1986

New Clique and Independent Set Algorithms for Circle Graphs

Alberto Apostolico

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Report Number:
86-608

NEW CLIQUE AND INDEPENDENT SET ALGORITHMS FOR CIRCLE GRAPHS

Alberto Apostolico
Mikhail J. Atallah
Susanne E. Hambrusch

CSD-TR-608
June 1986
(Revised May 1990)

New Clique and Independent Set Algorithms for Circle Graphs

Alberto Apostolico, Mikhail J. Atallah** and Susanne E. Hambrusch****
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA.

CSD-TR-608
June 1986
Revised November 1987, May 1990

Abstract. Given the interval model of an n-vertex, e-edge circle graph G, it is shown how to find a clique of G of maximum size l (resp., maximum weight) in $O(n\log n+\min[e, nl\log(2n/l)])$ (resp., $O(n\log n+\min[n^2, e\log\log n])$) time. The best previous algorithms required, respectively, $\Theta(n^2)$ and $O(n^2+e\log\log n)$ time. An $O(n\log n+dn)$ time and space algorithm that finds an independent set of maximum weight for the interval model of G is also presented. Here d is the maximum number of intervals crossing any position of the line in the interval model of G. The best previous solution for this problem took time $O(n^3)$.

* This research was supported in part by the French and Italian Ministries of Education, by the British Research Council Grant SERC-E76797, by NSF Grant CCR-89-00305, by NIH Library of Medicine Grant R01 LM05118, by AFOSR Grant 89NM682, and by NATO Grant CRG 900293.
** This research was supported in part by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.
*** This research was supported in part by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under Grant MIP-87-15652.

1. Introduction

Let I be a set of n (possibly weighted) intervals of the real line, such that no two intervals share a common endpoint. Interval i is represented by the ordered pair (le_i, re_i) of its endpoints on the real axis, and the weight of i is denoted by w_i, $1 \le i \le n$. Possibly at the cost of an $O(n \log n)$ time sorting, we can always assume that the intervals are numbered from 1 to n according to the natural order of their left endpoints, i.e., $i < j$ iff $le_i < le_j$. Let $i < j$. Interval i *contains* interval j if $re_j < re_i$. Intervals i and j are *disjoint* if $re_i < le_j$. Finally, intervals i and j *overlap* if they are not disjoint but neither one of them contains the other. As is well known (cf., for example, [Go]), the set I can be regarded as the *interval model* of a *circle* (or *overlap*) graph $G = (V,E)$, $|V| = n$, $|E| = e$, as follows. Intervals in I are in one-to-one correspondence with vertices in V, and two vertices are adjacent in G iff the corresponding intervals overlap. Thus, a set of mutually overlapping intervals of I models a clique of G. Likewise, a set of intervals that are pairwise either disjoint or contained in one another models an independent set of G.

In general, algorithms for circle graphs work with this interval model [Ga1, Go, GLL, H, RU], and so do the algorithms in this paper. We present new algorithms for the problems of finding optimal (i.e., maximum size l, maximum weight) cliques and independent sets of circle graphs G in interval form. For the unweighted case we present an algorithm that finds a clique of maximum size l in linear space and in time $O(n \log n + \min[e, nl\log(2n/l)])$. For the weighted case we present two algorithms which, when combined, give a $O(n \log n + \min[n^2, e \log\log n])$ running time. The best previous algorithms take $O(n^2)$ time for the unweighted [Bu, RU] and $O(n^2 + e \log\log n)$ time for the weighted case [H]. For the maximum independent set problem on a circle graph G with arbitrary weights we present an $O(n \log n + nd)$ time algorithm, where d is the *density* (i.e., the maximum number of intervals crossing any position on the line) of the interval model of G. The previously best algorithm requires $O(n^3)$ time [Ga1], although a straightforward extension of some results of [MS] would lead to a bound of $O(n^2)$.

The algorithm for the unweighted case and one algorithm for the weighted case are solved by reducing the problem to that of repeatedly finding an optimal (i.e., longest and heaviest, respectively) chain in suitably defined partially ordered sets. These reductions are described in Sections 2 and 3. They enable to set up the improved algorithms for the clique problems, as we show in Section 4. Section 4 also contains an $O(n^2)$ time algorithm for the weighted case which uses a different approach. Finally, Section 5 contains our maximum independent set algorithm.

2. Preliminaries for the Clique Algorithms

In this section we give a number of definitions used throughout the paper, define a family of permutation graphs of a circle graph, and describe an $O(n^2 \log\log n)$ time algorithm for the clique problem.

In $O(n \log n)$ time, the set of intervals I can be represented as a string $\alpha = \alpha_1 \alpha_2 \cdots \alpha_{2n}$ that we call the *encoding* of I (or G). An example is shown in Fig. 1. String α is a permutation of the set $\{1,1,2,2,3,3,\cdots,n,n\}$ that captures all mutual relations (disjointness, containment, overlap) among the intervals in I. The two occurrences of i in α mark the endpoints of interval i. Note that the first occurrence of i in α precedes the first occurrence of $i+1$. Through this transformation, the endpoints of each interval are encoded by integers (i.e., positions of α). We retain the notation (le_i, re_i) for the integer-encoded endpoints of interval i.

Our clique algorithms are based on some simple properties of circle graphs. In short, a vertical line drawn between positions m and $m+1$, $(m=1,2,\cdots,2n-1)$ of the (integer encoded) interval model induces, on all intervals intersected by that line, a subgraph G_m of G such that G_m is a permutation graph [PLE, EPL] (see also [Go]). Thus, our problem reduces to finding a maximum or maximum-weighted clique for all permutation graphs G_m.

The graphs G_m are easily extracted from the encoding α. For this, recall that there is a

natural total order for the intervals in I: $i < j$ if $le_i < le_j$. For each permutation graph G_m, the ordering of the vertices is a partial suborder of this initial order. Let the intervals from vertices of G_m be re-numbered in the suborder. Now, scanning the right-endpoints of those intervals from right to left yields the permutation π_m of G_m. In Fig. 1, the permutation induced by the vertical line is $\pi_7 = [5,4,1,2,3]$. It is well known (see, for example, [Go]), that the decreasing subsequences of π_7 and the cliques of G_7 are in one-to-one correspondence. Clearly, such a bijection carries on to the original sequence [6,5,2,3,4], from which π_7 was obtained. Thus the maximum (resp., heaviest) clique problem on $G_m, (m=1,2,\cdots,2n-1)$ translates into a corresponding longest (resp., heaviest) descending subsequence problem. For example, [6,5,2], [6,5,3] and [6,5,4] are the longest descending subsequences in [6,5,2,3,4], each one of which identifies a maximum clique of G_7. The longest (or heaviest) descending (or ascending) subsequence problem is a known restriction of the longest (or heaviest) common subsequence problem (see, for example, [Ap, HS, Fr2]). Such restrictions can be solved in $O(n\text{loglog}n)$ time and linear space. Iterating through the $2n-1$ graphs G_m leads therefore to an $O(n^2\text{loglog}n)$ algorithm. We show, however, that a closer look at how G_{m+1} is related to G_m yields some improvement. To highlight this relation, let every edge of G_m $(m=1,2,\cdots,2n-1)$ be transitively oriented from the higher to the lower numbered vertex. Observe now that G_{m+1} is obtained from G_m either by deletion of a vertex that is always a sink in G_m or by the addition of a vertex that is always a source in G_{m+1}. On the interval model, the first (resp., second) case occurs when moving the vertical line one position to the right decrements (resp., increments) by one the number of intervals crossing the line. In the following, we shall base our discussion on the permutations π_m rather than on the graphs G_m.

We write $\alpha^{(m)}$ and $\alpha_{(m)}$ to denote the prefix of α of length m $(m=1,2,\cdots,2n-1)$, and the corresponding suffix of α, respectively. Let # be the concatenation operation. With each decomposition $\alpha=\alpha^{(m)}\#\alpha_{(m)}$ $(m=1,2,\cdots,2n-1)$, we associate a $2n\times m$ binary array $M^{(m)}$.

Entry $M^{(m)}[i,j]=1$ iff $i>m$ and the j-th symbol of $\alpha^{(m)}$ equals the $(i-m)$-th symbol of $\alpha_{(m)}$ (or, equivalently, equals the i-th symbol of α). Nonzero entries of an M-array are called *points*, and we use $P^{(m)}$ to denote the set of points in $M^{(m)}$.

Fig. 2 shows the set of points resulting from the decompositions of the string α of Fig. 1 for $m=1,2,\cdots,7$. Observe that the transition from $M^{(m)}$ to $M^{(m+1)}$ follows one of the two following patterns. If the first symbol of $\alpha_{(m)}$ appears also in $\alpha^{(m)}$, then $M^{(m+1)}$ is obtained from $M^{(m)}$ by deleting the topmost point in $M^{(m)}$ and adding an empty column to it. (Note that the deleted point represents the sink of G_m having lexicographically least coordinates in $M^{(m)}$.) If, instead, the first symbol of $\alpha_{(m)}$ does not occur in $\alpha^{(m)}$, then $M^{(m+1)}$ consists of $M^{(m)}$ with a nonempty column added to it (the point in this column represents a source of G_m).

For any m, removing from $M^{(m)}$ all rows and columns that contain no points yields the permutation matrix of the reverse $(\pi_m)^R$ of π_m. The above observations, and the fact that the M-arrays are particular instances of the match-tables commonly used in the longest common subsequence and related string-editing problems, motivate our use of this representation.

With any set $P^{(m)}$, we associate the partial order R defined as follows: let $p=(i,j)$ and $q=(r,s)$ be elements of $P^{(m)}$; then pRq iff $i>r$ and $j>s$. Thus pRq iff q is "above and to the left of" p. Note that the comparability graphs induced by R are in fact the permutation graphs discussed in [PLE, EPL]. As usual, a subset Q of $P^{(m)}$ formed by elements that are linearly ordered by R will be called a *chain*; a subset S of $P^{(m)}$ formed by elements no two of which are in R is an *antichain*. In conclusion, the problem of computing a maximum (resp., heaviest) clique of G reduces to finding a longest (resp., heaviest) chain of $P^{(m)}$, $m=1,2,\cdots,2n-1$.

