Acta Informatica 15, 329-346 (1981)

Efficient Algorithms for Finding Maximum Matchings in Convex Bipartite Graphs and Related Problems

W. Lipski, Jr.[*], and F.P. Preparata[**]

Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA

Summary. A bipartite graph $G=(A, B, E)$ is convex on the vertex set A if A can be ordered so that for each element b in the vertex set B the elements of A connected to b form an interval of A; G is doubly convex if it is convex on both A and B. Letting $|A|=m$ and $|B|=n$, in this paper we describe maximum matching algorithms which run in time $O(m+nA(n))$ on convex graphs (where $A(n)$ is a very slowly growing function related to a functional inverse of Ackermann's function), and in time $O(m+n)$ on doubly convex graphs. We also show that, given a maximum matching in a convex bipartite graph G, a corresponding maximum set of independent vertices can be found in time $O(m+n)$. Finally, we briefly discuss some generalizations of convex bipartite graphs and some extensions of the previously discussed techniques to instances in scheduling theory.

1. Introduction

Matching problems constitute a traditionally important topic in combinatorics and operations research [9] and have been the object of extensive investigation. Particularly interesting is the problem of finding a maximum matching in a bipartite graph, which is stated as follows: Let $G=(A, B, E)$ be an undirected bipartite graph, where A and B are sets of vertices, and E is a set of edges of the form (a, b) with $a \in A$ and $b \in B$. A subset $M \subseteq E$ is a *matching* if no two edges in M are incident to the same vertex; M is of *maximum cardinality* (or simply, *maximum*) if it contains the maximum number of edges. As noted by Hopcroft and Karp [8], this problem has many applications, such as the chain decomposition of a partially ordered set, the determination of coset representatives in groups, etc. Hopcroft and Karp have also developed the best known algorithm for this problem.

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

[**] Also with the Departments of Electrical Engineering and of Computer Science

A special instance of the problem, with some industrial applications, was originally discussed by Glover [7] and referred to as matching in a convex bipartite graph. A bipartite graph G is *convex* on A if an ordering "$\leqq$" of the elements of A can be found so that for any $b \in B$ and distinct a_1 and a_2 in A (with $a_1 \leqq a_2$)

$$(a_1, b) \in E \quad \text{and} \quad (a_2, b) \in E \;\Rightarrow\; (a, b) \in E \qquad \text{for any } a \in A \qquad \text{such that } a_1 \leqq a \leqq a_2.$$

In other words, G is convex on A when there is an ordering on A such that for any $b \in B$ the set of vertices of A connected to b forms an interval in this ordering.

Glover describes the following practical situation leading to the maximum matching problem in a convex bipartite graph. Suppose a certain product requires one machined part from a set A and a second from a set B. Associated with each part $d \in A \cup B$ is its size $s(d)$. Assume that $a \in A$ can be fitted with $b \in B$ if and only if $|s(a) - s(b)| \leqq \varepsilon$, where ε is a specified tolerance. Clearly, maximizing the number of parts matched corresponds to finding a maximum matching in the convex bipartite graph $G = (A, B, E)$ where $E = \{(a, b)\colon a \in A, b \in B, |s(a) - s(b)| \leqq \varepsilon\}$ (in fact this graph is doubly convex, see Sect. 3).

Here is another example. Suppose there is a set A of skis and a set B of skiers. Assume that each of the skiers in B specifies the smallest and largest skis in A he or she will accept. The problem is to find a maximum assignment of skis to skiers.

Notice that in both examples the ordering "$\leqq$" on A required to exist for a graph to be convex is in fact given (by the values of some parameter associated with each $a \in A$). Since this is typical for applications involving convex bipartite graphs, we shall always assume that the graph $G = (A, B, E)$ is given by specifying the ordering on A and by specifying, for every $b \in B$, the elements BEG[b], END[b], the "smallest" and "largest" elements respectively, of the interval of the elements of A connected to b. Naturally, if $b \in B$ is isolated, this interval is empty and BEG[b] = END[b] = Λ, the empty symbol. In what follows we assume that there is no isolated vertex in B.

When a graph is convex, the maximum matching problem is considerably easier to solve. In fact Glover proved that the following simple procedure yields a maximum cardinality matching (we assume that both A and B be given as sequences of integers from 1 to $|A|$ and $|B|$ respectively; MATCH[i] denotes the element of B matched to $i \in A$):

Algorithm 0

```
1  begin  for $i := 1$ to $|A|$ do
2            begin $U := \{k:(i, k) \in E$ and $k$ has not been deleted from $B\}$
3               if $U \neq \emptyset$ then (* find $j \in U$ to be matched to $i$ *)
4                  begin $j :=$ element in $U$ with minimum value of END
5                     MATCH[$i$] := $j$
6                     delete $j$ from $B$
7                  end
8               else MATCH[$i$] := $\Lambda$ (* $i$ unmatched *)
9            end
10 end
```

In words, element i of A is matched to an available element j of B whose corresponding interval ends the closest to i. The most time-consuming task of this algorithm is the formation of the set U and the associated determination of an element $j \in U$ with the smallest value of END$[j]$: for any given $i \in A$, it involves scanning all the elements of B connected to i. Thus the running time of this task is clearly $O(|E|)$, as pointed out in [9].

In this paper we shall describe a considerably more efficient algorithm and investigate both specializations and generalizations of the original matching problem. Specifically, after considering (Sect. 2) the maximum matching problem in a convex bipartite graph, we shall analyze the further simplifications which are possible when the graph is doubly convex (Sect. 3), and the optimal time determination of the maximum set of independent vertices associated with a given maximum matching (Sect. 4). Finally (Sect. 5), we succinctly describe two generalizations of the convex matching problem and an extension of the techniques to weighted matching, which directly applies to the solution of a scheduling problem.

2. Maximum Matching in Convex Bipartite Graphs

Let $G=(A, B, E)$ be a bipartite graph convex on A, with $|A|=m$ and $|B|=n$. As before, $A=\{1,2,\ldots,m\}$ and $B=\{1,2,\ldots,n\}$. For $b \in B$, $A(b) \subseteq A$ denotes the set $\{a:(a,b) \in E\}$; similarly, for $a \in A$, $B(a) \subseteq B$ denotes the set $\{b:(a,b) \in E\}$. Again, we assume that A is ordered so that, for each $b \in B$, $A(b)$ is the interval [BEG$[b]$, END$[b]$]. Notice that if the set A is not initially ordered so that the property of convexity is manifest, the bipartite graph G can be tested for possession of this property – and, if so, rearranged – in time $O(|E|+m+n)$ by means of the Booth-Lueker algorithm [2]. Clearly, if the graph is not given by the arrays BEG, END and those arrays have to be computed first, then no improvement is possible over Glover's $O(|E|)$ algorithm.

