattan wiring on a single layer, which arises in the design automation of digital systems.

Key words. computational geometry, segment tree, orthogonal segment intersection search, intersection graph, graph algorithms, depth-first search, linear-time set union algorithm

1. Introduction. In most graph algorithms, graphs are represented by adjacency lists (e.g., see [1]), and the complexity of each algorithm is estimated on the basis of them. However, for graphs with some prescribed properties, it may be possible to design efficient graph algorithms by employing other data structures which make use of those properties. In this paper, we show that this is the case for some geometric intersection graphs, where an intersection graph of objects in the plane is obtained by identifying each object with a vertex, and connecting two vertices by an edge if their corresponding objects intersect. Specifically, we consider problems for an intersection graph of horizontal and vertical line segments. In Fig. 1.1, an example is shown.

We first extract several basic operations for a graph which are required in executing depth first search and breadth first search. The operations are mainly to find, for a vertex, an arc emanating from it (this operation is called LIST1), and to delete the incoming arcs incident to the vertex (this is called DELETE). These are a bit different from ordinary implementations in deleting arcs coming into a vertex from the graph in place of labeling the vertex. That is, in ordinary implementations for depth first search and breadth first search, a vertex which has been searched is labeled (or, marked) so that the vertex will never be searched again. However, in our implementations, we delete arcs coming into the vertex from the graph for this purpose.

We then present efficient data structures for the intersection graph so that the basic operations can be executed quickly. Our data structures are based on those for intersection problems of horizontal and vertical line segments in computational geometry, where the segment tree as introduced by Bentley [2] plays an important role. This paper was started in 1983 and was completed in 1984.

* Received by the editors December 8, 1983, and in revised form December 28, 1984.
† Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Tokyo, Japan 113.
‡ Current address. Department of Mechanical Engineering, School of Science and Technology, Sophia University, Tokyo, Japan 102.
role, and further on a linear-time set union algorithm developed by Gabow and Tarjan [11]. For an intersection graph of $n$ horizontal and vertical segments, our data structures enable us to execute, on-line, a sequence of $O(n)$ LIST1's and DELETE's in $O(n \log n)$ time and space.

We can develop efficient algorithms not only for depth first search and breadth first search but also for various other graph problems. Specifying the results obtained in this paper for an intersection graph of $n$ horizontal and vertical segments with $m$ pairs of intersecting segments, we have the following, where $N = \min \{m, n \log n\}$.

(i) Depth first search and breadth first search can be executed in $O(n \log n)$ time and $O(N)$ space.

(ii) The biconnected components can be found in $O(n \log n)$ time and $O(N)$ space.

(iii) A maximum matching and a maximum independent set can be found in $O(\sqrt{n} N)$ time and $O(N)$ space when no two horizontal (vertical) segments intersect.

(iv) The connectivity $k_G$ can be found in $O(k_G n^{3/2} N)$ time and $O(N)$ space.

These results can be applied to various practical problems, among which the following two practical problems are taken up in the paper.

(I) The problem of finding a minimum dissection of rectilinear region which consists of $l$ sides and may have holes (Lipski et al. [16], Ohtsuki [20]). This problem arises in manipulation of VLSI artwork data. By applying our maximum matching algorithm, we can solve the problem in $O(l^{3/2} \log l)$ time and $O(l \log l)$ space (in fact, we can solve the problem a bit more efficiently as discussed in §7.1).

(II) The problem of determining whether or not a set of $n$ point pairs can be wired in Manhattan fashion on a single layer (Masuda et al. [17], Raghavan et al. [21]). We can solve this problem in $O(n \log n)$ time.

2. Extracting basic operations for depth first search and breadth first search. We first consider, as an example, the depth first search of a directed graph $G = (V_G, A_G)$ with vertex set $V_G$ and arc set $A_G$. Let $n = |V_G|$ and $m = |A_G|$. An undirected graph $G = (V_G, E_G)$ with vertex set $V_G$ and edge set $E_G$ is regarded as a directed graph obtained from $G$ by replacing each edge by two reversely-oriented arcs which connect the same vertices. In Fig. 2.1, we give a procedure SEARCH for finding a depth first spanning forest, which is represented by $p(\ )$, and computing $dfnumber(\ )$ (Tarjan [24]).
procedure SEARCH;
  procedure DFS (u);
  begin
    dfnumber (u) := k; k := k + 1;
    while there is an arc (u, v) ∈ A with v ∈ W do
      begin
        remove v from W; p(v) := u;
        DFS (v)
      end
  end;
begin
  k:= 1; W:= V;
  while W do
    begin
      take an element w out of W; p(w):= nil;
      DFS (w)
    end
end;

FIG. 2.1. The procedure SEARCH.

In ordinary implementations of the procedure, a vertex removed from W is labeled (or, marked), and, in DFS (u), we scan each arc (u, v) going out of u and check whether v is labeled or not. If this technique is employed and the graph is represented by adjacency lists, the procedure can be executed in O(m) time and space.

We can implement the procedure in a different way. For this purpose, several procedures are introduced. For W ⊆ V, we denote by G(W) the graph obtained from G by deleting arcs coming into vertices in V - W. A data structure representing G(W) for W ⊆ V is kept in the course of the algorithm. For v ∈ W, DELETE (v, W) is a procedure which makes W := W - {v} and updates the data structure for G(W). For u ∈ V, LIST1 (u, W) is a function which returns a vertex v such that (u, v) is an arc in G(W) if such v exists, and returns nil otherwise. Also, for u ∈ V, LIST_DEL (u, W) is a function which returns a set {v|(u, v) is an arc in G(W)} and executes DELETE (v, W) for each v in the set.

The procedure SEARCH can be executed by DELETE and LIST1. That is, in the main part of the procedure, we execute DELETE (w, W), and, in DFS (u), we execute v := LIST1 (u, W) and DELETE (v, W). LIST1 and DELETE are executed O(n) times and the other parts of the procedure take O(n) time. Hence, we obtain the following.

PROPOSITION 2.1. Suppose that a sequence of O(n) LIST1's and DELETE's can be executed, on-line, in gT(n, m)(=Ω(n)) time and gS(n, m)(=Ω(n)) space. Then, the depth first search can be executed in O(gT(n, m)) time and O(gS(n, m)) space.

Of course, LIST1 and DELETE can be executed by representing G with adjacency lists. In such a case, we trivially have gT(n, m), gS(n, m) = O(m). However, if G has some prescribed properties, it may be possible to make gT(n, m) and gS(n, m) smaller. In § 3, we show that this is the case for an intersection graph of horizontal and vertical segments.