We focus first on the problem of finding a longest chain in a set $P^{(m)}$. By Dilworth's Theorem [Di], such a chain meets all the antichains of $P^{(m)}$. Thus the length l of the longest chain of $P^{(m)}$ equals the number of antichains in a minimal decomposition of $P^{(m)}$. (Note that such a decomposition corresponds to a minimal coloring of the underlying permutation graph

[PLE. EPL].) Fig. 3 shows one possible minimal antichain decomposition of a set of points and one longest chain. For a point p in $P^{(m)}$, let the *rank* of p be the length of a longest chain that p can form using only points below and to the right of itself. The following known scheme constructs a minimal antichain decomposition $S_1, S_2, \cdots$ that is often called *canonical* (cf. Fig. 4). In a canonical decomposition, the antichain S_r contains precisely the points of rank r.

We start at the bottom of the array $M^{(m)}$ and assign the point with the maximum row-coordinate to antichain S_1. Assume now that we have reached the i-th row and have computed the canonical decomposition for all points (x,y) with $x>i$. Let $S_1, S_2, \cdots, S_r$ be the antichains in this decomposition. Let s_k be the column-coordinate of the last point added to S_k, $1 \le k \le r$, and let $T = [s_1, s_2, \cdots, s_r]$. As Fig. 5 highlights, T is a sorted (in decreasing order) table. To assign the point (i,j), locate the pair (s_l, s_{l+1}) such that $s_l > j > s_{l+1}$. Now assign (i,j) to S_{l+1}, and replace s_{l+1} with j in T. If, in the above, s_{l+1} is not defined, simply initialize $S_{r+1}=\{(i,j)\}$ and append j to T. The data structure in [vE] can be used to implement table T. This data structure requires $\Theta(n)$ time for initialization, after which any *search*, *insert* or *delete* operation takes only $O(\text{loglog}n)$ time. The above strategy requires $|P^{(m)}|$ operations of search-with-insertion in T, and no more than $|P^{(m)}|$ deletions from T, whence its total cost is $O(n+|P^{(m)}|\text{loglog}n)$. In additional $\Theta(|P^{(m)}|)$ time, appropriate "chain-links" can be issued during the construction: such links will enable to retrieve a longest chain by backtracking through them at the end.

For the weighted case, a heaviest chain can be obtained by a similar bottom-up sequence of operations. This time we want to assign each point p to a heaviest chain among all chains that are formed using points of $M^{(m)}$ that lie below and to the right of p. The canonical decomposition is of no use in this case, but it suffices to maintain at each row i a simple variant of table T. Let $W_{i-1}[j]$, $(j=1,2,\cdots,m)$ be the weight of a heaviest clique among the cliques that use only points that fall both below row i and to the right of column j. Clearly, $W_{i-1}[j]$ cannot decrease as j goes from m to 1. Call *thresholds* the values of j for which $W_{i-1}[j] > W_{i-1}[j+1]$.

The new table T stores the sequence of threshold columns, each such column carrying its associated W-value. The rest of the construction is left to the reader (or see [PLE]). Note that possibly more than one threshold has to be deleted from T after an insertion. However, each threshold corresponds to a distinct point, and a deleted point is never re-inserted, i.e., the total number of deletions is still $O(|P^{(m)}|)$.

In conclusion, the two problems of finding a longest and a heaviest chain in $P^{(m)}$ can be both solved in time $O(n+|P^{(m)}|\log\log n)$. It is not difficult to see that iterating the above strategies $2n-1$ times yields another $O(n^2\log\log n)$ time solution for the corresponding clique problems on circle graphs. In the next two sections we describe algorithms that, given the encoding α of a circle graph G, find a maximum clique of G in $O(\min[e.nl\log(2n/l)])$ time, and a maximum weighted clique of G in time $O(\min[n^2, e\log\log n])$. Both bounds are advantageous for sparse graphs and neither is ever worse than $O(n^2)$. The first bound is also advantageous when l is expected to be small compared to n.

3. Maintaining Canonical Decompositions

We have already noted that, in the transition from $\alpha^{(m)}\#\alpha_{(m)}$ to $\alpha^{(m+1)}\#\alpha_{(m+1)}$, only one of two possible changes may affect the associated M-arrays. We call *contraction* the change that occurs when the topmost point is removed from $M^{(m)}$, *expansion* the change that occurs when a nonempty rightmost column is added to $M^{(m)}$. Assume that we have the canonical decomposition $S_1, S_2, \cdots, S_l$ for $P^{(m)}$. By construction, the points in S_r $(1\le r\le l)$ appear in lexicographically decreasing order. Thus the topmost point in $M^{(m)}$ is the lexicographically least among the last entries of $S_1, S_2, \cdots, S_l$. We call such a point *lowsink*, to remind that the corresponding vertex is a sink in (our transitive orientation of) G_m.

If $M^{(m+1)}$ is obtained from $M^{(m)}$ by contraction, its canonical decomposition can be trivially derived from that of $M^{(m)}$. We extract the list of the last elements in all antichains, and

search this list for the lexicographically least point. As is easily checked, removing *lowsink* from its antichain leaves with the canonical decomposition of $P^{(m+1)}$. With some trivial book-keeping, these manipulations can be carried out in $O(l)$ time.

In the rest of this section, we address the problem of maintaining the canonical decomposition of $P^{(m)}$ under expansion. We shall need the following notion. Given a point p in one of our sets P, $range(p)$ is subset of P which is formed by all points $q_1, q_2, \cdots q_h$ such that q_sRp $(s=1,2,\cdots,h)$, but for no value of s it is possible to find a point $\bar{q}\in P$, such that $q_sR\bar{q}Rp$. For the example of Fig. 6, $range((6,6))$ = {(9,8), (7,10)}. Clearly, $range(p)$ is an antichain. The notion of canonical decomposition can be re-formulated in terms of ranges, as follows. Points whose ranges are empty are assigned to S_1. Assume now that $S_1, S_2, \cdots, S_{r-1}$ have been constructed. Then, any currently unassigned point p of P belongs to S_r iff there is at least one point $\bar{p}$ such that $\bar{p}\in S_{r-1}\cap range(p)$.

As the example of Fig. 6 illustrates, the canonical decomposition of a set P can change considerably following an expansion. However, it is still possible to interpret the canonical decomposition of Fig. 6 as obtained from that of Fig. 4 through a series of elementary transformations. We use Fig. 7 to clarify this point. Let $p=(i,m+1)$ = (8,13) be the point being added to P. Let $S_1, S_2, \cdots, S_l$ and $\bar{S}_1, \bar{S}_2, \cdots, \bar{S}_{\bar{l}}$ represent, respectively, the canonical decompositions of P and $\bar{P}=P\cup\{p\}$. The first obvious observation is that p=(8,13) must belong to $\bar{S}_1$. Moreover, upon adding (8,13) to P, the rank of some of the points in P increases by one. In particular, all points in S_1 having (8,13) in their range (as, for example, (7,10)) acquire rank 2 in $\bar{P}$. Hence, adding point (8,13) to P splits S_1 into two segments, SL and SR, such that $\bar{S}_1$ = $SL\cup\{(8,13)\}$ and SR is to be made a suffix of $\bar{S}_2$. Forcing SR into $\bar{S}_2$ can, in turn, result into splitting S_2 into two parts: a prefix, which belongs to $\bar{S}_2$, and a suffix which is to become a suffix of $\bar{S}_3$. In our example, this suffix is ((6,6),(2,7)) which coincides already with $\bar{S}_3$. Setting

$S_3=\bar{S}_4$ concludes our construction.

For simplicity, we assume henceforth that the S-lists store only the row-coordinate of each point. Thus each such list is a sorted (in decreasing order) list of integers. The procedure EXPAND given below incorporates the above ideas. It assumes appropriate initializations and all parameters global except the first coordinate i of the point being added to $P^{(m)}$. The formalism follows that of [AHU].

Procedure EXPAND(i)
Input: *the canonical decomposition of* $P^{(m)}$*; the length* l *of a longest chain in* $M^{(m)}$.
Output: the canonical decomposition of $P^{(m+1)} = P^{(m)} \cup \{(i,m+1)\}$*; the updated length* $\bar{l}$.
begin
    $S=\{i\}$
    $r=1$
    while $S_r \neq \emptyset$ and $S \neq \emptyset$ and $Max(S)<Max(S_r)$ **do**
        begin
            $j=Max(S)$;
            $(SL,SR) = split(j,S_r)$;
            (* SL contains all entries of S_r which are larger than j. *)
            (* SR contains all entries of S_r which are smaller than j. *)
            (* At the next iteration, SR will be made a suffix of $\bar{S}_{r+1}$. *)
            $\bar{S}_r=concatenate(SL,S)$;
            $S=SR$;
            $r=r+1$;
        end
    if $S \neq \emptyset$ **then** $\bar{l}=l+1$;
end.

The correctness of EXPAND follows directly from our second definition of canonical decomposition. In fact, point $(i,m+1)$ trivially belongs to $\bar{S}_1$, and so do the points in the list SL that results from the splitting of S_1. The r-th iteration of the **while** loop uses the fact that, of all points considered so far, precisely the points placed in S have each at least one point of $\bar{S}_r$ in its range. Thus, precisely these points need to change their rank from r to $r+1$.

We now turn to the implementation of EXPAND. The critical part is in the searches (in the S-lists) implied by the *split* operations. One obvious way to locate the splitting site in S_r is

to start at the top (last inserted element) of the list and scan it until the splitting site is found. Observe that, in the graph G associated with the set of intervals I, the point $(i,m+1)$ being inserted by EXPAND is adjacent to all and only the points in $M^{(m)}$ that have row-coordinate smaller than i. Thus, there is a one-to-one correspondence between these points and the edges incident with $(i,m+1)$ in G. In the linear scan of the S-lists, we charge the work done in traversing each point p to the edge of G that connects p to $(i,m+1)$. Thus the total work in this implementation of EXPAND is $O(e_i)$, where e_i is the number of edges incident with $(i,m+1)$ in G. The space required is trivially $O(|P^{(m+1)}|)$. An alternate implementation of EXPAND is discussed in the following lemma.

Lemma 1. The procedure EXPAND can be implemented to run in time $O(\bar{l}\log(2|P^{(m+1)}|/\bar{l}))$ and linear space.