We begin by giving a generalization (and simpler proof) of Glover's rule.

Lemma 1. *If $(a,b) \in E$ and $A(b) \subseteq A(c)$, for all $c \in B(a)$, then there is a maximum matching containing (a,b).*

Proof. Suppose M is a maximum matching not containing (a,b). If a is unmatched then we may replace the edge of the matching incident to b with (a,b), similarly if b is unmatched. Suppose therefore that (a,c), $(d,b) \in M$ for some $c \in B$, $d \in A$. Since $d \in A(b) \subseteq A(c)$, it follows that $(d,c) \in E$, and we may replace (a,c), (d,b) by (a,b), (d,c) (see Fig. 1). □

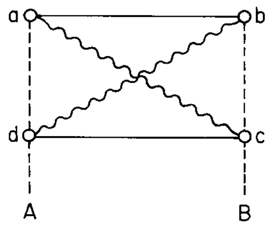

Fig. 1. To the proof of Lemma 1. Wiggly edges belong to the matching

In order to prove that Algorithm 0 correctly finds a maximum matching, let us denote by G_i the graph obtained from G by deleting $1, \ldots, i-1$ from A and MATCH[1], ..., MATCH[$i-1$] from B, together with the edges incident to all these vertices. Let M_i be the set of edges matched by Algorithm 0 to vertices $1, \ldots, i$ in A (we put $M_0 = \emptyset$), and let $A_i(b)$ and $B_i(a)$ be defined for G_i in the same way as $A(b)$ and $B(a)$ were defined for G. We say that M_i can be *extended* to a maximum matching of G if there is a maximum matching M of G containing M_i; this means that M is the union of M_i and of a maximum matching of G_{i+1}.

Assume inductively that $a \leqq m$ and that M_{a-1} can be extended to a maximum matching of G. (This is trivially true for $a=1$, since M_0 is empty and G_0 coincides with G.) We shall prove that M_a can also be extended to a maximum matching of G. This is obviously true if $B_a(a) = \emptyset$, so assume that $B_a(a) \neq \emptyset$, whence Algorithm 0 chooses MATCH[a]$= b \neq \Lambda$. It is then sufficient to show that there is a maximum matching of G_a containing (a, b). But this is immediate, since for any c in $B_a(a)$ we have $A_a(c) = [a, \text{END}[c]]$; by line 4 of Algorithm 0, we have $\text{END}[b] \leqq \text{END}[c]$ for any $c \neq b$ in $B_a(a)$, whence $A_a(b) \subseteq A_a(c)$, and, by Lemma 1, the claim is established.

The efficiency of Algorithm 0 can be improved substantially by some additional preprocessing and the use of appropriate data structures. Indeed, the set U of unmatched vertices of B connected to a currently inspected vertex $i \in A$ can be stored on a priority queue, and if the elements of B are relabelled so that $\text{END}[1] \leqq \ldots \leqq \text{END}[n]$, then the least element of the queue minimizes the value of END, as required by Glover's rule. Note that since the elements in the priority queue are integers in the range $[1, n]$, we may employ the implementation of [3, 4], which allows each of the standard queue operations to be performed in time $O(\log \log n)$ and uses space $O(n)$. The content of the priority queue can be updated efficiently as i is increased by one, and it should be clear the whole algorithm can be implemented in time $O(m + n \log \log n)$ (note that all sorting operations which are needed involve n integers in the range $[1, n]$, so they can be done in $O(m+n)$ time by standard bucket sorting, see e.g. [1]).

However, as noted by Hopcroft [personal communication], a better algorithm is possible if we use an idea closely related to that employed in an efficient solution of the off-line MIN problem (see [1], p. 139). We shall now describe such an algorithm in some detail.

Assume $\text{END}[1] \leqq \ldots \leqq \text{END}[n]$, and consider the sets

$$A_i = \{j \in B \colon \text{BEG}[j] = i\}, \qquad 1 \leqq i \leqq m.$$

The integer i will be called the *name* of A_i. The sets A_i are stored as a doubly linked collection of trees, with links PRED[i], SUCC[i] associated with each A_i, the trees being represented as in the UNION-FIND algorithm [12]. (Recall that the UNION-FIND algorithm processes a collection of pairwise disjoint sets and supports two operations: UNION, which takes two sets in the collection and replaces them by their union, and FIND, which takes an element and determines the set containing this element.) The main loop of the algorithm scans vertices in B (rather than in A, as in Algorithm 0) in order of nondecreasing value of END. For any such $j \in B$ the algorithm finds a vertex $i \in A$ to be matched to j.

This vertex i is chosen to be BEG[j], i.e., the name of the set containing j, or FIND(j) in the terminology of the UNION-FIND algorithm (here BEG[j] refers to the current graph, with all previously matched vertices deleted). After i is matched to j, the graph is updated by deleting i, i.e., increasing BEG[j] to the next available value, for all $j \in B$ for which BEG[j] was equal to i. This means we take a union of A_i and $A_{\mathrm{SUCC}[i]}$ and give this set the name SUCC[i] (that is, we execute (UNION(i, SUCC[i], SUCC[i]) in the terminology of the UNION-FIND algorithm), and then we update the links to reflect the fact that A_i has been deleted. This procedure may cause some $j \in B$ to become isolated, but this can easily be found out by checking whether FIND(j) $\leqq$ END[j]. The whole algorithm is summarized below.

Algorithm 1. *(Finding maximum matching in convex bipartite graph)*

```
Input:  BEG[1:n], END[1:n]
        END[1] ≦ ... ≦ END[n]
Output: MATCH[1:n] (MATCH[j] is the vertex in A matched to j∈B)
 1  begin
 2      for i := 1 to m do
 3          begin A_i := ∅
 4                PRED[i] := i−1
 5                SUCC[i] := i+1
 6          end
 7      SUCC[0] := 1, PRED[m+1] := m
 8      for j := 1 to n do A_BEG[j] := A_BEG[j] ∪ {j}
 9      for j := 1 to n do (* find vertex to be matched to j *)
10          begin i := FIND(j)
11              if i ≦ END[j] then MATCH[j] := i
12                            else MATCH[j] := Λ (* i unmatched *)
13              UNION(i, SUCC[i], SUCC[i])
14              SUCC[PRED[i]] := SUCC[i]
15              PRED[SUCC[i]] := PRED[i]
16          end
17  end
```

