

An Implicit Representation of Chordal Comparability Graphs in Linear-time

Andrew R. Curtis^a, Clemente Izurieta^b, Benson Joeris^c, Scott Lundberg^b,
Ross M. McConnell^{*,b}

^a*Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, N2L
3G1, Canada*

^b*Department of Computer Science, Colorado State University, Fort Collins, CO
80523-1873, USA*

^c*University of Cambridge, Centre for Mathematical Sciences, Cambridge, CB3 0WA, UK*

Abstract

Ma and Spinrad have shown that every transitive orientation of a chordal comparability graph is the intersection of four linear orders. That is, chordal comparability graphs are comparability graphs of posets of dimension four. Among other uses, this gives an implicit representation of a chordal comparability graph using $O(n)$ integers so that, given two vertices, it can be determined in $O(1)$ time whether they are adjacent, no matter how dense the graph is. We give a linear-time algorithm for finding the four linear orders, improving on their bound of $O(n^2)$.

Key words:

1. Introduction

A *partial order* or *poset* relation is a transitive antisymmetric relation. In this paper, we consider the graphical representation of a poset using a directed acyclic and transitive graph. When we say the graph is *transitive*, we mean that whenever (x, y) and (y, z) are directed edges, so is (x, z) . Whether the partial order is reflexive is irrelevant to our goals, so we only consider loopless graphs. The *comparability relation* of a partial order is the set of

*Corresponding author

Email addresses: `a2curtis@uwaterloo.ca` (Andrew R. Curtis), `blj24@cam.co.uk` (Benson Joeris), `rmm@cs.colostate.edu` (Ross M. McConnell)

pairs that are comparable in the partial order. That is, it is the symmetric closure, where, whenever (a, b) is in the partial order, (b, a) is added to it. The comparability relation has a natural representation as an undirected graph that has an edge ab whenever (a, b) and (b, a) are in the comparability relation; it is obtained by ignoring edge directions in the transitive graph that represents the partial order.

Given a comparability graph, it is possible to *transitively orient* it in linear time [1], that is, to recover a corresponding partial order. An earlier and simpler algorithm for this problem in the special case where the graph is chordal suffices here and is preferable [2].

A *chordal graph* is an undirected graph where each cycle of length four or greater has a *chord*, that is, an edge that is not on the cycle but whose endpoints are both on the cycle.

A co-comparability graph or co-chordal graph is one whose complement is a comparability graph or chordal graph, respectively. Many interesting graph classes are defined by intersecting the comparability, co-comparability, chordal and co-chordal graph classes.

An example is an *interval graph*, which is the intersection graph of a set of intervals on the line, that is, the graph that has one vertex for each of the intervals and an edge for each intersecting pair. These are exactly the intersection of the chordal and co-comparability graphs.

A *permutation graph* is defined by a permutation of a linearly ordered set of objects. The vertices are the objects, and the edges are the *non-inversions*, that is, the pairs of objects whose relative order is the same in the two permutations. These are exactly the intersection of the comparability and co-comparability graphs.

A *split graph* is a graph whose vertices can be partitioned into a clique and an independent set. These are exactly the intersection of the chordal and co-chordal graphs. More information about all graph classes mentioned here can be found in [3].

All of these graphs are subclasses of the class of perfect graphs, because comparability graphs and chordal graphs are perfect. Interval graphs can be represented with $O(n)$ integers, numbering the endpoints in left-to-right order and associating each vertex with its endpoint numbers. Adjacency can then be tested in $O(1)$ time by comparing the two pairs of endpoints of the vertices to see if they correspond to intersecting intervals. Similarly, permutation graphs can be represented by numbering the vertices in left-to-right order in two lists, and testing adjacency in $O(1)$ time by determining whether the two

vertices have the same relative order in both. These are examples of *implicit representations*; for more details see Spinrad’s book on the topic of implicit representations of graph classes [4], or the paper [5], where the topic was introduced.

A *linear order* is just a special case of a partial order, where the elements are numbered 1 through n , and the relation is the set of ordered pairs $\{(i, j) | i < j\}$. This partial order has $\Theta(n^2)$ elements, but can be represented implicitly by giving the ordering or numbering of the vertices. A *linear extension* of a partial order P is a linear order that has P as a subset. A *topological sort* of P is an ordering of elements that gives this implicit representation of a linear extension. It is any ordering where whenever $(i, j) \in P$, i is earlier in the ordering than j is.