Proof. Implement the lists S_r as balanced (e.g., 2-3) trees [AHU]. The linear space bound is then straightforward. The operations performed outside the **while** loop require $O(\log l)$ time. Inside the **while** loop $t \le l$ *split* and *concatenate* operations are executed which cost a total of $\sum_{r=1}^{t}\log|S_r|$ and $\sum_{r=1}^{t}\log|\bar{S}_r|$ time, respectively, up to a multiplicative constant. If we add to the first sum the work of a dummy split on $S=\{(i,m+1)\}$ and assume that S is concatenated with the empty set whenever the **while** loop is exited with $S\neq\emptyset$, each one of the above sums can be rewritten as $\sum_{r=1}^{t}\log x_r$, where $t\le\bar{l}$ and $\sum x_r \le k+1=|P^{(m+1)}|$. Under these constraints, both sums are maximized by choosing $x_r=(k+1)/\bar{l}$. Thus, the work accumulated in the **while** loop can be bounded by $\bar{l}(1+\log((k+1)/\bar{l}))$, and the total work performed by EXPAND is $O(\bar{l}\log(2|P^{(m+1)}|/\bar{l}))$. □

Combining the two above implementations of EXPAND (e.g., by running them concurrently) yields an algorithm taking $O(\min[\, e_i, \bar{l}\log(2|P^{(m+1)}|/\bar{l})\,])$.

4. Finding Maximum and Maximum-Weight Cliques

We now present the algorithms for the unweighted and weighted clique problem on circle graphs. Section 4.1 contains the algorithm for the unweighted case and the $O(n\log n+e\log\log n)$ time algorithm for the weighted case. Both algorithms make use of the techniques developed in the previous two sections and can thus be viewed as adaptive. As m goes from 2 to $2n$, the computation of an optimal clique for G_m makes use of the information accumulated while computing an optimal clique for G_{m-1}. Section 4.2 contains an $O(n^2)$ time algorithm for the weighted case. This algorithm is not of the adaptive kind. Rather, it computes the weights of many maximal chains for a set of points obtained from taking the union all sets $P^{(m)}$.

4.1. Algorithms Based on Expansion and Contraction

We have seen that the canonical decompositions of the graphs G_m can be maintained efficiently through the expansions and contractions dictated by the structure of α, as m goes from 0 to $2n-1$. Straightforward $\Theta(n)$ preprocessing of α enables subsequently to decide, in constant time for each of the above values of m, whether α_{m+1} encodes a left or a right endpoint. For example, one can construct a table *twin*, defined as follows: for $t=1,2,\cdots,2n$, $twin[t]=s$ iff $\alpha_t=\alpha_s$. A right endpoint is then detected at iteration m by the condition $twin[m+1]<m+1$. This calls for a contraction, i.e., the search and removal of the current *lowsink* from the canonical decomposition of $P^{(m)}$. (Incidentally, note that *lowsink* is the point $(twin[m+1],m+1)$ in $M^{(m)}$; cf. Fig 2.) As seen at the beginning of Section 3, these manipulations are trivially carried out on the antichains of the canonical decomposition of $P^{(m)}$ in $O(l)$ time, where l is the length of a globally optimal chain. If $twin[m+1]>m+1$, then iteration m involves an expansion. To invoke the procedure EXPAND, we need the row-coordinate in $M^{(m+1)}$ of the point being added to $P^{(m)}$ (the column-coordinate of this point is just $m+1$). It is easy to check (cf. Fig. 2) that this row-coordinate is precisely $twin[m+1]$. In conclusion, we know always how to locate *lowsink* in a contraction, and we also know how to generate quickly

the parameter value for EXPAND. The value of m for which $P^{(m+1)}$ achieves a longest possible chain can be computed in the process. At the end, running the algorithm of Section 2 on $P^{(m+1)}$ yields the final solution in additional time $O(n+|P^{(m+1)}|\log\log n)$. We omit the details. Thus, a maximum clique of a circle graph G can be computed in $O(\min[e, nl\log(2n/l)])$ time and $O(n)$ space from the encoding α of the interval model I of G. Adding the $O(n\log n)$ cost of producing α from I leads to our claimed bound.

We now turn to the computation of a maximum-weight clique of G. We have already observed that the canonical decompositions of the sets $P^{(m)}$ do not seem to help, in general, in finding a maximum weighted clique. However, we saw in Section 2 that a heaviest chain of $P^{(m)}$ can be found in time $O(n+|P^{(m)}|\log\log n)$. To simplify that discussion, we took the array $M^{(m)}$ as the input, and we made the implicit assumption that the column-coordinate of the point in any nonempty row of $M^{(m)}$ could be found in constant time. This can be arranged easily. However, a more natural input is the "vertical" list of points in $P^{(m)}$, sorted in lexicographically descending order. If such a list is available, this will spare us the time previously spent in examining empty rows of $M^{(m)}$. This does not change the above time bound, but the linear term in it is now charged solely by the initialization of the priority queue of [vE]. At this point, we can use instead the variant of this structure presented in [Jo], which carries an initialization cost proportional to the total cost of the insertions. Thus, if the input is the vertical list of points of $P^{(m)}$, the strategy of Section 2 can be implemented in $O(|P^{(m)}|\log\log n)$ time. Combined with the table *twin*, either priority queue in [vE, Jo] can be used to maintain our vertical list through the left-to-right scanning of α, at an overall cost of $O(n+n\log\log n)$. We will see next how these observations lead to a simple $O((e+n)\log\log n)$ algorithm for computing a heaviest clique of G from α.

Let p be a generic point in $P^{(m)}$, and let $W_m(p)$ be the cost of a heaviest chain that can be formed using only p and points of $P^{(m)}$ that lie below and to the right of p in $M^{(m)}$. Assume

that $P^{(m+1)}=P^{(m)}\cup\{q\}$, for some suitable point q. Clearly, $W_{m+1}(p)$ may differ from $W_m(p)$ only if qRp. In other words, the only points whose W-value may vary following an expansion are those adjacent to q in G, i.e., those falling above q in $M^{(m+1)}$. The W-value of any such point may be increased only by a chain containing that point and q. But such a chain cannot contain any point falling below q, since q is the rightmost point in $M^{(m+1)}$. In summary, we have the following. First, the W_{m+1}-values of all points below q are identical to the corresponding W_m-values. Second, $W_{m+1}(q)=w_q$. Finally, running the above heaviest-chain algorithm on the lexicographically sorted list of the remaining points enables to assess the W_{m+1} values for all such points, in $O(e_q\text{loglog}n)$ time. Note that the input list that we need is just a suffix of the vertical list associated with $P^{(m+1)}$. Accessing such a suffix is a trivial byproduct of the insertion of point q in that vertical list. Clearly, the W-value of a point cannot increase following a contraction. Along these lines, we establish the bound of $O((n+e)\text{loglog}n)$ for the computation of a heaviest chain of G from the encoding α. Adding to this the $O(n\log n)$ preprocessing cost leads to the bound of $O(n\log n+e\text{loglog}n)$.

4.2. Algorithm BESTCHAINS

In this section, we give an alternate algorithm for finding a heaviest chain of G. As mentioned in the beginning of Section 4, this $O(n^2)$ time algorithm is quite different from the one given in Section 4.1, and the combination of the two will establish the overall claimed bound of $O(n\log n+\min[n^2, e\text{loglog}n])$ for the heaviest clique problem.

Before we describe our $O(n^2)$ algorithm, we introduce the notion of the trace associated with a family of M-arrays. The *trace* M^* is the $2n\times 2n$ array that is obtained by taking the union of all the arrays $M^{(m)}$, $(m=1,2,...,2n)$, or, equivalently, by adding to the array $M^{(2n)}$ all points that were deleted by contraction in the sequence of transitions from $M^{(1)}$ to $M^{(2n)}$. The trace for the arrays of string α of Fig. 1 is shown in Fig. 8. The following observations motivate the introduction of M^*. Let P^* be the set of all points in M^*. For every point $p=(i,j)$ in P^* and

every value $k \le i$, let $W_p[k]$ be the weight of a heaviest chain among those chains that start at p and use no point (i', j') with $i' < k$. Observe that, in general, some of the values in W_p do not correspond to cliques of G, since they refer to chains that use points not in $M^{(j)}$. By construction, however, the points that are both in M^* and $M^{(j)}$ are precisely those points of M^* having the second coordinate not larger than j and first coordinate not smaller than h, for some $h \le i$ (cf. Fig. 2). The value of h is known from the structure of $M^{(j)}$. Since $W_p[k]$ is obviously nonincreasing with increasing k, we conclude that $W_p[h]$ is the weight of a heaviest chain of $P^{(j)}$ among all chains of $P^{(j)}$ that start at point p. In other words, the computation of the W_p arrays for all points of P^* makes the weight of a heaviest chain of G readily available. Our algorithm computes these arrays, and we can now undertake its description.

Since the result that follows holds for an arbitrary set $P=\{p_1, p_2, \cdots, p_n\}$ of n weighted points in the plane, we relax our definition of the relation R, by including in R also pairs of points that have identical first or second coordinate. For any point p of P, let $X(p)$ and $Y(p)$ be the x and y coordinate of p, respectively. We assume that the points are given sorted by nonincreasing y coordinates, i.e. $Y(p_1) \ge Y(p_2) \ge \cdots \ge Y(p_n)$. Note that (0,0) is now the bottom left corner of the coordinate system.

Let V_{Left} be a vertical line to the left of P. Let $Left(P)=\{a_1, \cdots, a_n\}$ where a_i has zero weight and is the horizontal projection of p_i on V_{Left} (see Fig. 9).

Let $DIST_P$ be the matrix of the weights of heaviest chains in $P \cup Left(P)$ that begin in P and end in $Left(P)$. That is, $DIST_P(i,j)$ is the weight of a heaviest chain of points that begins at p_i, ends at a_j, and all of whose intermediate points (if any) are in $P \cup Left(P)$. If $i < j$ then $DIST_P(i,j) = -\infty$ (p_i and a_j form an antichain).

Lemma 2. Given P, the matrix $DIST_P$ can be computed in $O(n^2)$ time.

The rest of this section gives an algorithm that proves the above lemma.

Let V_{Right} be a vertical line to the right of P. Let $Right(P)=\{b_1, \cdots, b_n\}$ where b_i has zero weight and is the horizontal projection of p_i on V_{Right} (see Fig. 9). Let RL_P ("RL" being mnemonic for "right to left") be the matrix of the weights of heaviest chains in $P \bigcup Left(P) \bigcup Right(P)$ that begin in $Right(P)$ and end in $Left(P)$. In other words, $RL_P(i,j)$ is the weight of a heaviest chain of points that begins at b_i, ends at a_j, and all of whose intermediate points are in $P \bigcup Left(P) \bigcup Right(P)$. If $i<j$ then $RL_P(i,j)=-\infty$ (b_i and a_j form an antichain). In order for the recursive procedure we are about to describe to work, it must compute the RL_P matrix as well as the $DIST_P$ one.