It can easily be proven that Algorithm 1 constructs exactly the same matching as Algorithm 0 does (if we assume the same labelling of vertices with END[1] $\leqq \ldots \leqq$ END[n] in both cases, and if we choose the vertex j in line 4 of Algorithm 0 to be the smallest element in U). Indeed, suppose inductively that for all $j < j_0$ both algorithms either match j to the same vertex in A or both leave j unmatched, and assume the main loop of Algorithm 1 reaches $j = j_0$. Let us denote $i_0 = \mathrm{FIND}(j_0)$. If $i_0 > \mathrm{END}[j_0]$ then it is impossible to match j_0 without violating the partial matching which was assumed to be chosen by both algorithms, and consequently both algorithms will leave j_0 unmatched. Now suppose $i_0 \leqq \mathrm{END}[j_0]$ so that Algorithm 1 matches j_0 to i_0. First notice that our inductive assumption and the fact that $\mathrm{FIND}(j_0) = i_0$ when Algorithm 1 reaches $j = j_0$ imply that all $i < i_0$ connected to j_0 are matched by both algorithms to the same vertices $j < j_0$. This means that $j_0 \in U$ when Algorithm 0 reaches $i = i_0$, and so it is easily seen that Algorithm 0 will also match i_0 to j_0. To estimate the complexity of Algorithm 1, notice that the loops in lines 2 and 8 take $O(m+n)$ steps, and the main loop in line 9 performs n FIND operations interleaved by n UNION operations, which requires $O(nA(n))$ steps, $A(n)$ being an extremely slowly growing function related to a functional inverse of Ackermann's function (see Tarjan [12]). So we conclude that the running time of Algorithm 1 is $O(m+nA(n))$.

3. Maximum Matching in Doubly Convex Bipartite Graphs

As noted by Glover, the maximum matching problem becomes even simpler when the bipartite graph G is *doubly convex*, i.e., orderings of both A and B exist such that every $A(b)$ is an interval of A and every $B(a)$ is an interval of B.

As before, we assume that G be given as a bipartite graph convex on A, that is, as a set $\{\langle \mathrm{BEG}[b], \mathrm{END}[b] \rangle : b \in B\}$ representing intervals of A. In the rest of this section we shall also assume that the convex bipartite graph G under consideration is connected. In fact, it is very easy to find connected components of a convex bipartite graph. It is sufficient to scan vertices $i \in A$ in increasing order and to count the number of beginnings and the number of endings of intervals found up to vertex i. Each time these two counts coincide, a new connected component is found. With the elements of B labelled so that $\mathrm{END}[1] \leqq \ldots \leqq \mathrm{END}[n]$, and with an additional array storing the same elements sorted by nondecreasing value of BEG (it can be formed in linear time by bucket sorting), the determination of connected components can be done in $O(m+n)$ time.

A preliminary task is to test whether the set B can be reordered so that for each $a \in A$ the set $B(a)$ be an interval of B. Pictorially, we may display G by means of a set of segments (Fig. 2a): specifically, in the plane (x, y), we let the segment

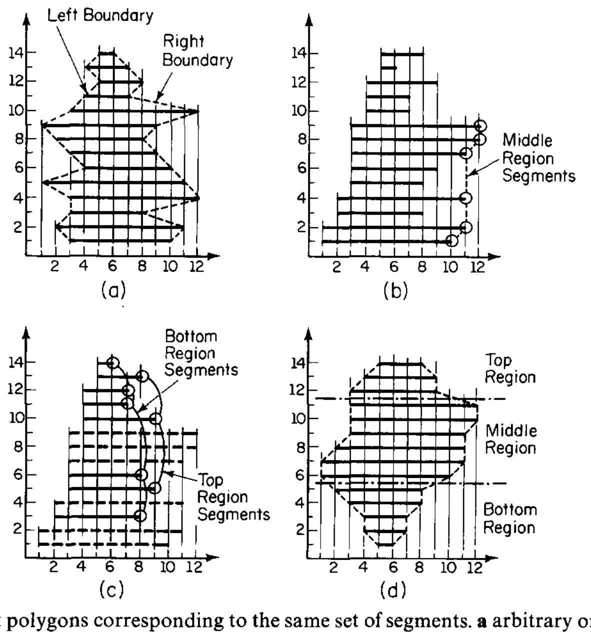

Fig. 2a-d. Different polygons corresponding to the same set of segments. **a** arbitrary order; **b, c** ordered by nonincreasing BEG; **d** ordered to exhibit double convexity

$\{(x, y): y=b, \text{BEG}[b] \leqq x \leqq \text{END}[b]\}$ represent the interval $A(b)$ (in the sequel this will be briefly referred to as segment b). If we next join the extremes of adjacent segments, i.e., introduce in this diagram the edges joining point $(\text{BEG}[i], i)$ with point $(\text{BEG}[i+1], i+1)$ and those joining $(\text{END}[i], i)$ with $(\text{END}[i+1], i+1)$ for $i=1, 2, \ldots, n-1$, the set of segments is enveloped by two polygonal lines called the *left* and *right boundaries*, which together with the first and last segments of the given set form a simple polygon. In this representation, G is convex on B if (after suitably rearranging the segments) the intercept of any vertical line with this polygon consists of a single segment: in other words, G is convex on B if and only if the segments can be rearranged so that both boundaries are *bitonic*, as shown in Fig. 2d (that is, in the resulting relabelling of elements of B, for some $1 \leqq r_1 \leqq n$, $\text{BEG}[1] \geqq \ldots \geqq \text{BEG}[r_1]$ and $\text{BEG}[r_1] \leqq \ldots \leqq \text{BEG}[n]$; similarly, for some $1 \leqq r_2 \leqq n$, $\text{END}[1] \leqq \ldots \leqq \text{END}[r_2]$ and $\text{END}[r_2] \geqq \ldots \geqq \text{END}[n]$). We shall now describe a linear time - hence optimal - algorithm which tests G for double convexity and, if this property holds, produces the desired ordering of B.

Referring to Fig. 2d, it is easy to see that the polygon displaying the double convexity of an arbitrary G consists - up to the reversal of the ordering of B - of three regions (not all simultaneously empty): a *middle* region, where both left and right boundaries are nondecreasing (i.e., both $\text{BEG}[j]$ and $\text{END}[j]$ are nondecreasing with increasing j, assuming that the labelling of elements of B coincides with the bottom to top ordering of segments in the given geometric representation); a *top* region where the left and right boundaries are nondecreasing and nonincreasing, respectively; a *bottom* region where the left and right boundaries are nonincreasing and nondecreasing, respectively. Moreover, all segments of the top region are nested, starting with the topmost segment of the middle region, and similarly, all segments of the bottom region are nested, starting with the bottommost segment of the middle region. It is easy to see that our description need not define the three regions uniquely.