Similar techniques can be applied to the breadth first search. The problem we consider is to execute a breadth first search on G from a set S of vertices (S ⊆ V), which computes, for each v ∈ V, the length d(v) of a shortest directed path from S to v in G. Here, the length of a directed path is the number of arcs contained in it. Note that d(s) = 0 for s ∈ S, and d(v) = ∞ if there is no directed path from a vertex in S to v. In Fig. 2.2, a procedure BFS is given, where Q is a queue consisting of vertices in V.

```plaintext
procedure SEARCH;
  procedure DFS (u);
  begin
    dfnumber (u) := k; k := k + 1;
    while there is an arc (u, v) ∈ A with v ∈ W do
      begin
        remove v from W; p(v) := u;
        DFS (v)
      end
  end;
begin
  k:= 1; W:= V;
  while W do
    begin
      take an element w out of W; p(w):= nil;
      DFS (w)
    end
end;
```
procedure BFS;
begin
  for each \( s \in S \) do \( d(s) := 0 \);
  let \( Q \) be a queue consisting of all vertices in \( S \); \( W := V_G - S \);
  while \( Q \neq \emptyset \) do
    begin
      take an element \( u \) out of \( Q \); \( V_u := \text{LIST\_DEL}(u, W) \);
      for each \( v \in V_u \) do begin \( d(v) := d(u) + 1 \); insert \( v \) into \( Q \) end
    end;
end;

FIG. 2.2. The procedure BFS (breadth first search from the set \( S \) of vertices).

Then, we have the following, where it is noted that a sequence of \( O(n) \) LIST\_DEL's can be executed by a sequence of \( O(n) \) LIST1's and DELETE's.

**Proposition 2.2.** Suppose that a sequence of \( O(n) \) LIST1's and DELETE's can be executed in \( g_T(n, m)(=\Omega(n)) \) time and \( g_S(n, m)(=\Omega(n)) \) space. Then, the breadth first search can be executed in \( O(g_T(n, m)) \) time and \( O(g_S(n, m)) \) space.

3. The data structures. Let \( H \) (resp. \( V \)) be a set of horizontal (resp. vertical) line segments in the \((x, y)\)-plane. Let \( n_h = |H| \), \( n_v = |V| \) and \( n = n_h + n_v \). We denote by \( G_S(H, V) \) the intersection graph of these horizontal and vertical segments. We say that \( G_S(H, V) \) is geometrically bipartite (abbreviated as g-bipartite) if there are no pair of intersecting horizontal segments and no pair of intersecting vertical segments. \( G_S(H, V) \) is an undirected graph, and, if \( G_S(H, V) \) is g-bipartite, \( G_S(H, V) \) is a bipartite graph.

In this section, we present efficient data structures for \( G_S(H, V) \), which are based both on the data structures for a special kind of one-dimensional range search problem and on segment trees, so that LISTI and DELETE introduced in §2 can be executed quickly. We first consider the problems for g-bipartite graph \( G_S(H, V) \), and then consider the general case.

3.1. A special kind of one-dimensional range search problem. Suppose that, on the \( x \)-axis, a set \( S \) of points \( P_i = (x_i) (i = 1, \ldots, p) \) and a set \( I \) of intervals (ranges) \( R_k = [l_k, r_k] (k = 1, \ldots, q) \) are given. It is also supposed that \( x_i (i = 1, \ldots, p) \) are distinct, and that the values of \( x_i, l_k \) and \( r_k \) are given in a sorted order by \( O((p + q) \log (p + q)) \)-time preprocessing. Further, for brevity, we provide dummy points \( P_0 \) and \( P_{p+1} \) in \( S \) such that \( x_0 = -\infty \) and \( x_{p+1} = +\infty \). For these points and intervals, we are concerned with the problem of executing the following procedures DEL and L1 efficiently. For \( P_i \in S \), \( \text{DEL}(P_i, S) \) is a procedure which removes \( P_i \) from the set \( S \). For \( R_k \in I \), \( \text{L1}(R_k, S) \) is a function which returns a point \( P \in S \) contained in the interval \( R_k \) if such a point exists, and returns nil otherwise. In fact, \( \text{L1}(R_k, S) \) given below returns a point which is leftmost among points included in \( R_k \).

By means of the set union algorithm developed by Gabow and Tarjan [11], we can execute a sequence of \( O(r) \) L1's and DEL's in \( O(p + q + r) \) time as follows. We keep the set \( S \) of points \( P_i \) by a doubly linked list \( L \) in increasing order with respect to their \( x \)-coordinates. For \( P_i \in S \), let \( P_{p(i)} \) be the predecessor element of \( P_i \) in the list \( L \). \( P_{p(i)} \) is a point that is rightmost among points in \( S \) lying in the left side of \( P_i \). Then, define \( R(P_i) \) to be a set of all intervals \( R_k \in I \) whose left endpoints lie in the interval \( (x_{p(i)}, x_i] \) (i.e., \( l_k \in (x_{p(i)}, x_i] \)). By definition, \( R(P_i)'s \) \((P_i \in S)\) are disjoint sets. For a pair of consecutive points \( P_i \) and \( P_{p(i)} \) in the list \( L \), consider a procedure \( \text{UNION}(P, P_i) \) which makes \( R(P_i) := R(P_i) \cup R(P) \) and \( R(P) := \emptyset \). For \( R_k \in I \), consider a function \( \text{FIND}(R_k) \) which returns a point \( P \in S \) such that \( R_k \in R(P) \), where \( P \) is uniquely determined (recall that we consider dummy points \( P_0 \) and \( P_{p+1} \)). Then, DEL and L1 can be executed as in Fig. 3.1.
procedure DEL \((P_n, S)\);
begin
    let \(P _{s(i)}\) be the successor element of \(P_i\) in the list \(L\);
    remove \(P_i\) from the set \(S\) (i.e., the list \(L\));
    UNION \((P_n, P _{s(i)})\)
end;
function \(L1(R_k, S)\): an element in \(S\);
begin
    \(P := \text{FIND}(R_k)\); (recall \(R_{I/k}, ra))
    if \(x_i > r_k\) then return nil
    else return \(P_i\)
end;

FIG. 3.1. The procedure DEL and the function L1.

In Fig. 3.2, an example is given. The validity of DEL and L1 is evident. Since we know the structure of the UNION’s in advance (i.e., the union tree is a path, which is the simplest case; see [11]), and UNION and FIND are executed \(O(r)\) times in the sequence, we can execute UNION and FIND in \(O(p + q + r)\) time in total, with \(O(p + q)\) preprocessing time (Gabow and Tarjan [11]). Hence, we obtain the following.

**Theorem 3.1.** A sequence of \(O(r)\) \(L1\)’s and DEL’s for \(p\) points and \(q\) intervals can be executed in \(O(p/ q + r)\) time, using \(O(p + q)\) preprocessing time.