We are now ready to describe the procedure BESTCHAINS, which takes as input $P=\{p_1, \cdots, p_n\}$, $Y(p_1) \geq \cdots \geq Y(p_n)$, and computes the matrices $DIST_P$ and RL_P. The basic idea is that of partitioning the problem of size n into two subproblems, of size $n/2$ each, which are then solved recursively. The $DIST$ and RL matrices returned by the two recursive calls are then combined in $O(n^2)$ time to obtain the $DIST$ and RL matrices of the original problem. The resulting recurrence relation for the time complexity is then $T(n) \leq 2T(n/2)+cn^2$, whose solution is $T(n)=O(n^2)$. The main difficulty is in combining sub-solutions in quadratic time. A more detailed description of BESTCHAINS is as follows.

Step 1. If P is small (e.g., contains less than 20 points) then solve the problem in constant time by using any brute force method. Otherwise proceed to Step 2.

Step 2. Let V_{Middle} be a vertical line partitioning P into two sets of points A and B, each of which contains $n/2$ points, and such that A is to the left of B (see Fig. 9). Using as $Left(A)$ (resp. $Right(A)$) the horizontal projection of A on V_{Left} (resp. V_{Middle}), recursively solve the problem for A. Then, using as $Left(B)$ (resp. $Right(B)$) the horizontal projection of B on V_{Middle} (resp. V_{Right}), recursively solve the problem for B. This step takes time equal to $2T(n/2)+O(n)$.

Comment. These recursive calls return $DIST_A$, RL_A, $DIST_B$, RL_B. The matrix RL_A contains the weights of the heaviest chains in $A \cup Left(A) \cup Right(A)$ that begin in $Right(A)$ and end in $Left(A)$. The matrix $DIST_A$ contains the weights of the heaviest chains in $A \cup Left(A)$ that begin in A and end in $Left(A)$. The matrix RL_B contains the weights of the heaviest chains in $B \cup Left(B) \cup Right(B)$ that begin in $Right(B)$ and end in $Left(B)$. The matrix $DIST_B$ contains the weights of the heaviest chains in $B \cup Left(B)$ that begin in B and end in $Left(B)$.

Before we proceed to Step 3, let $Middle(P)=\{c_1, \cdots, c_n\}$ where c_i has weight zero and is the horizontal projection of p_i on V_{Middle} (see Fig. 9). Recall that $Left(P)=\{a_1, \cdots, a_n\}$ (resp. $Right(P)=\{b_1, \cdots, b_n\}$) is the horizontal projection of P on V_{Left} (resp. V_{Right}).

Step 3. Use the matrices RL_A and RL_B to obtain RL_P. We need to do this in $O(n^2)$ time. This is done in the following sub-steps (3.1)-(3.3).

Sub-step 3.1. From RL_A, obtain the $n \times n$ matrix $RL1$ of the weights of heaviest chains in $Left(P) \cup A \cup Middle(P)$ that begin in $Middle(P)$ and end in $Left(P)$. This is easy to do in $O(n^2)$ time, as follows. We set the entry $RL1(i,j)$ equal to $RL_A(f(i),g(j))$ where:

(i) Row $f(i)$ of RL_A corresponds to the lowest point of $Right(A)$ that is not below c_i (possibly it is c_i itself).

(ii) Column $g(j)$ of RL_A corresponds to the highest point of $Left(A)$ that is not above a_j (possibly it is a_j itself).

If $f(i)$ or $g(j)$ is undefined (e.g., if all points of A are below c_i or above a_j), then we set $RL1(i,j)$ equal to $-\infty$. Of course, locating $f(i)$ and $g(j)$ for each i,j pair is not done by binary search (this would result in an unacceptable $O(n^2 \log n)$ cost for this sub-step). Rather, the computation of the functions f and g is done all at once in $O(n)$ total time, as follows. To compute f, merge $Right(A)$ with $Middle(P) - Right(A)$ (each sorted by nonincreasing y components) and, during the merge, compute for each point of $Middle(P)$ the lowest point of $Right(A)$ that is

not below it. To compute g, merge $Left(A)$ with $Left(P) - Left(A)$ (each sorted by nonincreasing y components) and, during the merge, compute for each point of $Left(P)$ the highest point of $Left(A)$ that is not above it.

Sub-step 3.2. From RL_B, obtain the $n \times n$ matrix $RL2$ of the weights of heaviest chains in $Middle(P) \cup B \cup Right(P)$ that begin in $Right(P)$ and end in $Middle(P)$. This is done in $O(n^2)$ time in a manner similar to the way $RL1$ was obtained in sub-step 3.1.

Sub-step 3.3. Use $RL1$ and $RL2$ to obtain RL_P. We show that this can be done in $O(n^2)$ time. Note that:

$$RL_P(i,j) = \max_{1 \le k \le n} (RL2(i,k) + RL1(k,j)) \qquad (*)$$

Thus the problem we face is that of "multiplying" the matrix $RL2$ and the matrix $RL1$ in the closed semiring (max,+). The key observation which enables us to perform this multiplication in $O(n^2)$ time is now given. For every row i of $RL2$ and every column j of $RL1$, let $\theta(i,j)$ be the value of k which maximizes (*), i.e. $RL_P(i,j) = RL2(i,\theta(i,j)) + RL1(\theta(i,j),j)$. If there is more than one value of k which maximizes (*) then we break the tie by choosing $\theta(i,j)$ to be the smallest such k (this correspond to breaking ties in favor of chains that cross V_{Middle} as high as possible). The key observation is that for every row i of $RL2$ and every column j of $RL1$, we have:

$$\theta(i,1) \le \theta(i,2) \le \cdots \le \theta(i,n) \quad \text{and} \quad \theta(1,j) \le \theta(2,j) \le \cdots \le \theta(n,j). \qquad (\dagger)$$

Before proving property ($\dagger$), we explain how a consequence of it would be an $O(n^2)$ time algorithm for doing the matrix multiplication defined by (*). We give an $O(n_1 n_2)$ time procedure for the (more general) case where $RL2$ is an $n_1 \times n_2$ matrix, and $RL1$ is an $n_2 \times n_1$ matrix, $n_1 \le n_2$. The only structure of these matrices that our algorithm uses is the property ($\dagger$). To compute the product of $RL2$ and $RL1$ in the closed semiring (max,+), it clearly suffices to compute $\theta(i,j)$ for all $1 \le i,j \le n_1$. To compute the product of $RL2$ and $RL1$ (i.e. the function θ), we use the following recursive procedure.

1. Recursively solve the problem for the product of $RL2'$ and $RL1'$ where $RL2'$ (resp. $RL1'$) is the $(n_1/2)\times n_2$ (resp. $n_2\times(n_1/2)$) matrix consisting of the odd rows (resp. odd columns) of $RL2$ (resp. $RL1$). This gives $\theta(i,j)$ for all pairs (i,j) such that i and j are odd. If $T(n_1,n_2)$ denotes the time complexity of the overall procedure, then this step takes $T(n_1/2,n_2)$ time.

2. Compute $\theta(i,j)$ for all even i and odd j, as follows. For each odd j, compute $\theta(i,j)$ for all even i. The fact that we already know $\theta(i,j)$ for all odd i, together with property ($\dagger$), implies that this can be done in $O(n_2)$ time for each such j. The total time taken by this step is then $O(n_1 n_2)$.

3. Compute $\theta(i,j)$ for all odd i and even j. The method used is identical to that of the previous step and is therefore omitted.

4. Compute $\theta(i,j)$ for all even i and even j. The method is very similar to that of the previous two steps and is therefore omitted.

The time complexity of the above method obeys the recurrence: $T(n_1,n_2)\le T(n_1/2,n_2)+cn_1n_2$, where c is a constant. This implies that $T(n_1,n_2)=O(n_1n_2)$.

Thus it suffices to prove ($\dagger$). We give the detailed proof that $\theta(i,1)\le\theta(i,2)\le\cdots\le\theta(i,n)$ and omit the proof of $\theta(1,j)\le\theta(2,j)\le\cdots\le\theta(n,j)$ since it is symmetrical. Since the row i of $RL2$ is understood, we use $\theta(k)$ as a shorthand for $\theta(i,k)$. The proof is by contradiction: suppose that for some j we have $\theta(j)>\theta(j+1)$. By definition of the function θ there is, in $P\cup Left(P)\cup Middle(P)\cup Right(P)$, a heaviest chain from b_i to a_j going through $c_{\theta(j)}$ (call this chain μ), and one from b_i to a_{j+1} going through $c_{\theta(j+1)}$ (call it chain β). Let $path(\mu)$ be the piecewise linear path obtained by joining by a straight line segment every two consecutive points of μ, and let $path(\beta)$ be defined similarly for β (see Fig. 10). Since $c_{\theta(j+1)}$ is above $c_{\theta(j)}$, the two continuous paths $path(\mu)$ and $path(\beta)$ must cross at least once somewhere in between V_{Middle} and V_{Left}: let q be such an intersection point (q need not belong to P; see Fig. 10). Let

prefix(μ) (resp. *prefix*(β)) be the chain consisting of the portion of the chain μ (resp. β) that is (geometrically) to the right of q. We obtain a contradiction in each of two possible cases:

Case 1. The length of *prefix*(μ) differs from that of *prefix*(β). Without loss of generality, assume it is the length of *prefix*(β) that is the larger of the two. But then, the chain obtained from μ by replacing *prefix*(μ) by *prefix*(β) is better (i.e. heavier) than μ, a contradiction.

Case 2. The length of *prefix*(μ) is same as that of *prefix*(β). In μ, replacing *prefix*(μ) by *prefix*(β) yields another heaviest chain between b_i and a_j, one that crosses V_{Middle} at a point higher than $c_{\theta(j)}$, contradicting the definition of the function θ.

This completes the proof of (†).

Step 4. Use the matrices $DIST_A$, $DIST_B$, and $RL1$ to obtain $DIST_P$. We need to do this in $O(n^2)$ time. This is done in the following sub-steps (4.1)-(4.3).