Suppose that we initially index the elements of B so that the pairs $\langle \text{BEG}[j], \text{END}[j] \rangle$, $j=1, \ldots, n$ are in lexicographic ascending order; this can be done by bucket-sorting these elements on the parameter BEG, and then (stably) bucket-sorting the resulting sequence on the parameter END, all in time $O(m+n)$. Once this ordering of segments $\{A(b): b \in B\}$ is available (see Fig. 2b), we shall first extract from it the subsequence of segments to be assigned to the middle region. To complete the test, we must verify whether the remaining segments can be successfully assigned to either the top or bottom regions. Since for segments in these regions, the orderings BEG and END are contragradient, we must preliminarily alter the order of the segments not assigned to the middle region, so that for any two such consecutive segments j and $j+1$, $(\text{BEG}[j] = \text{BEG}[j+1]) \Rightarrow (\text{END}[j] \geqq \text{END}[j+1])$: this can be obviously done in linear time by a straightforward use of a stack (Fig. 2c). Next, we must test whether the resulting sequence can be partitioned into *two* subsequences, for each of which the parameter END is nonincreasing: if this is feasible, then the two subsequences of segments will respectively form the top and bottom regions. More exactly, we should do the partitioning in such a way, that the resulting subsequences of segments be nested as previously explained. We guarantee this by assigning the extremal segments of the middle region to the sequence to be partitioned.

The whole task is performed by the following algorithm, which computes for each segment j a parameter $Y[j]$ denoting its order in the final arrangement. This algorithm also makes use of a special subroutine, which – if at all possible – partitions in linear time a sequence of integers into two nonincreasing subsequences; for example (4, 6, 3, 5, 4) is partitioned into (4, 3) and (6, 5, 4). This simple subroutine is described formally in an appendix. Its additional feature, which is important for the correctness of our algorithm, is that the first term of the sequence is assigned to the *first* subsequence.

Algorithm 2. *(Testing for double convexity of a connected convex bipartite graph)*

```
Input:  $\mathrm{BEG}[1{:}n]$, $\mathrm{END}[1{:}n]$
        The pairs $\langle \mathrm{BEG}[j], \mathrm{END}[j]\rangle$, $j=1,\dots,n$ are in lexicographic
        increasing ordering
Output: $Y[1{:}n]$
        Vertices $j\in B$ relabelled so that for $1\leqq j<n$
        $\mathrm{BEG}[j]<\mathrm{BEG}[j+1]$, or $\mathrm{BEG}[j]=\mathrm{BEG}[j+1]$ and $\mathrm{END}[j]\geqq\mathrm{END}[j+1]$
begin (* find last segment $jm$ of middle region *)
   $jm:=1$
   for $j:=2$ to $n$ do
      if $\mathrm{END}[j]\geqq\mathrm{END}[jm]$ then $jm:=j$
   (* extract segments not in internal part of middle region *)
   $e:=\mathrm{END}[1]$, $l:=0$
   for $j:=1$ to $n$ do
      if $(\mathrm{END}[j]\geqq e)$ and $(j\neq 1)$ and $(j\neq jm)$ then $e:=\mathrm{END}[j]$
      else begin $l:=l+1$
                 $S[l]:=j$
           end
      relabel the elements of $B$ so that for $1\leqq j<n$
         $(\mathrm{BEG}[j]=\mathrm{BEG}[j+1])\Rightarrow(\mathrm{END}[j]\geqq\mathrm{END}[j+1])$
      reorder $S[1{:}l]$ so that for $1\leqq p<l$
         $(\mathrm{BEG}[S[p]]=\mathrm{BEG}[S[p+1]])\Rightarrow(\mathrm{END}[S[p]]\geqq\mathrm{END}[S[p+1]])$
      partition $S[1{:}l]$ into two subsequences $\mathrm{SUB1}[1{:}l1]$ and
         $\mathrm{SUB2}[1{:}l2]$, such that $\mathrm{END}[\mathrm{SUB1}[1]]\geqq\dots\geqq\mathrm{END}[\mathrm{SUB1}[l1]]$
         and $\mathrm{END}[\mathrm{SUB2}[1]]\geqq\dots\geqq\mathrm{END}[\mathrm{SUB2}[l2]]$
      $k1:=k2:=k3:=1$
      for $j:=1$ to $n$ do (* determine $Y[j]$ *)
         if $\mathrm{SUB1}[k1]=j$ then (* $j$ belongs to bottom region *)
            begin $Y[j]:=l1-k1+1$
                  $k1:=k1+1$
            end
         else if $\mathrm{SUB2}[k2]=j$ then (* $j$ belongs to top region *)
               begin $Y[j]:=n-l2+k2$
                     $k2:=k2+1$
               end
            else (* $j$ belongs to middle region *)
               begin $Y[j]:=l2+k3$
                     $k3:=k3+1$
               end
end
```

It is straightforward to conclude that Algorithm 2 runs in time $O(n)$.

We can now describe the maximum matching algorithm, which makes use of a DEQUE (doubly-ended-queue) as an auxiliary data structure; as is well-known, DEQUE has two distinguished elements, *top* and *bottom*, and the following repertoire of instructions: INSERTTOP, DELETETOP, INSERTBOTTOM, and DELETEBOTTOM.

Algorithm 3. *(Finding maximum matching in doubly convex bipartite graph)*

```
Input:  BEG[1:n], END[1:n], Y[1:n]
        BEG[j] < BEG[j+1], or BEG[j] = BEG[j+1] and END[j] ≧ END[j+1]
        for 1 ≦ j < n
Output: MATCH[1:m] (MATCH[i] is a vertex in B matched to i ∈ A)
begin DEQUE := ∅, j := 1
   for i := 1 to m do
      begin (* find element in B to be matched to i ∈ A *)
         while (BEG[j] = i) and (j ≦ n) do
            begin (* insert j into DEQUE *)
               if (DEQUE = ∅) or (Y[j] > Y[top]) then INSERTTOP(j)
               else INSERTBOTTOM(j)
               j := j + 1
            end
         if (DEQUE = ∅) then MATCH[i] := Λ (* i unmatched *)
         else if END[top] < END[bottom] then
               begin MATCH[i] := top
                     DELETETOP
               end
            else begin MATCH[i] := bottom
                     DELETEBOTTOM
             end
         while (DEQUE ≠ ∅) and (END[top] = i) do DELETETOP
         while (DEQUE ≠ ∅) and (END[bottom] = i) do DELETEBOTTOM
      end
end
```

Notice that this algorithm is a specialization of Algorithm 0. We use the DEQUE to store the set U of unmatched vertices connected to a currently inspected vertex $i \in A$. By keeping elements on the DEQUE ordered so that $Y[j]$ increases from bottom to top, we guarantee that the element minimizing the value of END is always either *bottom* or *top*. Each element of B is inserted into and deleted from the DEQUE exactly once, and each of the standard DEQUE operations can be executed in constant time; it follows that the entire matching can be computed in time $O(m+n)$.