FIG. 3.2. An example.

3.2. Segmentation. Our data structure below is essentially based on the partitions of segments as induced by segment trees (Bentley [2]), which are now outlined with a slight modification.

We first normalize the \(y\)-coordinates of segments in the following way. We shorten vertical segments as much as possible so long as, for each vertical segment, the set of intersecting horizontal segments does not change. Supposing that there are \(n_y\) distinct \(y\)-coordinates of endpoints of segments, we then normalize the \(y\)-coordinates of segments by replacing them by their ranks from 1 to \(n_y\) in the set (there, of course, may be ties). Similarly, the \(x\)-coordinates of segments are normalized.

For an integer interval \([a, b]\), a segment tree \(T(a, b)\) consists of a root \(r\) with interval \([a, b]\), and in the case of \(b-a \geq 1\), of a left subtree \(T(a, [(a+b)/2])\) and a right subtree \(T([(a+b)/2] + 1, b)\); in the case of \(b-a = 0\), the left and right subtrees are empty. In Fig. 3.3, the tree \(T(1, 6)\) is depicted, where each node is labeled with its associated interval. We call an interval which equals interval \([i, j]\) associated with some node of \(T(1, n_y)\) a standard interval, and denote it by \(I_{ij}\). The partition of a segment \([a, b]\) \((1 \leq a \leq b \leq n_y)\) is a collection of standard intervals contained in \([a, b]\) such that

(i) each of \(a, a+1, a+2, \ldots, b\) is in exactly one standard interval in the collection, and
(ii) for each standard interval in the collection, which is associated with node $n_T$ in $T(1, n_y)$, a standard interval associated with a father of $n_T$ in $T(1, n_y)$ is not contained in $[a, b]$.

In Fig. 3.4, the partitions of vertical segments depicted in Fig. 1.1(a) are shown. The partition of a segment consists of $O(\log n_y)$ standard intervals, and can be obtained in $O(\log n_y)$ time.

3.3. Executing DELETE, LIST1 and LIST_DEL. Since a sequence of $O(n)$ LIST_DEL's can be replaced by a sequence of $O(n)$ LIST1's and DELETE's, we only consider a sequence of $O(n)$ LIST1's and DELETE's in the following. We first consider how to execute LIST1($h, H \cup V$) for $h \in H$ and DELETE($v, H \cup V$) for $v \in V$ in g-bipartite $G_6(H, V)$. LIST1 for $v \in V$ and DELETE for $h \in H$ can be executed similarly. Consider the partitions of all the vertical segments in $V$. Then, with each standard interval $I_{i,j}$, we can associate a set $V(I_{i,j})$ of vertical segments whose partitions contain the standard interval $I_{i,j}$ (In Fig. 3.4, $V(I_{4,6}) = \{v_5, v_7\}$.) For each horizontal segment $h$, consider a set $I_H(h)$ of standard intervals $I_{i,j}$ such that the y-coordinate of $h$ is contained in the standard interval $I_{i,j}$. (In Fig. 3.4, $I_H(h_2) = \{I_{1,6}, I_{1,3}, I_{1,2}, I_{2,2}\}$.) For a standard interval $I_{i,j}$, consider a set $H_I(I_{i,j})$ of horizontal segments $h \in H$ such that the y-coordinate of $h$ is contained in the standard interval $I_{i,j}$. (In Fig. 3.4, $H_I(I_{4,6}) = \{h_4, h_5, h_6\}$.)

For each standard interval $I_{i,j}$, by projecting the set $V(I_{i,j})$ of vertical segments and the set $H_I(I_{i,j})$ of horizontal segments on the x-axis, we can identify each vertical segment with a point and each horizontal segment with an interval on the x-axis. Therefore, we can consider DEL($v, V(I_{i,j})$) for $v \in V(I_{i,j})$ and L1($h, V(I_{i,j})$) for $h \in H_I(I_{i,j})$ in an obvious way. For each standard interval $I_{i,j}$, we keep the data structure, which is discussed in § 3.1, for the set $V(I_{i,j})$ of points and the set $H_I(I_{i,j})$ of intervals in order to execute DEL($v, V(I_{i,j})$) and L1($h, V(I_{i,j})$). The data structure can be
constructed in $O(n \log n)$ time, and takes $O(n \log n)$ space in total. Then, DELETE and LIST1 can be executed as in Fig. 3.5.

```
procedure DELETE (v, H U V);
begin
  for each $I_{i,j}$ such that $v \in V_i(I_{i,j})$ do DEL (v, $V_i(I_{i,j})$)
end;
function LIST1 (h, H U V): an element of V;
begin
  while $I_H(h) \neq \emptyset$ do
  begin
    let $li$ be an element of $I_H(h)$;
    v := L1 ($h$, $V_i(I_{i,j})$);
    if $v \neq$ nil then return v
    else $I_H(h) := I_H(h) \setminus \{I_{i,j}\}$
  end;
  return nil
end;
```

**FIG. 3.5.** DELETE and LIST1.

Let us evaluate the time complexity to execute a sequence of $O(n)$ LIST1's for $h \in H$ and DELETE's for $v \in V$ given in Fig. 3.5. First, note that, in LIST1, $|I_H(h)| = O(\log n)$ and, in DELETE, $|\{I_{i,j} | v \in V_i(I_{i,j})\}| = O(\log n)$. Suppose that, in the sequence, L1 and DEL concerning $V_i(I_{i,j})$ and $H_i(I_{i,j})$ are executed $n(I_{i,j})$ times for each standard interval $I_{i,j}$. Then, the total complexity required by the whole LIST1's and DELETE's is the complexity for executing a sequence of $O(n(I_{i,j}))$ L1($h$, $V_i(I_{i,j})$)’s and DEL($v$, $V_i(I_{i,j})$)’s for all standard intervals $I_{i,j}$. For each $I_{i,j}$, the sequence of L1’s and DEL’s can be executed in $O(n(I_{i,j}) + |V_i(I_{i,j})| + |H_i(I_{i,j})|)$ time by Theorem 3.1. Since $\sum (n(I_{i,j}) + |V_i(I_{i,j})| + |H_i(I_{i,j})|) = O(n \log n)$ where the summation is taken over all standard intervals $I_{i,j}$, the total complexity to execute the whole L1’s and DEL’s is $O(n \log n)$. Thus, the sequence of $O(n)$ LIST1’s for $h \in H$ and DELETE’s for $v \in V$ can be executed in $O(n \log n)$ time and $O(n \log n)$ space. By considering partitions of horizontal segments, a sequence of $O(n)$ LIST1’s for $v \in V$ and DELETE’s for $h \in H$ can be executed similarly in $O(n \log n)$ time and $O(n \log n)$ space.