Sub-step 4.1. From $DIST_A$, obtain the $(n/2)\times n$ matrix $D1$ of the weights of heaviest chains in $A \cup Left(P)$ that begin in A and end in $Left(P)$. This is easy to do in $O(n^2)$ time, as follows. Consider the entry of $D1$ corresponding to the chain from $p_i \in A$ to $a_j \in Left(P)$. If a_j is in $Left(A)$, then this entry of $D1$ is the same as the entry of $DIST_A$ whose row corresponds to p_i and whose column corresponds to a_j. If a_j is not in $Left(A)$ then this entry of $D1$ is the same as the entry of $DIST_A$ whose row corresponds to p_i and whose column corresponds to $a_{g(j)}$ (the function g was defined and computed in sub-step 3.1). Note that $D1$ contains half of the rows of the matrix $DIST_P$ (the rows corresponding to heaviest chains beginning in A).

Sub-step 4.2. From $DIST_B$, obtain the $(n/2)\times n$ matrix $D2$ of the weights of heaviest chains in $Middle(P) \cup B$ that begin in B and end in $Middle(P)$. This is done in $O(n^2)$ time in a manner similar to the way $D1$ was obtained in sub-step 4.1.

Sub-step 4.3. Use matrices $RL1$ and $D2$ to obtain the $(n/2)\times n$ matrix $D3$ of the weights of heaviest chains in $P \cup Left(P) \cup Middle(P)$ that begin in B and end in $Left(P)$. Note that $D3$

contains half of the rows of the matrix $DIST_P$ (the rows corresponding to heaviest chains beginning in B). We show that this sub-step can be done in $O(n^2)$ time. The algorithm, which we sketch next, is similar to sub-step 3.3. Note that:

$$D3(i,j) = \max_{1 \le k \le n} (D2(i,k) + RL1(k,j)) \qquad (**)$$

Thus the problem we face is that of "multiplying" the matrix $D2$ and the matrix $RL1$ in the closed semiring (max,+). The key observation which enables us to perform this multiplication in $O(n^2)$ time is a monotonicity property similar to $(\dagger)$. More specifically, for every row i of $D2$ and every column j of $RL1$, let $\gamma(i,j)$ be the value of k which maximizes (**), i.e. $D3(i,j) = D2(i,\gamma(i,j)) + RL1(\gamma(i,j),j)$. If there is more than one value of k which maximizes (**) then we break the tie by choosing $\gamma(i,j)$ to be the smallest such k. The key observation is that for every row i of $D2$ and every column j of $RL1$, we have:

$$\gamma(i,1) \le \gamma(i,2) \le \cdots \le \gamma(i,n) \quad \text{and} \quad \gamma(1,j) \le \gamma(2,j) \le \cdots \le \gamma(n/2,j). \qquad (\dagger\dagger)$$

The proof of $(\dagger\dagger)$ and the discussion about how it implies an $O(n^2)$ time algorithm for $D3$ are similar to the arguments given about $(\dagger)$ in sub-step 3.3 and are therefore omitted. This completes the description of the algorithm and hence the proof of Lemma 2.

BESTCHAINS can be easily upgraded so as to produce not only the weights of the heaviest chains starting at each point but also the chains themselves. These modifications do not alter the time bound. We leave them to the reader.

We conclude by noting that there is a connection between our implementation of substep 3.3 of the algorithm of Subsection 4.2 and recent work (independent of ours) by Aggarwal and Park [AgPa]: the problem they call "computing the tube maxima of a three-dimensional Monge matrix" is similar to our implementation of substep 3.3. The first two authors of the present paper, as well as Aggarwal and Park, have also independently considered the parallel version of this problem (these investigations are reported in [AALM] and [AgPa], respectively, and the techniques they use are quite different).

5. Finding a Maximum Independent Set

In this section we present an algorithm that finds a maximum independent set of a weighted n-vertex circle graph given by its interval model in $O(dn)$ time, where d is the *density* of the interval model. The density d is defined as $\max_q\{d_q\}$, where d_q is the number of intervals with $le_i<q<re_i$ (i.e., d_q is the number of intervals crossing from position q to position $q+1$). We assume that we are given the encoding α and that every one of the $2n$ positions in α knows whether it corresponds to the left or the right endpoint of an interval. Furthermore, position le_i (resp. re_i) in α corresponds to the left (resp. right) endpoint of interval i.

We start by briefly outlining a known $O(n)$ time dynamic programming algorithm that finds a maximum independent set of a weighted interval graph given in interval form by its encoding α [Fr1, MS, GLL]. In the interval model of an interval graph any two intervals in an independent set must be disjoint. Recall that in the interval model for a circle graph any two intervals in an independent set must either be disjoint or one must contain the other. The algorithm computes for every position m, $1\leq m\leq 2n$, an entry $MIS[m]$ that contains the sum of the weights of the intervals in a maximum independent set when considering all intervals i with $re_i\leq m$. Hence, $MIS[2n]$ contains the value of the optimal solution. The MIS-entries are computed in a left-to-right scan of the encoding α. Assume the scan has just reached position m. If this position corresponds to a left endpoint, then we set $MIS[m]$ to $MIS[m-1]$. If the position corresponds to a right endpoint, say of interval i, then we set $MIS[m] = \max\{MIS[m-1], MIS[le_i-1]+w_i\}$. It is straightforward to shown that this procedure determines a maximum independent set.

We now return to the maximum independent set problem on weighted circle graphs. The algorithm described above immediately leads to an $O(\min\{n^2, d^2n\})$ time solution. This solution is obtained by an implementation of Gavril's algorithm [Ga] in which for every interval i the value of the maximum independent set formed by interval i and the intervals contained by i

is computed. Let $CMIS[i]$ be this value. Our maximum independent set algorithms determines the $CMIS$-entries in $O(dn)$ time using a different method than the one described in [Ga]. Our algorithm computes the entries in a single left-to-right scan of the encoding α, with appropriate book-keeping. Once the $CMIS$-entries are known, the final value of the maximum independent set is obtained in $O(n)$ time by using the algorithm described above for interval graphs.

Assume the left-to-right scan reaches position m of the encoding α. At this point, the entries $CMIS[i]$ for all intervals i with $re_i<m$ have been computed. The entries $CMIS[i]$ for all intervals i with $le_i<m\leq re_i$ already have received a preliminary value. The intervals with $le_i<m\leq re_i$ are in a set, called set $OPEN$ (we call these intervals *open* intervals). Assume that for every open interval x the algorithm maintains a list $clist(x)$, which contains the following information about the intervals contained by x. Let u be an interval having both endpoints in $[le_x,m-1]$ (i.e., $le_x\leq le_u<re_u\leq m-1$). Then, let $clist(x)$ contain an entry (re_u,ww_u), where ww_u is the weight of a maximum independent set formed only by intervals having both endpoints in $[le_x,re_u]$. We next describe the actions taken at position m during the scan when the $clist$'s are available.

If position m corresponds to a left endpoint of some interval i, then add interval i to $OPEN$, set $CMIS[i]$ to zero, and create the (initially empty) list $clist(i)$. If position m corresponds to the right endpoint of some interval i, then remove i from $OPEN$ and assign to $CMIS[i]$ its final value; i.e., $CMIS[i] = CMIS[i]+w_i$. The algorithm then determines the effect of the final value of $CMIS[i]$ on other open intervals. The intervals that need to be considered are the open intervals x with $le_x<le_i$ (i.e., the ones that contain interval i). For every such interval x, $CMIS[x]$ is possibly updated and a new entry in $clist(x)$ is created. The right endpoint of this new entry is obviously re_i and its weight is determined as follows. Let (r,ww) be the entry in $clist(x)$ with $r<le_i$ and r as large as possible. Assume the right endpoint r belongs to interval u. We then say that interval u *updates* interval x at position m. If no interval updates x at m

(i.e., no such entry (r,ww) exists), assume $ww=0$. Next update $CMIS[x]$ to the maximum of the current $CMIS[x]$ and $CMIS[i]+ww$. Then, set the weight of the newly created entry in $clist(x)$ to $CMIS[x]$.

Standard balanced-tree implementations of the *clist*'s lead immediately to $O(dn\log n)$ time and $O(dn)$ space bounds since there are at most d *clist*'s at work at any position during the scan. Note that a right endpoint may form an entry in a number of *clist*'s and that these entries have, in general, different weights. We next describe an implementation of the above algorithm that achieves the claimed $O(dn)$ time bound. In order to remove the factor of $\log n$ in the time bound it is crucial that the open intervals are updated fast. Our new implementation makes use of the fact that if interval u updates interval x at position m, then u also updates every other open interval y with $le_y<le_x$. Of course, the weights needed to perform the actual updates on entries $CMIS[x]$ and $CMIS[y]$ may be different. The following lemma states that the interval updating x has its right endpoint within $2d$ positions to the left of position le_i.

Lemma 3. Let u be the interval updating the open interval x at position m. Then, $re_u>le_i-2d$.

Proof. Let q be the largest position in encoding α that corresponds to the left endpoint of an interval disjoint with interval i and $q<le_i$. Clearly, any right endpoint updating an open interval at position m must be in $[q+1, le_i-1]$. By our choice of q, any left endpoint falling in $[q+1, le_i-1]$ must have its corresponding right endpoint to the right of position le_i. And, any right endpoint falling in $[q+1\ le_i-1]$ must have its corresponding left endpoint to the left of position q. Thus, there cannot be more than $d-1$ such left (resp. right) endpoints. □

We next describe the data structures used by our maximum independent set algorithm. Set *OPEN* is implemented as a doubly-linked list with the open intervals arranged by increasing left endpoint. Hence, if list *OPEN* contains the intervals $\beta1, \beta2, \cdots \beta t$, then $le_{\beta1}<le_{\beta2}<\cdots<le_{\beta t}$, $t\le d$. Every position q of the encoding α corresponding to a right endpoint already encountered in the scan has a doubly linked list L_q associated with it. Let u be the

interval with $re_u=q$. Then, every element in L_q corresponds to an interval that contains interval u. Hence, there can be at most $d-1$ elements in L_q. If L_q contains the intervals $\gamma 1$, $\gamma 2$, ..., γt, $t \leq d$, then $le_{\gamma 1} > le_{\gamma 2} > \cdots > le_{\gamma t}$ (cf. Fig. 11). Every interval γs in L_q has a weight entry associated with it, and we refer to it as $w(L_q, \gamma s)$. This weight entry has the same function as the weight entry of the elements in the previously used *clist*'s, and it corresponds to the weight of a maximum independent set formed by intervals with both endpoints in $[le_{\gamma s}, q]$, $1 \leq s \leq t$.