4. Finding a Maximum Independent Set of Vertices in a Convex Bipartite Graph

Closely related to the maximum matching problem in bipartite graphs is the determination of a maximum independent set (of vertices), that is, of a maximum cardinality set of vertices of a bipartite graph G such that no two of them are connected. It is well-known (see, e.g. [6, 11]) that a maximum independent set can be derived from a maximum matching M by standard alternating path techniques as follows (see Fig. 3): (i) direct every edge $e \in M$ from A to B, and any $e \in E - M$ from B to A; (ii) letting B_0 denote the set of unmatched vertices in B, find the sets $A_1 \subseteq A$ and $B_1 (B_0 \subseteq B_1 \subseteq B)$ of vertices reachable from B_0; (iii) construct the maximum independent set as $I = B_1 \cup (A - A_1)$. Therefore the entire problem is reduced to finding all the vertices of G which are reachable from B_0. A most interesting fact we shall now show is that, when G is convex,

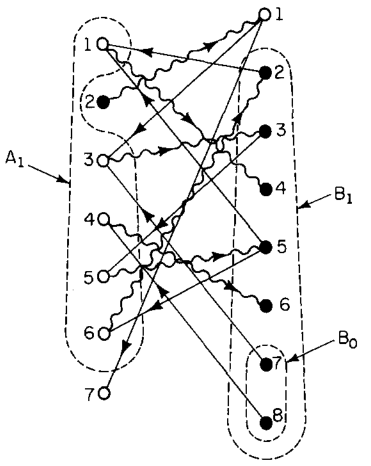

Fig. 3. Illustration of the derivation of a maximum independent set from a maximum matching (wiggly edges are in the matching M): vertices in the independent set shown as ●

this reachable set can be obtained in time $O(n+m)$, i.e., linear in the number of *vertices*, and not just linear in the number of *vertices and edges*, as in [6], [11]. Consequently, the determination of a maximum independent set runs in total time $O(m+nA(n))$, or $O(m+n)$ if G is doubly convex, the computation of the maximum matching being the dominant task (note that, once A_1 and B_1 are known, I is obtainable in time $O(n+m)$).

As usual, the graph G is described by the two arrays BEG[1:n] and END[1:n]; MATCH[1:n] gives for each $i\in A$ either Λ or the element of B matched to it. We assume that the elements of B be ordered so that BEG[i]$\leqq$BEG[$i+1$], $1\leqq i<n$. Due to the property of convexity, for each $b\in B_0$ the set $A(b)$, i.e. the set of vertices reachable by a single edge from b, forms an interval of A; from any matched vertex a in this interval we reach a single vertex MATCH[a]$\in B$, which in turn reaches another interval A(MATCH[a]) of A. Notice that $A(b)$ and A(MATCH[a]) necessarily overlap, so by the convexity of G their union is a single interval. Therefore, initially we place in a queue all the elements of B_0 in increasing order, and starting with the smallest one j_1, we determine a single *extended interval* $A^*(j_1)$ of A (note that $A^*(j_1)\supseteq A(j_1)$), which is the set of all elements of A which are reachable from j_1 ($A^*(j_1)$ could be informally viewed as the "closure" of $A(j_1)$). This extended interval is constructed by scanning $A(j_1)$ in *decreasing* order starting from END[j_1] and currently updating the extremes of the reached interval; once the scanning reaches the lower extreme without further downward extension of the interval, then if the interval has been extended upward beyond END[j_1], scanning is resumed in ascending order starting from END[j_1]+1 until the same terminating condition occurs, and this process is repeated until no further extension - either downward or upward - is possible. At this point the construction of interval $A^*(j_1)$ has been completed. We then extract the next element j_2 from the queue and begin the construction of $A^*(j_2)$. Note that if $A^*(j_1)$ and $A(j_2)$ are disjoint (Fig. 4a), BEG[j_2] must

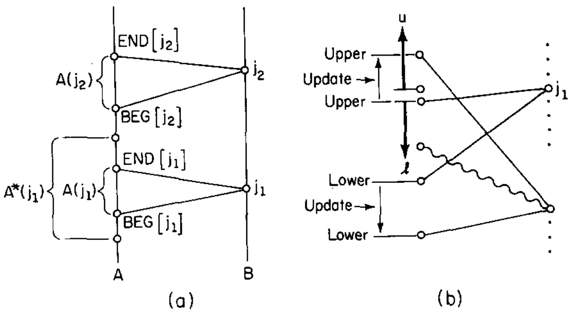

Fig. 4. a Illustration of the case where $A^*(j_1)$ and $A(j_2)$ are disjoint. **b** Explanation of the meaning of variables *lower*, *upper*, *l* and *u*

be larger than the upper extreme of $A^*(j_1)$, because by hypothesis, $\text{BEG}[j_1] \leqq \text{BEG}[j_2]$. It follows that only downward extensions of $A(j_2)$ may meet previously scanned elements of A. To avoid any time-consuming unnecessary repeated scanning, we must ensure than any previously scanned interval be skipped in subsequent processing, so that each element of A be scanned at most once. This objective is achieved by means of a stack: as soon as the construction of $A^*(j)$, for some $j \in B_0$, is completed, its lower and upper extremes are inserted into the stack, whose content - at a generic instant - is a sequence $-1, i_1, e_1, i_2, e_2, \ldots, i_k, e_k$, such that, for $1 \leqq p < k$, $e_p + 1 < i_{p+1}$, $[i_p, e_p]$ is an interval of A, and $\bigcup_{p=1}^{k} [i_p, e_p]$ is the set of all scanned elements of A. The reachability algorithm uses as auxiliary data structures a QUEUE, containing the elements of B_0 ordered according to nondecreasing value of BEG, and a STACK, for storing the sequence of scanned intervals, as already noted. The intuitive significance of the program variables *lower*, *upper*, *l*, and *u* is as follows (see Fig. 4b): lower and upper denote respectively the current boundaries of the extended interval being constructed; *l* and *u* are pointers used in scanning, running downward and upward respectively.