Next, consider the case $G_S(H, V)$ is not $g$-bipartite. In this case, we must find pairs of intersecting horizontal segments and of intersecting vertical ones. Here, we consider the problem for horizontal segments only. First, note that two horizontal segments intersect only if their $y$-coordinates are the same. The set of horizontal segments is partitioned into disjoint subsets so that two segments in the same subset iff their $y$-coordinates are the same. Then, we have only to find pairs of intersecting segments in each subset. The problem for each subset is just the one-dimensional interval intersection problem, for which efficient data structures such as a tile tree in McCreight [18] and an interval tree in Edelsbrunner [6], [7] have been developed. Given a set $I$ of $p$ intervals on a line, these data structures report a set $I'$ of all intervals in $I$ that intersect a query interval in $O(\log p + |I'|)$ time, and delete an interval in $I$ from the data structure in $O(\log p)$ time (in fact, in our case, we can delete it in a constant time). The data structures take $O(p)$ space.

Thus, we obtain the following.

**Proposition 3.1.** A sequence of $O(n)$ LIST_DEL’s, LIST1’s and DELETE’s for $G_S(H, V)$ can be executed in $O(n \log n)$ time and $O(n \log n)$ space.

The above algorithms may be worse than naive algorithms in case, for instance, $m = \Theta(n)$ ($m$ is the number of arcs in $G_S(H, V)$). However, by processing $G_S(H, V)$
Given the set of horizontal and vertical segments, we first compute the number \( m \) of pairs of intersecting segments, which can be done in \( O(n \log n) \) time and \( O(n) \) space (Bentley and Ottmann [3]). If \( m \leq n \log n \), the intersection graph \( G_S(H, V) \) is constructed by enumerating all pairs of intersecting segments, which can be done in \( O(n \log n + m) \) time and \( O(n + m) \) space [3], and then the problem is solved by ordinary adjacency lists as the data structures. Otherwise, the problem is solved by the data structures given above. Thus, we obtain the following.

**Theorem 3.2.** A sequence of \( O(n) \) LIST_DEL's, LIST1's and DELETE's for an intersection graph of \( n \) horizontal and vertical segments with \( m \) pairs of intersecting segments can be executed in \( O(N) \) time and \( O(N) \) space with \( O(n \log n) \) preprocessing, where \( N = \min \{m, n \log n\} \).

### 4. Simple graph search problems
In the following §§ 4–6, we consider that, for a graph \( G \) with \( n \) vertices and \( m \) edges, a sequence of \( O(n) \) LIST1's and DELETE's can be executed in \( g_T(n, m) \) time and \( g_S(n, m) \) space (\( g_T(n, m), g_S(n, m) = \Omega(n) \)), and estimate the complexity of algorithms for various graph problems. In this section, simple graph search problems are investigated. By Theorem 3.2, for an intersection graph \( G_S(H, V) \) of \( n \) horizontal and vertical segments with \( m \) pairs of intersecting segments, \( g_T(n, m) = O(n \log n) \) when \( O(n \log n) \) preprocessing is done in advance, where \( N = \min \{m, n \log n\} \).

#### 4.1. Depth first search, breadth first search and the connected components
By Propositions 2.1 and 2.2, depth first search and breadth first search for the graph \( G \) can be executed in \( O(g_T(n, m)) \) time and \( O(g_S(n, m)) \) space. Hence, we obtain the following.

**Theorem 4.1.** Depth first search and breadth first search for \( G_S(H, V) \) can be executed in \( O(n \log n) \) time and \( O(N) \) space.

As is well known, the connected components of \( G \) can be found by executing depth first search or breadth first search, hence we obtain the following.

**Corollary 4.1.** The connected components of \( G_S(H, V) \) can be found in \( O(n \log n) \) time and \( O(N) \) space.

However, concerning the problem of finding the connected components of \( G_S(H, V) \), an \( O(n \log n) \)-time and \( O(n) \)-space algorithm is known (Edelsbrunner et al. [8], Imai and Asano [14]) which employs the so-called plane-sweep techniques.

#### 4.2. The biconnected components
We first show that the biconnected components of an undirected graph \( G = (V_G, E_G) \) with vertex set \( V_G \) and edge set \( E_G \) can be found in \( O(g_T(n, m)) \) time and \( O(g_S(n, m)) \) space (by the biconnected components, we here mean the decomposition of the vertex set obtained by decomposing \( G \) into the biconnected components). We suppose that \( G \) is connected. By executing depth first search for the graph, a depth first spanning tree represented by \( p( \) and the values of \( df \)number \( (u) \) for \( u \in V_G \) are found, which takes in \( O(g_T(n, m)) \) time and \( O(g_S(n, m)) \) space. Then, the main problem for the biconnected components is to compute \( low(u) \) for each vertex \( u \) defined as follows (Tarjan [24]).

\[
low(u) = \min \{\{dfmin(u)\} \cup \{low(s) | s \in V_G, p(s) = u\}, \quad \text{where}
\]

\[
dfmin(u) = \min \{dfnumber(v) | v = u \text{ or } \{u, v\} \text{ is an edge in } G \text{ with } v \neq p(u)\}.
\]

If the values of \( dfmin(u) \) for all \( u \) can be computed, we can find in \( O(n) \) time and space the values of \( low( ) \) by traversing the tree in postorder, and then we can find
the biconnected components in $O(n)$ time and space [24]. So, the problem is to find the values of $dfmin(u)$ for all vertices $u$. In Fig. 4.1, an algorithm for this problem is given. Note that each edge $\{u, v\}$ of $G$ is considered to be two reversely-oriented arcs $(u, v)$ and $(v, u)$ in the procedure.

**procedure** COMPUTE_DFMN;

begin

$W := V_G$; let $u$ be a vertex with $dfnumber(u) = 1$;

dfmin($u$) := 1; DELETE ($u$, $W$);

for $i := 1$ to $n$ do

begin

let $v$ be a vertex with $dfnumber(v) = i$; $U := LIST_{DEL}(v, W)$;

for each $u \in U$ do if $v = p(u)$ then $dfmin(u) := dfnumber(u)$

else $dfmin(u) := i$

end

end;

**FIG. 4.1.** The procedure COMPUTE_DFMN.

Since it can be considered that the values of $dfnumber(u)$ for $u \in V_G$ have been sorted in executing depth first search, we obtain the following.

**Proposition 4.1.** The biconnected components of $G$ can be found in $O(g_T(n, m))$ time and $O(g_S(n, m))$ space.

Then the following is obtained by applying the proposition to graph $G_S(H, V)$.