The actions taken at the endpoints encountered during the left-to-right scan of α are as follows. As before, assume the scan is at position m. If the position corresponds to the left endpoint of some interval i, insert interval i into list *OPEN* and set $CMIS[i]$ to zero. If the position corresponds to the right endpoint of some interval i, then remove interval i from list *OPEN* and give $CMIS[i]$ its final value (i.e., $CMIS[i] = CMIS[i]+w_i$). The algorithm next performs an update stage in which the effect of $CMIS[i]$ on other open intervals is determined. The update stage traverses list *OPEN* starting from both ends. Let pointer *p_begin* point to the open interval with the smallest left endpoint (i.e., interval $\beta 1$ in list *OPEN*), and let *p_end* point to the first open interval x with $le_x < le_i$. Only the open intervals between *p_begin* and *p_end* in *OPEN* may need their *CMIS*-entry updated. In order to do so we start a local right-to-left scan in encoding α at position $q = le_i - 1$. Because of Lemma 3, this local scan needs never extend beyond the $2d$-*th* position to the left of position $le_i - 1$.

Assume the local scan is at position q in encoding α. If q corresponds to the left endpoint of some interval, it is either the left endpoint of some interval x with $re_x < m$ (in which case no action is taken), or it is the left endpoint of some open interval x (which has $re_x > m$). In the latter case there exists no interval that updates x at position m (otherwise the local scan would have encountered this interval and updated x at some earlier point). We set $CMIS[x]$ to the maximum of $CMIS[x]$ and $CMIS[i]$. Moreover, before moving to position $q-1$, we advance pointer *p_end*, which also points to interval x, to the next interval in *OPEN*.

If q corresponds to the right endpoint of some interval u, we determine the open intervals updated by u and perform the necessary updates. This step, which we call CHECK_UPDATE, scans, at least partially, list L_q. Recall that list L_q contains the intervals $\gamma 1, \gamma 2, \cdots, \gamma t$ with $le_{\gamma 1} > \cdots > le_{\gamma t}$, $t \leq d$, and that every interval γs contains u. CHECK_UPDATE starts by scanning list L_q to find the first interval $z = \gamma r$ that is open. Intervals $\gamma 1, \cdots, \gamma(r-1)$, which are no longer open, obviously need no more updating and are at this point removed from L_q. Every open interval between and including intervals *p_begin* and z contains interval u and is updated by u (cf. Fig. 11). The *CMIS*-entries are updated as before (see also step (2) of procedure CHECK_UPDATE given below). Furthermore, intervals encountered in L_q that are no longer open are removed from L_q. After the open intervals between *p_begin* and z have been updated, CHECK_UPDATE sets pointer *p_begin* for the next position, namely $q-1$, in the the local scan. The local scan still needs to update the open intervals y with $le_u < le_y < re_u$. These intervals (like, for example, $\beta 5$ in Fig. 11) could not be updated by interval u and their left endpoints have not yet been encountered in the local scan.

Procedure CHECK_UPDATE
Input: *lists L_q and OPEN.*
Output: *The updated CMIS-entries for all the open intervals that contain u.*

(* Currently, *b_begin* points to the open, not yet updated interval in *OPEN* with the *)
(* smallest left endpoint, and *p_end* points to the open, not yet updated interval in *OPEN* *)
(* with the largest left endpoint. Intervals $\gamma 1, \gamma 2, \cdots, \gamma t$ are the intervals in list L_q with *)
(* $le_{\gamma 1} > \cdots > le_{\gamma t}$, $t \leq d$ *)
begin

(1) let z be the first open interval in L_q;
assume $z = \gamma r$; if $r > 1$, delete intervals $\gamma 1, \cdots \gamma(r-1)$ from L_q;

(2) **while** the interval *p_begin* points to has not been passed in list L_q **do**
$CMIS[z] = \max\{CMIS[z], w(L_q, z) + CMIS[i]\}$;
set z to the index of the next open interval in L_q and delete intervals encountered in L_q that are no longer open
endwhile;

(3) advance *p_begin* so that it points to the first interval in *OPEN* encountered with a left endpoint > le_u

end.

After the update stage has been completed, the final action to be taken at a position m is the creation of list L_m. Recall that position m corresponds to the right endpoint of some interval i. List L_m is formed by scanning through *OPEN* and including all the intervals x that contain interval i and setting $w(L_m,x)$ to $CMIS[x]$.

Before giving the time analysis of our algorithm, we specify which intervals are in list L_q at any time during the scan. As before, let $re_u=q$. At the time L_q is created, every interval that contains u is in L_q and every such interval is also in *OPEN*. As the main scan moves to the right, L_q may contain intervals that are no longer open. Recall that such intervals are deleted from L_q when the local scan traverses L_q in either steps (1) or (2) of procedure CHECK_UPDATE. In general, list L_q contains a non-open interval x if no local scan initiated between positions $m-1$ and re_x-1 used interval u to update an interval open at the time of that local scan.

The $O(dn)$ total time of the algorithm is then established as follows. The update stage considers at most $2d$ positions in its right-to-left scan. For every interval updated at position q a constant amount of time is charged to the total time. No time is charged at position q for open intervals that were updated before the local scan reached position q. At most d intervals are updated in one update stage. Intervals no longer open are removed from list L_q (there can be up to $d-1$ intervals that are removed). One update stage can make $O(d)$ calls to procedure CHECK_UPDATE and can thus spend a total of $O(d^2)$ time on removing intervals. Nevertheless, the overall time spent on removing intervals from lists L_q can be at most $O(dn)$ since every list contains at most d intervals. Thus the $O(dn)$ time bound for computing the weight of a maximum independent set follows.

We now briefly describe how to modify the algorithm so as to generate also the intervals of a maximum independent set. We associate an initially empty linked list V_x with every

interval x, $1 \leq x \leq n$. We also add an array U of size n. At termination of the left-to-right scan, $U[x]$ is equal to the interval with the largest right endpoint in the maximum independent set formed by all intervals contained by x. Whenever interval x is updated and the value of $CMIS[x]$ increases, we proceed as follows: If the updating is done in step (2) of procedure CHECK_UPDATE (in which case $x=z$), we add the element consisting of the pair (i,u) to list V_x and we set $U[x]=i$. If interval x is updated outside CHECK_UPDATE (in which case no right endpoint updating x was found), we set $U[x]$ to i. Using the V lists and the array U, the intervals in a maximum independent set can easily be obtained in $O(dn)$ time. This concludes our discussion of the $O(dn)$ time and space algorithm for determining a maximum independent set of a circle graph given by its interval model.

Acknowledgements. The authors wish to thank the referees for their helpful and constructive comments and are grateful to one of the referees for finding a flaw in an earlier version of the paper.

References

[AALM] A. Apostolico, M.J. Atallah, L. Larmore, and H.S. McFaddin, 'Efficient Parallel Algorithms for String Editing and Related Problems', Purdue CS Tech Rept 724 (November 1987).

[AHU] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AgPa] A. Aggarwal and J. Park, 'Notes on Searching in Multidimensional Monotone Arrays', *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988, IEEE Computer Society, Washington, DC, pp. 497--512.

[Ap] A. Apostolico, 'Improving the Worst-Case Performance of the Hunt-Szymanski Strategy for the Longest Common Subsequence of Two Strings', *Information Processing Letters 23* , pp. 63-69, 1986.

[Bu] M. A. Buckingham, 'Efficient Stable Set and Clique Finding Algorithms for Overlap Graphs', Dept. of Computer Science, New York University, 1981

[Di] R.P. Dilworth, 'A Decomposition Theorem for Partially Ordered Sets', *Annals of Math.*, pp 161-165, 1950.

[EPL] S. Even, A. Pnueli and A. Lempel, 'Permutation Graphs and Transitive Graphs', *Journal of the ACM* 19, 3, pp. 400-410, 1972.

[Fr1] A. Frank, 'Some Polynomial Algorithms for Certain Graphs and Hypergraphs', *Proc. 5-th British Combinatorial Conference* , Aberdeen 1975, Congressus Numerantium, No XV, Utilitas Math., Winnipeg, 1976.

[Fr2] M.L. Fredman, 'On Computing the Length of Longest Increasing Subsequences', *Discrete Mathematics*, pp 29-35, 1975.

[Ga1] F. Gavril, 'Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph', *Networks*, pp 261-273, 1973.

[Ga2] F. Gavril, 'Algorithms on Circular-Arc Graphs', *Networks*, pp 357-369, 1974.

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

[GLL] U.I. Gupta, D.T. Lee, Y.-T. Leung, 'Efficient Algorithms for Interval Graphs and Circular Arc Graphs', *Networks*, pp 459-467, 1982.

[H] W.-L. Hsu, 'Maximum Weight Clique Algorithms for Circular-Arc Graphs and Circle Graphs', *SIAM J. on Computing*, pp 224-231, 1985.

[HS] J.W. Hunt, T.G. Szymanski, 'A Fast Algorithm for Computing Longest Common Subsequences', *Comm. of the ACM*, pp 350-353, 1977.

[Jo] D.B. Johnson, 'A priority Queue in which Initialization and Queue Operations Take $O(\log\log D)$ Time', *Math. Systems Theory* 15, pp. 295-309, 1982.

[MS] G.K. Manacher, C.J. Smith, 'Efficient Algorithms for New Problems on Interval Graphs', manuscript, 1985.

[PLE] A. Pnueli, A. Lempel and S. Even, 'Transitive Orientation of Graphs and Identification of Permutation Graphs', *Canadian Journal of Math.* 23, 1, pp.160-175, 1971.

[RU] D. Rotem, U. Urrutia, 'Finding Maximum Cliques in Circle Graphs', *Networks*, pp 269-278, 1981.

[vE] P. van Emde Boas, 'Preserving Order in a Forests in less than Logarithmic Time and Linear Space', *Inform. Processing Letters*, pp 80-82, 1977.