Algorithm 4. *(Finding the set of vertices in A reachable by alternating paths from the set of unmatched vertices in B in a convex bipartite graph)*

```
Input:  BEG[1:n], END[1:n], MATCH[1:m]
        QUEUE containing the unmatched vertices $b \in B$ in increasing order
        $\text{BEG}[1] \leqq \ldots \leqq \text{BEG}[n]$
Output: The set $\bigcup_{p=1}^{k} [i_p, e_p] \subseteq A$ of vertices reachable from unmatched
        vertices $b \in B$, represented by a sequence $-1, i_1, e_1, i_2, e_2, \ldots, i_k, e_k$
        stored on STACK
1  begin
2     STACK $\Leftarrow -1$
3     while QUEUE $\neq \emptyset$ do (* find vertices reachable from first (QUEUE) *)
4        begin $j \Leftarrow$ QUEUE
5           if END[j] > top(STACK) then (* new vertices to be scanned *)
6              begin $l := \text{END}[j]+1$, $lower := \text{BEG}[j]$, $u := upper := \text{END}[j]$
7                 repeat (* extend interval of vertices reached from $j$ *)
```

```
 8                 while $l > lower$ do (* scan downward *)
 9                    begin $l := l - 1$
10                       if MATCH[$l$] $\neq \Lambda$ then (* $l$ is matched *)
11                          begin $lower := \min(lower$, BEG[MATCH[$l$]])
12                               $upper := \max(upper$, END[MATCH[$l$]])
13                          end
14                       if $l <$ top(STACK)+1 then (* skip interval *)
15                          begin $l \Leftarrow$ STACK
16                               $l \Leftarrow$ STACK
17                               $lower := \min(lower, l)$
18                          end
19                    end
20                 while $u < upper$ do (* scan upward *)
21                    begin $u := u + 1$
22                       if MATCH[$u$] $\neq \Lambda$ then (* $u$ is matched *)
23                          begin $lower := \min(lower$, BEG[MATCH[$u$]])
24                               $upper := \max(upper$, END[MATCH[$u$]])
25                          end
26                    end
27              until ($l = lower$) and ($u = upper$) (* extended interval completed *)
28              STACK $\Leftarrow lower$
29              STACK $\Leftarrow upper$
30           end
31     end
32  end
```

To analyze the performance of Algorithm 4, we note that each element of A is scanned at most once (either by loop 8 or by loop 20); the extremes of extended intervals are pushed into (lines 28 and 29) and popped from STACK (lines 15 and 16) at most once, thereby allowing the conclusion that the algorithm runs in time $O(m+n)$.

5. Generalizations and Related Problems

In this section we shall briefly describe two interesting generalizations of the notion of a convex bipartite graph to which Glover's rule, and hence the efficient algorithms previously described are applicable, and an extension of the techniques to a weighted matching problem, which models a significant scheduling application.

5.1. Simple Chessboards: A Generalization of Doubly Convex Bipartite Graphs

Algorithm 3 can be applied to a class of convex bipartite graphs more general than that of doubly convex graphs. In order to describe this class we shall need some definitions. By a *chessboard* we shall mean any finite collection of unit squares with integer coordinates on a plane. Any such unit square will be denoted by coordinates $\langle x, y\rangle$ of its left lower corner. A chessboard is *simple* if for any of its squares $\langle x, y_1\rangle$, $\langle x, y_2\rangle$, where $y_1 \leqq y_2$, it contains all squares $\langle x, y\rangle$, $y_1 \leqq y \leqq y_2$ (see Fig. 5). *Rows* and *columns* of a chessboard are defined in the natural way

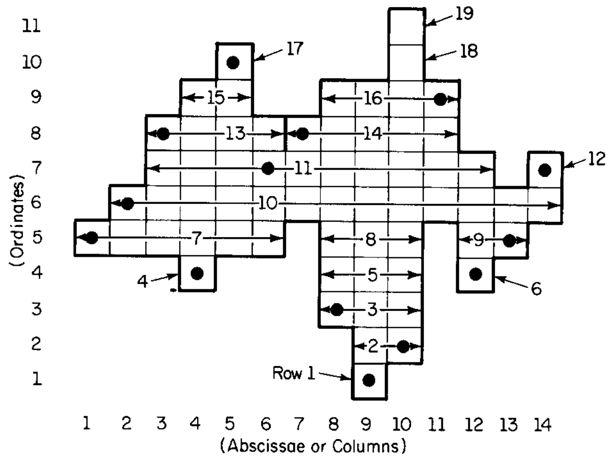

Fig. 5. A simple chessboard with numberings for rows and columns. Also shown is a maximum set of nonattacking rooks found by Algorithm 3

as maximal horizontal and vertical sequences of adjacent squares, respectively. We may allow a simple chessboard to be cut vertically in some places to make some squares nonadjacent (such as $\langle 6, 8\rangle$ and $\langle 7, 8\rangle$ in Fig. 5), provided the line along which we cut touches the boundary of the chessboard.

Let A and B be the set of columns and rows of a simple chessboard, respectively. Each column corresponds to a unique integer abscissa of the plane, while more than one row may correspond to a given ordinate. (For example, in Fig. 5 there are 19 chessboard rows on 11 ordinates.) Each chessboard column is numbered by its abscissa, while chessboard rows are numbered according to the lexicographic ordering of $\langle y, x_{\min}\rangle$,where y is the ordinate of a given row and $x_{\min}$ is its smallest abscissa. We now define a bipartite graph $G=(A, B, E)$, where $(a, b)\in E$ if and only if column a and row b intersect (i.e., have a square in common). This graph is convex on A (but not necessarily doubly convex). It is easily seen that any matching in G corresponds to a set of nonattacking rooks on this chessboard. The maximum cardinality set of nonattacking rooks on this chessboard is found by Algorithm 3 in time linear in the number of rows and columns. The reason why Algorithm 3 works correctly is that, similarly to the doubly convex case,

(a) the sequence of right ends of rows intersected by any column of a simple chessboard is bitonic, whence the sequence of the values of END for vertices $j\in B$ (rows of the chessboard) stored on the DEQUE is also bitonic, and we may find a vertex with the minimal value of END either at the top or at the bottom of the DEQUE;

(b) the sequence of left ends of rows intersected by any column is also bitonic, whence the element to be inserted into or deleted from the DEQUE is always either at the top or at the bottom of DEQUE.