**Theorem 4.2.** The biconnected components of $G_S(H, V)$ can be found in $O(n \log n)$ time and $O(N)$ space.

5. Maximum flow algorithms. In this section, we consider the problem of finding a maximum flow in a directed graph $G = (V_G, A_G)$ with unit capacities such that, for each vertex, there is at most one arc that emanates from it or at most one arc that comes into it. We estimate the complexity of Ford and Fulkerson’s and Dinic’s algorithms for this problem, based on the complexity to execute the basic operations, LIST1 and DELETE, for graphs obtained by modifying the original graph $G$ in the course of the algorithms.

Let $S$ and $T$ be disjoint subsets of $V_G$, where it is supposed that, for each vertex in $S$ (resp. $T$), there is the only one arc emanating from (resp. coming into) it. We consider the problem of finding a maximum (integral) flow from $S$ to $T$ in the graph $G$ with unit capacities. This problem is equivalent to that of finding a maximum vertex-disjoint directed paths from $S$ to $T$ in $G$. Note that the value of a maximum flow is at most $n/2$.

We first describe the basic part of the maximum flow algorithms. Let $f$ be an integral flow on $G$, that is, for each arc $a \in A_G$, $f(a) = 0$ or $1$, and, for each vertex $v \in V_G - (S \cup T)$, $\sum_{(u,v) \in A_G} f(a) = \sum_{(v,u) \in A_G} f(a)$. Define an auxiliary graph $G(f)$ with vertex set $V_G$ and arc set $\hat{A} = A_G \cup A_1$ by $A_0 = \{a | a \in A_G, f(a) = 0\}$, $A_1 = \{a' | a \in A_G, f(a) = 1, a': \text{reorientation of } a\}$. Let $S(f) = \{s \in S, \text{there is an arc emanating from } s \in G(f)\}$ and $T(f) = \{t \in T, \text{there is an arc coming into } t \in G(f)\}$. If, in $G(f)$, there is a directed path $P$ from $S(f)$ to $T(f)$, we can augment the flow by letting $f(a) := 1$ ($a \in P \cap A_0, f(a) := 0$ ($a' \in P \cap A_1, a': \text{reorientation of } a'$).

In the maximum flow algorithms, we must execute the basic operations, LIST1 and DELETE for auxiliary graphs $G(f)$. Since $G(f)$ can be obtained by slightly modifying the original graph $G$, those operations for $G(f)$ can be executed by those for $G$ (it should be noted that, for each vertex $v$, there is at most one arc $a$ emanating from (coming into) $v$ with $f(a) = 1$ in the graph $G$). Hence, we have the following.
PROPOSITION 5.1. Suppose that a sequence of \(O(n)\) LIST1's and DELETE's for the graph \(G\) can be executed in \(g_T(n, m)\) time and \(g_S(n, m)\) space. Then a sequence of \(O(n)\) LIST1's and DELETE's for the auxiliary graph \(G(f)\) can be executed in \(O(g_T(n, m))\) time and \(O(g_S(n, m))\) space.

5.1. Ford and Fulkerson's algorithm. This algorithm [10] starts with an appropriate integral flow \(f\), which may be zero flow, and iterates to find a directed path from \(S(f)\) to \(T(f)\) in \(G(f)\) and to augment the flow \(f\) until there comes to be no directed path from \(S(f)\) to \(T(f)\) in \(G(f)\). Since the value of a maximum flow is at most \(n/2\) in this graph \(G\), a maximum flow can be found by solving the problem of finding a directed path in the auxiliary graph at most \(n/2\) times. Since the path-finding problem can be solved by executing depth first search or breadth first search, from Proposition 5.1, we obtain the following.

PROPOSITION 5.2. Ford and Fulkerson's algorithm finds a maximum flow in the graph \(G\) in \(O(n g_T(n, m))\) time and \(O(g_S(n, m))\) space.

5.2. Dinic's algorithm. Dinic's algorithm [5] (see also Even and Tarjan [9] and Hopcroft and Karp [13]) consists of phases, where the number of phases is \(O(\sqrt{n})\) for this type of graph \(G\) [9] (see also [13]). Each phase consists of two steps. The first step is to determine the layer of each vertex of \(G(f)\) by breadth first search. The second step is to find a maximal set of vertex-disjoint directed paths in the layered \(G(f)\) by depth first search.

The first step is to execute breadth first search on \(G(f)\) from \(S(f)\) and compute \(d(v)\) for each \(v\) as in § 2. If \(d(t) = \infty\) for each \(t \in T(f)\) (i.e., there is no directed path from \(S(f)\) to \(T(f)\) in \(G(f)\)), \(f\) is a maximum flow. This breadth first search can be done in \(O(g_T(n, m))\) time and \(O(g_S(n, m))\) space by Propositions 2.2 and 5.1.

The second step is subtler than the first step, and is executed as follows. Let \(d^* = \min\{d(v) | v \in T(f)\}\), which is assumed to be finite. Define \(T_d(f)\) to be \(\{t | t \in T(f), d(t) = d^*\}\). Then, the second step of the phase is to find a maximal set of vertex-disjoint directed paths of length \(d^*\) in \(G(f)\) from \(S(f)\) to \(T_d(f)\), where a set is maximal with a given property if it is not properly contained in any set that has the property. For this purpose, we need a new graph \(G(f, d)\), which is defined to be a directed graph with vertex set \(V_G\) and arc set \{(u, v) | (u, v) is an arc in \(G(f)\), \(d(v) = d(u) + 1\)\}. A maximal set of such directed paths in \(G(f)\) as mentioned above can be found by executing the depth first search on \(G(f, d)\) (for details, see [5], [9], [13]). Then, from the arguments in § 2, we see that the problem is to execute the basic operations, LIST1 and DELETE, for the graph \(G(f, d)\). Since, concerning the structure of a graph, \(G(f, d)\) is quite different from \(G\), we cannot generally obtain a proposition, similar to Proposition 5.1, for this graph \(G(f, d)\), Considering the complexity to execute those basic operations for \(G(f, d)\) as well as that for the original graph \(G\), we obtain the following.

PROPOSITION 5.3. Suppose that \(O(n)\) LIST1's and DELETE's for the graph \(G(f, d)\) can be executed in \(h_T(n, m)\) time and \(h_S(n, m)\) space. Then, Dinic's algorithm finds a maximum flow in the graph \(G\) in \(O(\sqrt{n}(g_T(n, m) + h_T(n, m)))\) time and \(O(g_S(n, m) + h_S(n, m))\) space.

That is, it is a little complicated to implement Dinic's algorithm, since it requires data structures for executing a sequence of LIST1's and DELETE's not only for the graph \(G\) but also for the graph \(G(f, d)\).