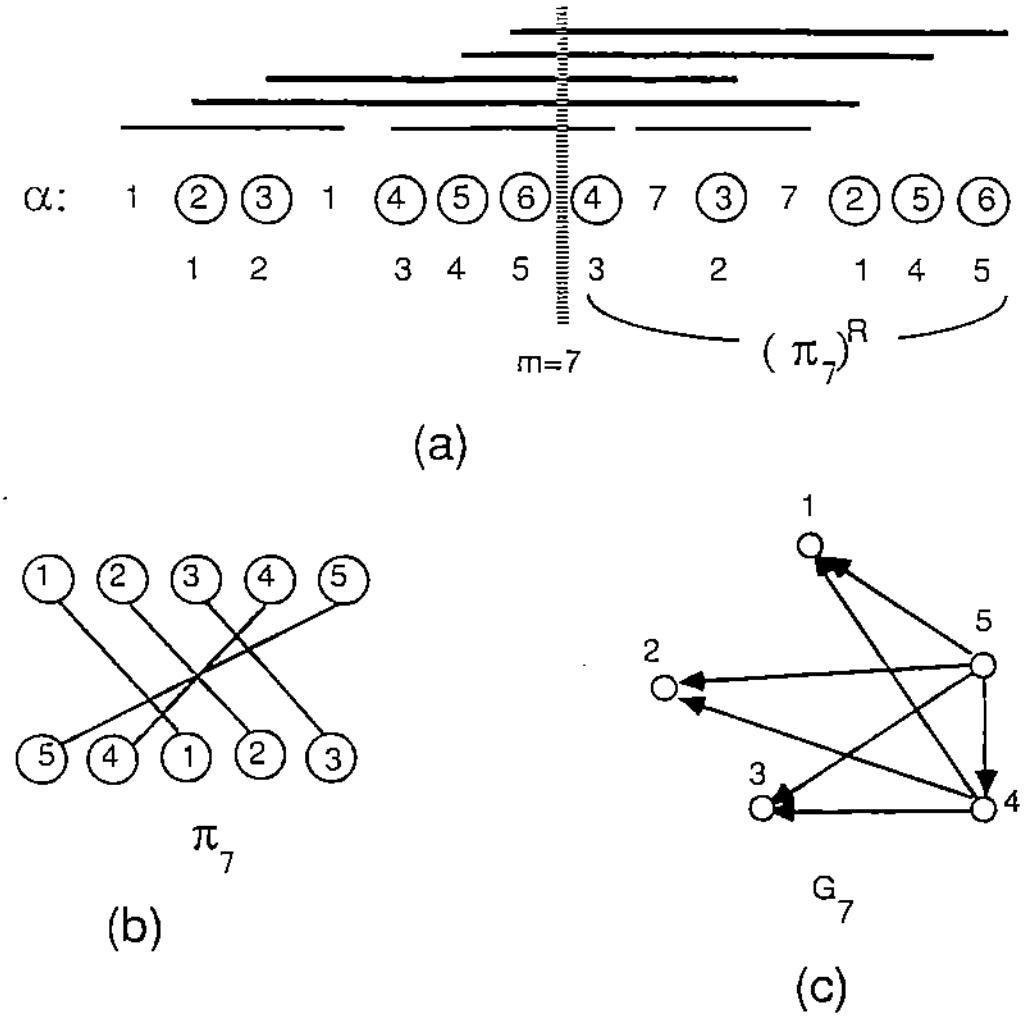


Figure 1

The interval model of a circle graph with its encoding α= 12314564737256 and the permutation graph induced by a crossing vertical line. Fig.1.a: Renumbering all the endpoints of intervals crossed by the vertical line and then reading the right renumbered endpoints from left to right yields the permutation π_7 associated with the decomposition $\alpha = \alpha^{(7)} \# \alpha_{(7)}$. Fig 1.b: The diagram of π_7. Fig. 1.c: Our transitive orientation of the permutation graph G_7 associated with π_7.

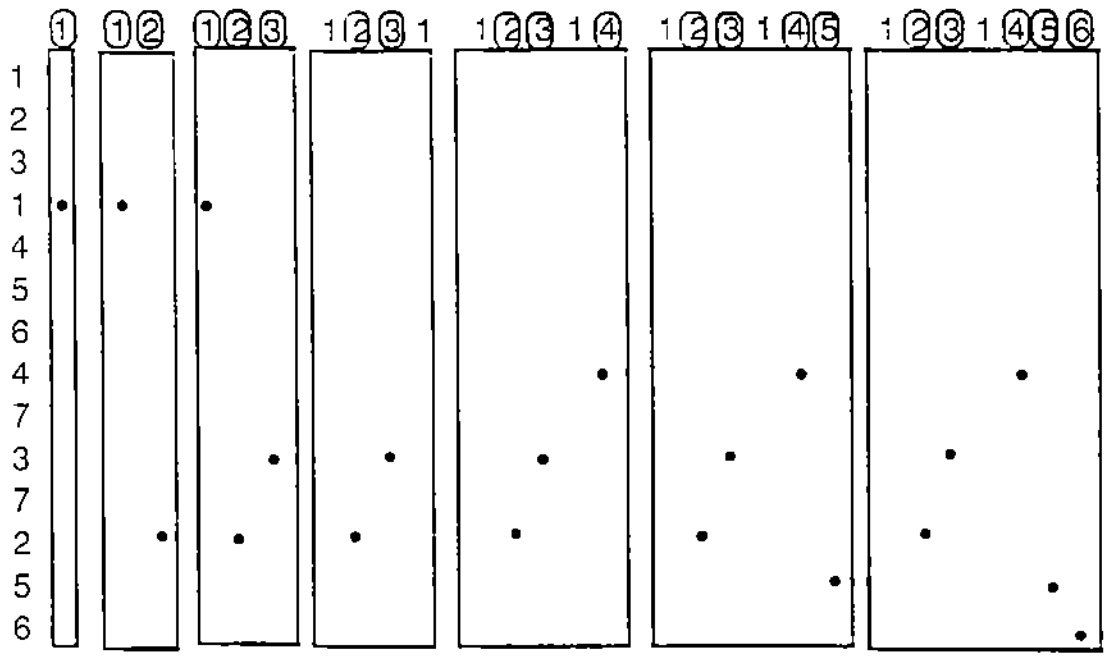


Figure 2

The sets of points for the first seven decompositions of α.

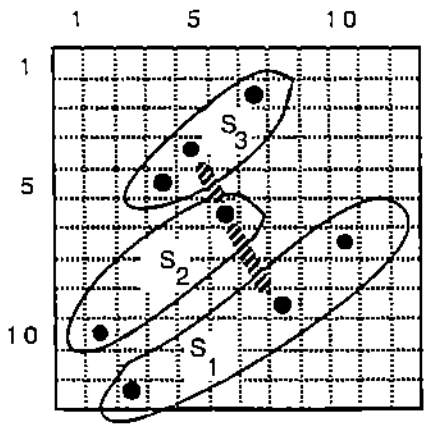


Figure 3

A minimal antichain decomposition for α . Points (9,8), (6,6) and (4,5) form a longest chain .

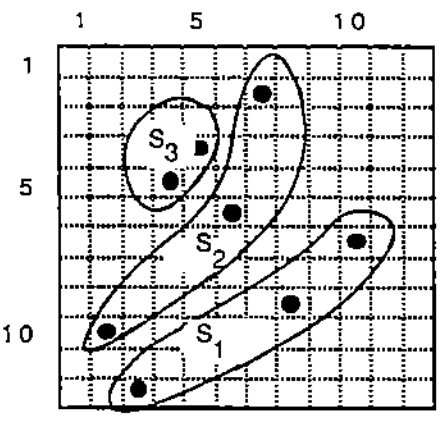


Figure 4

The canonical decomposition for Fig. 3.

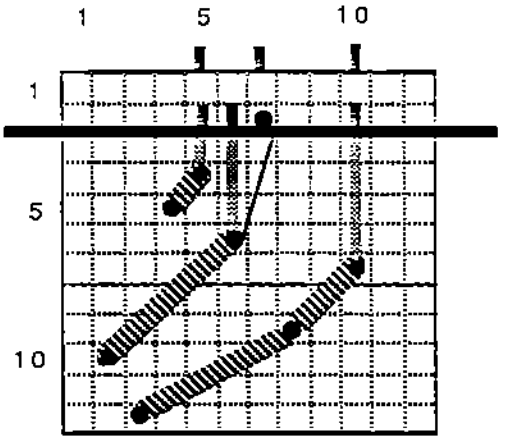


Figure 5

The last step in the construction of the canonical decomposition of Fig. 4. Prior to handling row 2, all points below this row have been assigned to antichains. At this point, the table T is (10,6,5) (these columns are marked by solid vertical bars above the horizontal line). Inserting point (2,7) in T causes the second entry to change from 6 to 7. The vertical solid bars on top of the array mark the new entries of T.

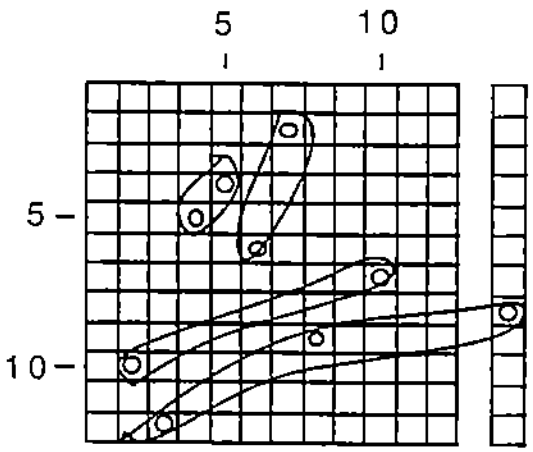


Figure 6

The canonical decomposition of set $P \cup \{(8,13)\}$.

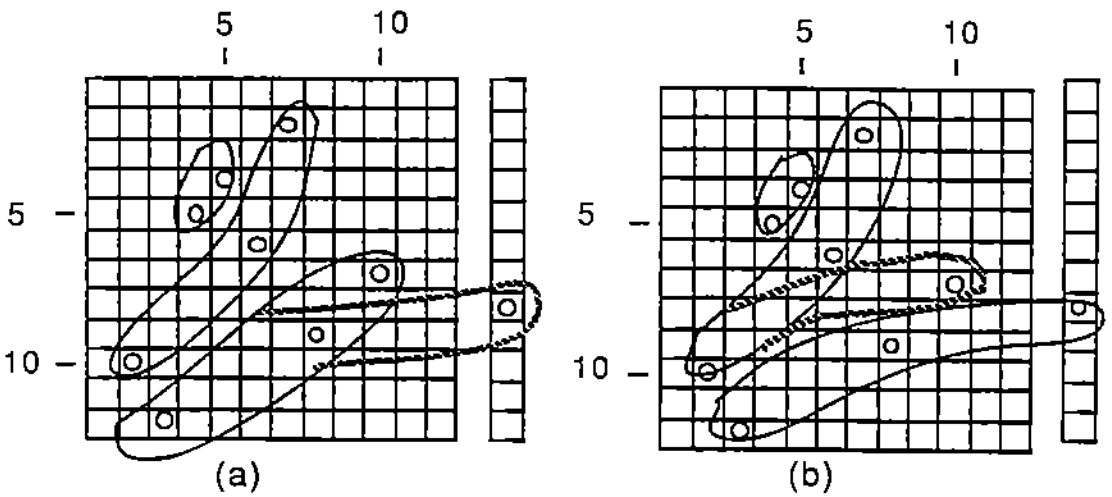


Figure 7

Fig. 7.a: The splitting effect of point (8,13) on antichain S_1. Fig. 7.b: The splitting effect of point (7,10), formerly a suffix of antichain S_1, on S_2. Point (6,6) will not split S_3.