5.2. Bipartite Graphs Convex on a Tree-ordered Set

Algorithm 1 can be extended to a more general situation, where the sets $A(b)$, $b \in B$ are paths in a directed tree. More precisely, a bipartite graph $G=(A, B, E)$ is said to be *convex on a tree-ordered set* if there is a directed tree with vertex set A (for concreteness we shall assume the tree is directed toward the root) such that for every $b \in B$, $A(b)$ is the set of vertices of a directed path in this tree. (Families of sets which can be represented as paths in a directed tree are of some importance in file organization [10].) The convex case is easily seen to correspond to a tree degenerating into a single path. Assume that a directed tree with vertex set A is represented by an array $S[1:m]$ which gives the successor $S[a]$ of any vertex $a \in A$ ($S[a]=\Lambda$ if a is the root). Similarly as in the convex case, let $A(b)$ be represented by the pair $\langle \mathrm{BEG}[b], \mathrm{END}[b] \rangle$, meaning that $A(b)$ is the set of vertices of the path in the tree, beginning at BEG$[b]$ and ending at END$[b]$. From the array S we can easily produce, in $O(m)$ time, a *topological ordering* of A, i.e., a linear ordering of the elements of A, in which the distance to the root, or the rank of a vertex is nonincreasing.

We may assume that the vertices in A are labelled according to the topological ordering, and that the vertices in B are labelled so that $\mathrm{END}[1] \leqq \ldots \leqq \mathrm{END}[n]$. As in the convex case two approaches to the determination of a maximum matching are possible. In the first one we scan vertices in A in increasing order, and – as in Algorithm 0 – we match a current vertex $i \in A$ to the smallest unmatched $j \in B$ connected to i (i.e., due to our labelling of vertices, a $j \in B$ for which END$[j]$ is smallest). The correctness of such an extension of Glover's rule can easily be proved by a straightforward application of Lemma 1. In the second approach, we scan vertices in B in increasing order, and – as in Algorithm 1 – we match a current vertex j to the smallest available $i \in A$. As in the convex case, the second approach turns out to be more efficient, in fact it can be implemented in $O(m+nA(n))$ time. A modification that should be incorporated into Algorithm 1 is to link the sets A_i into a tree-like structure corresponding to our directed tree. More exactly, A_i is linked to $A_{S[i]}$. Now that a vertex i can have an arbitrary number of predecessors, we cannot afford storing – and updating – backward links, and consequently we are not able to efficiently delete a matched vertex $i \in A$ from our tree structure. To avoid this difficulty, we mark a matched vertex "empty" instead of deleting it. This may cause that we will have to traverse a "long" path of empty vertices in order to reach the first nonempty vertex i^* which follows in our tree structure a nonempty vertex i (we would then perform UNION(i, i^*, i^*)). This problem, however, can be solved efficiently by the same techniques as those used in the UNION-FIND algorithm. We treat our tree structure of current sets A_i as a collection of UNION-FIND trees, where find(i) is the first nonempty ancestor of i (a vertex $i' \in A$ is an ancestor of i if it can be reached by a directed path starting at i in our tree structure; the length of the path may be 0, so that i is an ancestor of itself). Initially, each such tree consists of just a root. Notice that two UNION-FIND structures are involved here: one connected with the sets A_i which form a partition of B, and the second one connected with a partition of A (to avoid confusion the operations union, find referring to the second structure will be written in lower case letters). Now suppose that we

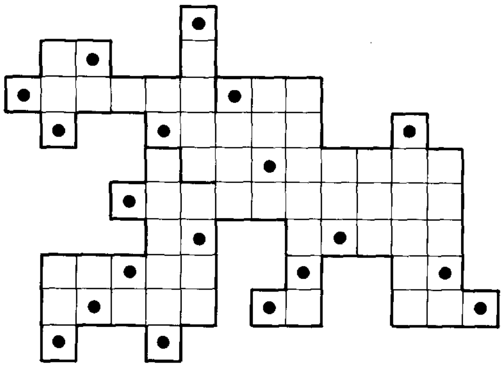
Fig. 6. A chessboard and a maximum set of nonattacking rooks found by a modification of Algorithm 1

found a vertex $i = \mathrm{FIND}(j)$ to be matched to j. The vertex i becomes "empty", and to update our UNION-FIND structures we first determine $i^* = \mathrm{find}(S[i])$, and then perform $\mathrm{union}(i, i^*, i^*)$ and $\mathrm{UNION}(i, i^*, i^*)$. Details of the proof of correctness and of the $O(m+nA(n))$ bound are left to the reader.

Our algorithm can be used to find a maximum set of nonattacking rooks on a chessboard satisfying the following condition: any two squares $\langle x, y_1\rangle\langle x, y_2\rangle$ can be joined by a sequence $\langle x, y_1\rangle = \langle x^{(1)}, y^{(1)}\rangle, \langle x^{(2)}, y^{(2)}\rangle, \ldots, \langle x^{(k)}, y^{(k)}\rangle = \langle x, y_2\rangle$ of adjacent (i.e., having an edge in common) squares with $x^{(i)} \geqq x$, $1 \leqq i \leqq k$. In words, the chessboard does not branch as we go from left to right (see Fig. 6). The tree-like ordering of the set A of columns of such a chessboard is defined so that a column containing square $\langle x+1, y\rangle$ is the successor of column containing square $\langle x, y\rangle$.

5.3. Gale-optimal Matchings and One-processor Scheduling of Independent Tasks

It is clear that Algorithm 4 can be modified so that it finds an alternating path in a convex bipartite graph - if there is one - in linear time. Indeed, it is sufficient to store, together with each vertex reached by the algorithm, a pointer to the vertex from which it was reached. Using such a modified algorithm as a subroutine in the standard method of finding a maximum matching, based on repeatedly augmenting a matching along an alternating path, we can obtain an algorithm of complexity $O(n(m+n))$. (Recall that augmenting a matching M along an alternating path $(a_0, a_1), (a_1, a_2), \ldots, (a_{2k}, a_{2k+1})$, where a_0 and a_{2k+1} are unmatched, $(a_{2i-1}, a_{2i}) \in M$ and $(a_{2i}, a_{2i+1}) \notin M$ $(0 \leqq i \leqq k)$, consists in replacing in M the k edges (a_{2i-1}, a_{2i}) by the $(k+1)$ edges (a_{2i}, a_{2i+1}), see e.g. [9].) Of course, such an algorithm would be less efficient than the $O(m+nA(n))$ Algorithm 1. However, there is a situation when the standard alternating path algorithm is of interest.