5.3. A maximum matching. In this section, we consider the problem of finding a maximum matching of \(g\)-bipartite \(G_S(H, V)\). \(G_S(H, V)\) is considered to be a directed
bipartite graph \((H, V; A_G)\) with left vertex set \(H\), right vertex set \(V\) and arc set \(A_G \subseteq H \times V\). For this \(G_S(H, V)\), construct a directed graph \(G\) by adding copies \(H'\) and \(V'\) of \(H\) and \(V\), respectively, to \(G_S(H, V)\), and making an arc \((h', h)\) for each \(h \in H\) and an arc \((v, v')\) for each \(v \in V\). As is well known, the problem of finding a maximum matching of \(G_S(H, V)\) is equivalent to that of finding a maximum integral flow from \(H'\) to \(V'\) in \(G\) with unit capacities, hence the arguments in \(\S\S\ 5.1\) and 5.2 can be applied.

Concerning the time and space complexities \(g_T(n, m)\) and \(g_S(n, m)\), respectively, for executing a sequence of \(O(n)\) LIST1's and DELETE's for this \(G\), by using the data structure for \(G_S(H, V)\) presented in \(\S\ 3\), we have \(g_T(n, m), g_S(n, m) = O(N)\) with \(O(n \log n)\) preprocessing. (Thus, by Ford and Fulkerson’s algorithm with the data structure in \(\S\ 3\) and from Proposition 5.2, we can find a maximum matching of \(G_S(H, V)\) in \(O(n^2)\) time and \(O(N)\) space.)

Concerning the time and space complexities \(h_T(n, m)\) and \(h_S(n, m)\), respectively, for executing a sequence of \(O(n)\) LIST1's and DELETE's for \(G(f, d)\) obtained from this \(G\), we can no more use the data structure in \(\S\ 3\) directly, because it cannot handle the condition on \(d\) efficiently. However, we can modify it, in the following way, so as to work well even for the graph \(G(f, d)\). For each standard interval \(I_{i,j}\), let \(D(I_{i,j}) = \{d(h) | h \in H_f(I_{i,j})\}\). For each \(k \in D(I_{i,j})\), define \(H(k, I_{i,j})\) and \(V(k, I_{i,j})\) by

\[
H(k, I_{i,j}) = \{h | h \in H_f(I_{i,j}), d(h) = k\}, \quad V(k, I_{i,j}) = \{v | v \in V_f(I_{i,j}), d(v) = k + 1\}.
\]

Since the \(H(k, I_{i,j})\)'s (resp. \(V(k, I_{i,j})\)'s) for all \(k \in D(I_{i,j})\) are disjoint sets. Hence, a sequence of \(O(n)\) LIST1's for \(h \in H\) and DELETE's for \(v \in V\) on \(G_f(d)\) can be executed in \(O(n \log n)\) time and space by providing, for each standard interval \(I_{i,j}\) and \(k \in D(I_{i,j})\), the data structure given in \(\S\ 3.1\) for \(V(k, I_{i,j})\) and \(H(k, I_{i,j})\). We can construct the data structure in \(O(n \log n)\) time (in partitioning \(H(k, I_{i,j})\) and \(V(k, I_{i,j})\) into \(H(k, I_{i,j})\)'s and \(V(k, I_{i,j})\)'s \((k \in D(I_{i,j}))\), we use a technique similar to the so-called bucket sort). Concerning LIST1 for \(v \in V\) and DELETE for \(h \in H\), since there is at most one arc going out of \(v\) in \(G(f, d)\) owing to the bipartite structure of \(G_S(H, V)\), we can execute each LIST1 and DELETE in a constant time. Then, concerning \(h_T(n, m)\) and \(h_S(n, m)\), applying the arguments similar to Theorem 3.2, we have \(h_T(n, m), h_S(n, m) = O(N)\). Then, the following is obtained from Proposition 5.3.

**Theorem 5.1.** A maximum matching of \(g\)-bipartite \(G_S(H, V)\) can be found in \(O(\sqrt{n} N)\) time and \(O(N)\) space by Dinic’s algorithm with the data structures presented above.

**5.4. The connectivity.** For \(G_S(H, V)\), consider the directed graph \(G\) obtained by splitting each vertex \(u\) to \(u^-\) and \(u^+\), and making arc \((u^+, u^-)\), and making arcs \((u^-_i, u^+_j)\) iff there is an edge \([u_i, u_j]\) in \(G_S(H, V)\). The connectivity \(k_G\) of \(G_S(H, V)\) can be found by solving the problem of finding a maximum flow in the graph \(G\) repeatedly. Further, in solving the maximum flow problem on \(G\) by Dinic’s algorithm, we can execute DELETE and LIST1 for \(G\) by those for \(G_S(H, V)\), and we can apply the data structures developed here to this problem. That is, we can solve the maximum flow problem on this graph \(G\) with unit capacities in \(O(\sqrt{n} N)\) time and \(O(N)\) space.

Even and Tarjan’s algorithm [9] solves \(O(k_G n)\) times the maximum flow problem on \(G\) for \(G_S(H, V)\). Thus, we can find the connectivity \(k_G\) of \(G_S(H, V)\) in \(O(k_G \sqrt{n} N)\) time and \(O(N)\) space. Also, by applying Galil’s algorithm [12], we can find \(k_G\) in \(O(k_G \sqrt{n} N \max k_G, \sqrt{n})\) time and \(O((k_G^2 + n) n)\) space (note that, since vertex-disjoint directed paths are concerned, the information of each phase in Dinic’s algorithm can be kept in \(O(n)\) space not by \(\Theta(m)\) space).
Stating the former result as a theorem, we have the following.

**Theorem 5.2.** The connectivity $k_G$ of $G_S(H, V)$ can be found in $O(k_G n^{3/2} N)$ time and $O(N)$ space.

6. An independent set.

6.1. A maximum independent set of $g$-bipartite $G_S(H, V)$. As is well known, a maximum independent set of a bipartite graph can be found by any bipartite matching algorithm which finds a minimum cover of the graph, hence we obtain the following.

**Theorem 6.1.** A maximum independent set of $g$-bipartite $G_S(H, V)$ can be found in $O(\sqrt{n} N)$ time and $O(N)$ space.

6.2. Determining the existence of an independent set of maximally possible size. Let $G = (V_G, E_G)$ be a graph with $n$ vertices as in § 4. Let $X$ be a matching of $G$. For each $u \in V_G$, we define $mate(u)$ to be $v$ if there is an edge $\{u, v\} \in X$ and to be nil otherwise. A vertex $u$ with $mate(u) = \text{nil}$ is called *unmatched*. Apparently, the size of a maximum independent set of $G$ is at most $n - |X|$. Further, if there is an independent set $S$ of $G$ of size $n - |X|$, then, for each edge $\{u, v\} \in X$, exactly one of $\{u, v\}$ is an element of $S$, and each unmatched vertex is an element of $S$.