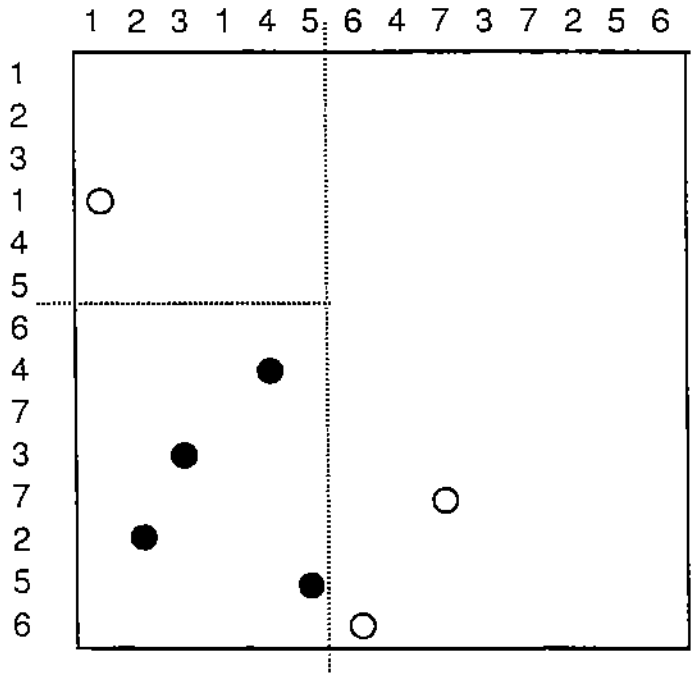


Figure 8

The trace array M^* associated with the set of intervals of Fig. 1. The portion of M^* left of the vertical broken line contains (properly) $M^{(5)}$. Note that all points in $M^{(5)}$ (shown solid) lie below the horizontal broken line. Only these points can be part of a chain starting at the point representing interval 5.

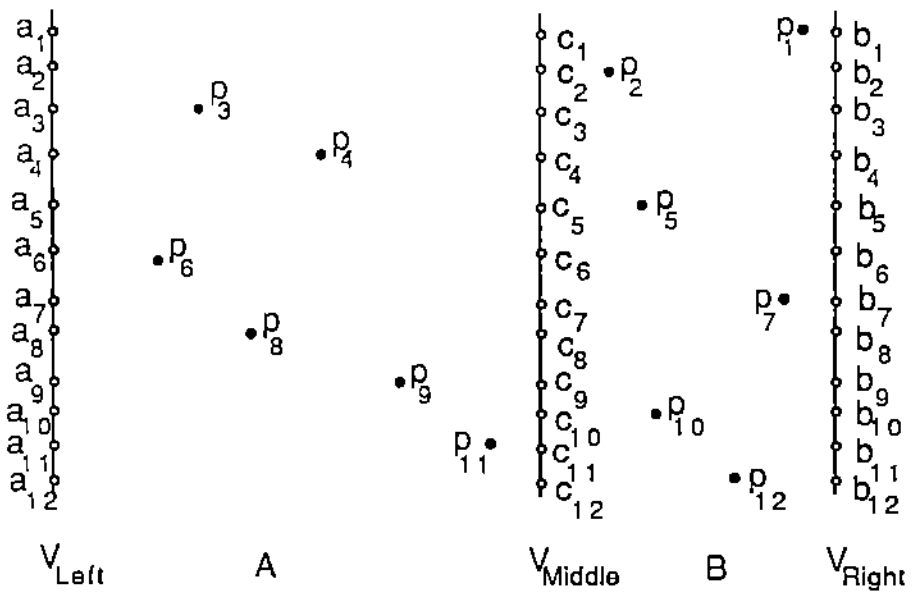


Figure 9

Illustrating the definitions for Lemma 2 and algorithm BESTCHAINS.

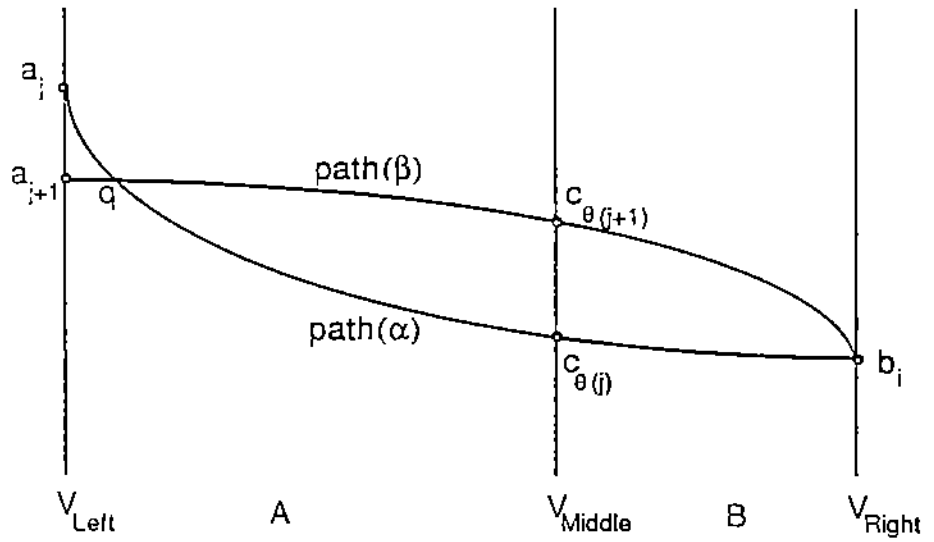
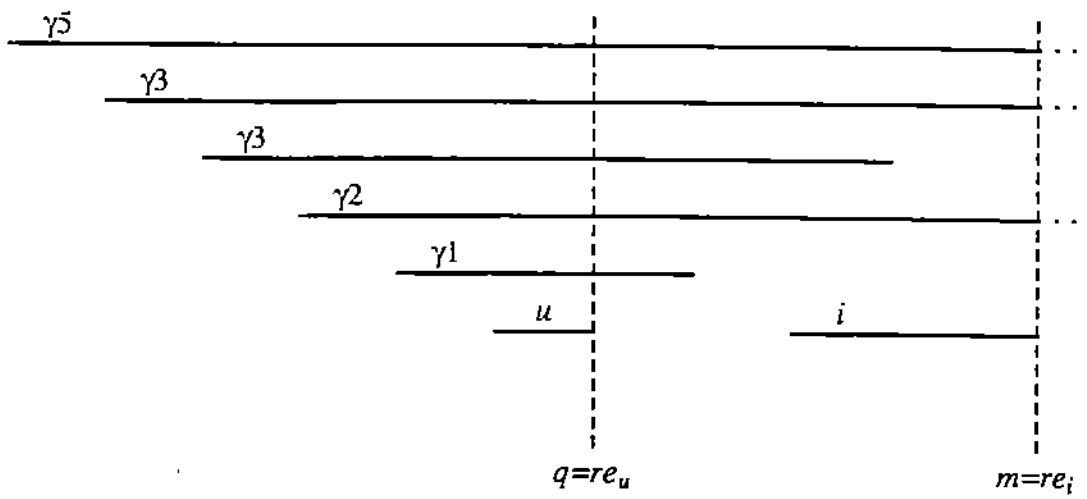


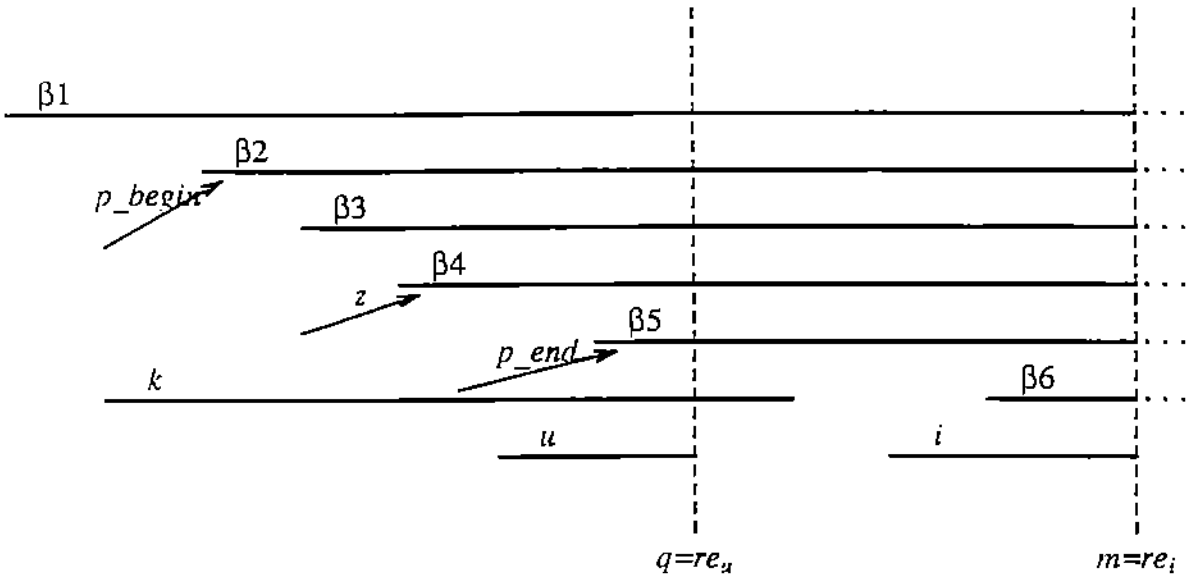
Figure 10

Illustrating the proof of †.



List L_q contains intervals γ1, · · · , γ5; γ1 and γ3 are no longer open.

Figure 11



Position of pointers *p_begin* and *p_end* and interval z after step (1) in CHECK_UPDATE when called for position *q*; interval *k* did update β1, interval *u* updates β4, β3, and β2.

Figure 12