Suppose that there is a *weight* $w(b) \geqq 0$ associated with every $b \in B$, and that we are looking for a matching which maximizes the sum of weight of matched vertices in B. Since assignable subsets of B - i.e., subsets that can be covered by a matching - form a matroid, it follows that the matching we are looking for can be found by a matroid greedy algorithm (see [9] for the explanation of all notions related to matroids). More exactly, our matching can be obtained as follows: (i) order the vertices in B according to nonincreasing weight, (ii) starting with the empty matching, scan B in this order; for any $b \in B$, augment the current matching along an alternating path starting at b and ending at an unmatched vertex in A, if such a path exists, or leave b unmatched otherwise. Notice that after the augmentation process in step (ii), vertices which were matched remain matched (probably to different vertices), and vertices which were left unmatched before, remain unmatched. It can be proved ([5], see also [9]) that the matching M so obtained is *Gale-optimal*, i.e. optimal in the following strong sense: Let $\{b_1, \dots, b_k\} \subseteq B$, $w(b_1) \geqq \dots \geqq w(b_k)$ be the set of vertices covered by M. Then for any other matching M', the set $\{c_1, \dots, c_l\} \subseteq B$, $w(c_1) \geqq \dots \geqq w(c_l)$ of vertices covered by M' satisfies the condition $l \leqq k$, $w(b_1) \geqq w(c_1), \dots, w(b_l) \geqq w(c_l)$. (Notice that both the greedy algorithm and the notion of Gale-optimality depend only on the ordering of B according to the weights, and not on the actual values of the weights.)

It is obvious that a Gale-optimal matching of a convex bipartite graph can be obtained in $O(n(m+n))$ time by the greedy algorithm, using a modification of Algorithm 4, as explained at the beginning of this subsection.

There is an interesting relationship between Gale-optimal matchings in convex bipartite graphs and the problem of scheduling a set B of n independent (no precedence constrains) tasks on one processor, where each task takes one unit of processing time, there is a *starting time* BEG[j] and *deadline* END[j] for every task j, and a *penalty* $p(j)$ which must be paid if this task is not executed in the time interval [BEG[j], END[j]] (we assume that time is integer-valued). It is easy to see that a schedule minimizing the total penalty can be obtained from a Gale-optimal matching in a convex bipartite graph defined by arrays BEG, END, and with $w(j) = M - p(j)$ $(M > \max_{1 \leqq j \leqq n} p(j))$: the vertex i matched to task $j \in B$ determines the unit interval of time when j is to be executed (see [9], Chapter 7). We conclude that an optimal schedule for this problem can be obtained in $O(n(m+n))$ time (m is the maximal deadline). Of course, if all penalties are equal, i.e., when we simply maximize the number of tasks executed, then the optimal schedule can be obtained in $O(m+nA(n))$ time by Algorithm 1. Notice that the less efficient $O(m+n \log\log n)$ implementation of Algorithm 0 mentioned in Sect. 2 may be of some interest in this context, since it results in an "on-line" scheduling algorithm: tasks can be selected for execution as they arrive, without knowing the tasks which are still to come (of course it is not so in the case of Algorithm 1).

Acknowledgments. The authors are indebted to John Hopcroft for suggesting the possibility of using the off line MIN technique in Algorithm 1 and to a referee for invaluable comments which have greatly contributed to the presentation of this paper. This work was supported in part by the National Science Foundation under grant MCS 78-13642 and in part by the Joint Services Electronics Program under Contracts DAAB-07-72-C-0259 and DAAG-29-78-C-0016.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms. Reading, MA: Addison-Wesley, 1974
2. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J. Comput. System Sci. **13**, 335–379 (1976)
3. Emde Boas, P. van: 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
4. Emde Boas, P. van: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Lett. **6**, 80–82 (1977)
5. Gale, D.: Optimal assignments in an ordered set: an application of matroid theory. J. Combinatorial Theory **4**, 176–180 (1968)
6. Gavril, F.: Testing for equality between maximum matching and minimum node covering. Information Processing Lett. **6**, 199–202 (1977)
7. Glover, F.: Maximum matching in convex bipartite graph. Naval Res. Logist. Quart. **14**, 313–316 (1967)
8. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. **2**, 225–231 (1973)
9. Lawler, E.L.: Combinatorial Optimization: Networks and matroids. New York, NY: Holt, Rinehart and Winston, 1976
10. Lipski, W.: Information storage and retrieval – mathematical foundations II (Combinatorial problems). Theor. Comput. Sci. **3**, 183–211 (1976)
11. Lipski, W., Lodi, E., Luccio, F., Mugnai, C., Pagli, L.: On two dimensional data organization II. Fundamenta Informaticae **2**, 227–243 (1977)
12. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. Assoc. Comput. Mach. **22**, 215–224 (1975)

Received July 17, 1979/March 18, 1981

Appendix

Algorithm A. *(Partitioning a sequence of n integers into two nonincreasing subsequences)*

```
Input:  $S[1:l]$ – the original sequence
Output: $\mathrm{SUB1}[1:l1]$, $\mathrm{SUB2}[1:l2]$ – two nonincreasing subsequences
        into which $S[1:l]$ is partitioned; $S[1]=\mathrm{SUB1}[1]$
 1  begin $l1:=l2:=0$, $\mathrm{SUB1}[0]:=\mathrm{SUB2}[0]:=\infty$
 2    for $i:=1$ to $l$ do
 3      if $S[i]\leq\mathrm{SUB1}[l1]$ then (* add $S[i]$ to first subsequence *)
 4        begin $l1:=l1+1$
 5          $\mathrm{SUB1}[l1]:=S[i]$
 6        end
 7      else if $S[i]\leq\mathrm{SUB2}[l2]$ then (* add $S[i]$ to second subsequence *)
 8          begin $l2:=l2+1$
 9            $\mathrm{SUB2}[l2]:=S[i]$
10          end
11        else stop (* no partitioning possible *)
12  end
```

To prove the correctness of the algorithm, first notice that we always have $\mathrm{SUB1}[l1]\leq\mathrm{SUB2}[l2]$, the inequality being strict except for $l1=l2=0$. If now, for some i, we reach the condition $\mathrm{SUB1}[l1]<\mathrm{SUB2}[l2]<S[i]$ (line 11) it is clear that the original $S[1:l]$ contains an *increasing* subsequence of length 3, which makes impossible its partitioning into two *nonincreasing* subsequences.

One may note that the algorithm easily generalizes to an algorithm for partitioning an arbitrary sequence of length l into the minimal possible number of nonincreasing subsequences, in time $O(l \log d)$, where d is this minimal number of subsequences, or – equivalently – the maximal length of an increasing subsequence in the given sequence.