Masuda et al. [17] gave an $O(m)$-time and $O(m)$-space algorithm for determining whether or not there is an independent set of size $n - |X|$. Their algorithm executes the depth first search on $G$ by stretching alternating paths (an alternating path is a path of edges which are alternately in $X$ and not in $X$). In this section, we give an efficient implementation of their algorithm, which solves the problem for $G$ in $O(g_T(n, m))$ time and $O(g_S(n, m))$ space.

Following their algorithm, we color vertices as follows. A vertex $v$ that is found to be in every independent set (resp. no independent set) of size $n - |X|$ is colored *green* (resp. *red*). A vertex $v$ contained (resp. not contained) in the current candidates for such an independent set is colored *blue* (resp. *orange*). The other vertices $v$ that are unsearched are colored *white*. In Fig. 6.1, a procedure INDEPENDENT_SET for the problem is given.

The outline of the algorithm is as follows. Since unmatched vertices must be in every independent set of size $n - |X|$, we first color these vertices $v$ green. If vertex $v$ is colored green (in such a case, the procedure GREEN ($v$) is called), a set $V_v$ of vertices adjacent to $v$ must be colored red. Then, for each $w \in V_v$, vertex $mate(w)$ must be colored green, and so, coloring $mate(w)$ green, we apply this procedure recursively. It should be noted that, after the execution of GREEN ($v$) for unmatched vertices $v$, all vertices colored white are matched. Then we take a vertex $v$ colored white as a candidate, and color $v$ blue. Starting with $v$, we execute the depth first search (the procedure BLUE ($v$)). If a vertex $u$ adjacent to $v$ is still colored white, we color $u$ orange and vertex $u' = mate(u)$ blue, and recursively execute depth first search from $u'$. If a vertex $u$ adjacent to $v$ is already colored blue, there is a path $u = u_0, u_1, u_2, \ldots, u_{2k-1}, u_{2k} = v$ such that $u_{2i} = mate(u_{2i-1})$, color($u_{2i}$) = blue and color($u_{2i-1}$) = orange (this is because BLUE essentially executes the depth first search). If $u = u_0$ is in an independent set $S$ of size $n - |X|$, $u_1$ cannot be in $S$, and then $u_2$ must be in $S$, $\ldots$, and $u_{2k} = v$ must be in $S$. However, there has been found an edge connecting $u$ and $v$, which contradicts the fact that $S$ is an independent set. Thus, in this case, we see that $u$ cannot be in any independent set of size $n - |X|$, and we color $u$ red, and $u' = mate(u)$ green, and execute GREEN ($u'$).

In the procedure, we consider operations, LIST_DEL and DELETE, for the set $S$ on $G$, and operations, LIST1 and DELETE, for the set $T$ on $G$ (we must treat them separately). In the course of the algorithm, $V_G - S$ is a set of vertices colored red, and,
procedure INDEPENDENT_SET;
procedure GREEN (v);
begin
V_v := LIST_DEL (v, S); U := \emptyset;
for each w \in V_v do
begin
if color(w) = green then
begin
report "There is not an independent set of size n - |X|"; halt
end;
end;
if w \in T then DELETE (w, T); if u \in T then DELETE (u, T)
end;
for each u \in U do GREEN (u)
end;
procedure BLUE (v);
begin
u := LIST1 (v, T);
if u = \null then
begin
DELETE (u, T); u' := mate(u);
if color(u) = blue then
begin
color(u') := green; color(u) := red;
DELETE (u, S); GREEN (u')
end
else begin color(u) := orange; color(u') := blue; BLUE (u') end;
if color(v) = blue then BLUE (v)
end
end;
begin
S := V_G; T := V_G; for each v \in V_G do color(v) := white;
for each unmatched vertex v do begin color(v) := green; DELETE (v, T) end;
for each unmatched vertex v do GREEN (v);
while there is a vertex colored white in T do
begin
let v be a vertex colored white in T; v' := mate(v); DELETE (v', T);
if color(v') = orange; color(v) := blue; BLUE (v)
end;
report "There is an independent set of size n - |X|"
end;

FIG. 6.1. The procedure INDEPENDENT_SET.

for each u \in V_G - S, vertex mate(u) is colored green. T is a set of vertices colored blue or white. If the algorithm reports "There is an independent set of size n - |X|," then \{v | color(v) = green or blue\} is an independent set of size n - |X| at the end of the algorithm.

Next, consider the complexity of the procedure. Since BLUE essentially executes the depth first search, LIST1 and DELETE for T and DELETE for S in BLUE are executed O(n) times. Hence, we can execute all the BLUE's except GREEN's in them in O(g_T(n, m)) time and O(g_S(n, m)) space. GREEN is executed at most once for each v \in V_G. Hence, the total complexity to execute all GREEN's is the complexity to execute a sequence of O(n) LIST_DEL for S and DELETE for T. The other parts of the algorithm obviously take O(n) time and space. Thus, we obtain the following.

PROPOSITION 6.1. Let G be a graph with n vertices and m edges, to which a matching of size k is given. Then, we can determine whether or not there is an independent set of size n - k in O(g_T(n, m)) time and O(g_S(n, m)) space.
7. Applications.

7.1. Minimum dissection of rectilinear region. In manipulation of VLSI artwork data, there arises the problem of dissecting a rectilinear region into a minimum number of nonoverlapping rectangles (Lipski et al. [16], Ohtsuki [20]). Here, rectilinear regions are polygonal regions bounded only by horizontal and vertical edges which may have "holes." In Fig. 7.1, an example is given.

This problem can be solved as follows. Let \( R \) be a rectilinear region with \( l \) edges. A chord of \( R \) connecting two concave vertices whose \( x \)- or \( y \)-coordinates are the same is called an effective chord. Let \( H \) (resp. \( V \)) be the set of horizontal (resp. vertical) effective chords, where \( n = |H| + |V| \) and there are \( m \) pairs of intersecting effective chords. An intersection graph \( G_{\delta}(H, V) \) of those chords is \( g \)-bipartite. Let \( S \) be a maximum independent set of \( G_{\delta}(H, V) \). We first dissect the region by chords in \( S \) into subregions (note that each subregion has no effective chords). We then dissect each subregion by horizontal chords connecting concave vertices and points on edges. The above algorithm yields a minimum dissection [16], [20].

We now evaluate the complexity of the above algorithm. Given a rectilinear region, we can easily find all effective chords in \( O(l \log l + n) \) time and \( O(l) \) space by the plane-sweep algorithm. A maximum independent set of \( G_{\delta}(H, V) \) can be found in \( O(\sqrt{n} N) \) time and \( O(N) \) space by Theorem 6.1 (\( N = \min \{m, n \log n\} \)). Finally, all the subregions can be dissected by such horizontal chords totally in \( O(l \log l) \) time and \( O(l) \) space by the plane-sweep algorithm. Thus, a minimum dissection of the rectilinear region can be found in \( O(\sqrt{n} N + l \log l) \) time and \( O(l + N) \) space. This improves the previous time bound \( O(n^{3/2} + l \log l) \) in [20], and \( O(n^{3/2} \log n \log n + l \log l) \) in [15], [16].

7.2. Manhattan wiring problem. We have the following problem in the design automation of digital systems (Masuda et al. [17], Raghavan et al. [21]). On the grid, \( n \) pairs of points are given. The Manhattan wiring problem is to connect all the pairs of points by wires along grid lines so that the wires do not intersect one another and no wire has more than one bend. Such a wiring is called a Manhattan wiring [17], [21]. Then the problem we consider is to determine whether there is a Manhattan wiring for given \( n \) pairs of points.

There are two rectilinear segments connecting the pair of points and bent at most once (in the case two points have the same \( x \)- or \( y \)-coordinates, there is the only one), each of which is called an \( M \)-wire of the pair of points. Then, there is a Manhattan wiring iff there is an independent set of size \( n \) in an intersection graph \( G_{M} \) of all the \( M \)-wires. In Fig. 7.2, an example is given.
Let $n_1$ be the number of pairs of points which have the only one $M$-wire, and $n_2 = n - n_1$. The number of $M$-wires is $n_1 + 2n_2$. Since $G_M$ has a trivial matching of size $n_2$, the problem of determining the existence of an independent set of size $(n_1 + 2n_2) - n_2 = n$ can be solved by the algorithm in § 6.2. Apparently, LIST_DEL, LIST1 and DELETE for the graph $G_M$ can be implemented by those for an intersection graph of horizontal and vertical segments of which $M$-wires are composed. Let $m$ be the number of edges in $G_M$, and $N = \min \{m, n \log n\}$. Then, by Proposition 6.1, we can solve the problem in $O(n \log n)$ time and $O(N)$ space. Our result is an improvement compared with an $O(n^3)$-time algorithm in [21] and an $O(n \log n + m)$-time algorithm in [17].

8. Concluding remarks. On the intersection problem of horizontal and vertical segments, and rectangles with sides parallel to the axes, efficient data structures have been developed. For the static problem of reporting all the intersections of a family of given $n$ horizontal and vertical segments with an arbitrary horizontal or vertical query segment, an optimal algorithm due to Chazelle [4] is known. His algorithm takes $O(n \log n)$ preprocessing time, $O(n)$ space and $O((\log n)^2 + k)$ query time, where $k$ is the number of reported intersections. However, for our purpose, the operation to delete a segment from the family is necessary, so that such a static algorithm does not work. McCreight’s priority search tree [19] can be applied to this problem. His data structure is a dynamic one, and takes $O(n)$ space, $O((\log n)^2 + k)$ query time and $O((\log n)^2)$ update (deletion or insertion) time. By his data structure, a sequence of $O(n)$ LIST1’s and DELETE’s for $G_S(H, V)$ can be executed in $O(n \log n)^2$ time and $O(n)$ space. In this paper, by utilizing the special structures of our problems, we have given the data structure which decreases a factor $\log n$ in respect to the time complexity with increasing the same factor in respect to the space complexity.

It is straightforward to generalize the arguments in the paper for intersection graphs of horizontal and vertical segments to the problems for intersection graphs of $n$ rectangles with sides parallel to the axes (and further for graphs of arbitrary boxicity $k$, where the boxicity of a graph $G$ is the smallest $k$ such that $G$ is the intersection graph of hyperrectangles with sides parallel to the axes in $k$-dimensional space [22]). By employing the general data structure developed by Edelsbrunner [7], a sequence of $O(n)$ LIST_DEL’s, LIST1’s and DELETE’s for the graph of rectangles can be executed in $O(n (\log n)^2)$ time and $O(n \log n)$ space. (It may be possible to execute the sequence in $O(n \log n)$ time and space by taking advantage of special properties of the sequence.) Hence, we can also solve the simple graph search problems for the graph of those rectangles in an efficient manner. For instance, the biconnected components of an intersection graph of $n$ rectangles with sides parallel to the axes can be found in $O(n (\log n)^2)$ time and $O(n \log n)$ space (again, by counting the number of
pairs of intersecting rectangles in advance, and enumerating those intersecting rect-
tangles if necessary, the above complexities can be modified, where the algorithms in
Edelsbrunner [6], McCreight [18] and Six and Wood [23] may be used).

In [15], Lipski independently developed an algorithm for the problem of finding
a maximum matching of g-bipartite \( G_S(H, V) \) with \( n \) vertices, and claimed that the
time and space complexities are \( O(n^{3/2} \log n \log \log n) \) and \( O(n \log n) \), respectively.
His algorithm is rather complicated and a little worse with respect to the time complexity
than our maximum matching algorithm. Furthermore, it seems that the data structures
in [15] only would not be enough to execute the bipartite-matching algorithm in [13]
correctly.

Finally, we remark that it is a challenging problem to develop an algorithm for
executing a sequence of \( O(n) \) basic operations, LIST1 and DELETE, for the intersec-
tion graph of horizontal and vertical segments in \( O(n \log n) \) time, using \( O(n) \) space.

Acknowledgments. The authors wish to thank Professor Masao Iri of the University
of Tokyo and Professor David Avis of McGill University for their valuable discussions
and comments on the paper.

REFERENCES

Addison-Wesley, Reading, MA, 1974.
[6] H. Edelsbrunner, A time- and space-optimal solution for the planar all intersecting rectangles problem,
[7] ——, Dynamic data structures for orthogonal intersection queries, Report 59, Institut für Informations-
verarbeitung, Technische Universität Graz, 1980.
[8] H. Edelsbrunner, J. van Leeuwen, Th. Ottmann and D. Wood, Connected components of
orthogonal geometric objects, Report 72, Institut für Informationsverarbeitung, Technische Universi-
tät Graz, 1981.
pp. 507-518.
[14] H. Imai and T. Asano, Finding the connected components and a maximum clique of an intersection
Papers of the Technical Group on Circuits and Systems, CAS 83-20, Institute of Electronics and
Communication Engineers of Japan, 1983. (In Japanese.)
[18] E. M. McCreight, Efficient algorithms for enumerating intersecting intervals and rectangles, CSL-80-9,
pp. 257-276.