It is easy to see that the intersection of two partial orders (the ordered pairs that are common to both) is also a partial order, hence this applies to the intersection of linear orders. In fact, every partial order is the intersection of a set of linear orders [6]. A partial order has *dimension* k if there exist k linear orders whose intersection is exactly that partial order. It is easy to see from this that the permutation graphs are just the comparability graphs of two-dimensional partial orders. Two-dimensional partial orders and permutation graphs can be recognized and their representation with two linear orders can be found in linear time [1]. In general, k linear orders gives an $O(nk)$ representation, but unfortunately, it is NP-complete to determine whether a partial order has dimension k for $k \geq 3$ [7].

In this paper, we examine *chordal comparability graphs*, that is, the intersection of the class of chordal graphs and the class of comparability graphs. Ma and Spinrad have shown that all chordal comparability graphs are the comparability graphs of partial orders of dimension at most four [8, 4]. The four linear orders give a way of representing the graph in $O(n)$ space so that for any two vertices, it can be answered in $O(1)$ time whether they are adjacent. Each vertex is labeled with the four position numbers of the vertex in the four linear order, and for two vertices, they are adjacent iff one of them precedes the other in each of the four orders. This type of implicit representation is desirable as a data structure for representing the partial order or its chordal comparability graph, and for organizing algorithmic solutions for combinatorial problems on the graphs.

This bound was shown to be tight by Kierstead, Trotter, and Qin in [9], who used a non-constructive Ramsey-theoretic proof to show that some chordal comparability graphs actually require four linear orders, but as is

typical of Ramsey-theoretic proofs, the known upper bound of the smallest one requiring four is an enormous $27^{27} + 1$ vertices. It seems likely that there exist small examples that require four linear orders. Testing a candidate is complicated by the NP-completeness of determining whether a partial order has dimension 3, though we do not know whether that problem remains NP-complete when restricted to chordal comparability graphs. Finding a smallest one, or even a small one, is an open problem.

We should note that, unlike the implicit representations of interval graphs and permutation graphs, this representation does not characterize chordal comparability graphs, as there are posets of dimension four whose comparability graphs are not chordal comparability graphs.

Ma and Spinrad have given a linear-time algorithm for recognizing chordal comparability graphs, but the best bound they give for finding the four linear orders is $O(n^2)$, where n is the number of vertices. In this paper, we improve this latter bound to $O(n + m)$.

The conference version of this paper appeared in [10].

2. Preliminaries

In this paper, G denotes a simple, finite graph with vertex-set $V(G)$ and edge-set $E(G)$. For convenience, we assume that G is connected. If there exists an edge between $v, u \in V(G)$, we say that v and u are adjacent or are neighbors in G . Let the neighbors of a vertex v be $N(v)$ and non-neighbors of v be $X(v)$. If G is directed, then we use (u, v) to denote an edge from u to v in G . Given a set of directed edges, X , denote by $G(X)$ the graph with vertex-set $V(G(X)) = \{v \mid v \rightarrow u \in X \text{ or } u \rightarrow v \in X\}$ and edge-set X . Given a directed graph G , we say that G is *acyclic* or is a *DAG* if G does not contain any directed cycles. The *transitive closure* of a DAG G adds the minimum number of edges to G such that the resulting graph is transitive.

If G is an interval graph, then there is a representation of G where each vertex v in G interval I_v whose endpoints are in $\{1, 2, \dots, 2n\}$. Two vertices v and u are adjacent if I_u and I_v share an element of $\{1, 2, \dots, 2n\}$. We say that v *contains* u if all points in I_u are also in I_v . Two intervals *overlap* if they share some point, but neither contains the other, and two intervals are *disjoint* if they share no common points. An interval I_v is *right* of another interval I_u if all points in I_v are less than all points in I_u ; in this case I_u is *left* of I_v .

2.1. Union-find Data Structure

Our algorithm primarily makes use of elementary data structures; however, we do use the union-find data structure. This data structure maintains a family of disjoint sets under the union operation. In order to identify the sets, each set has a *leader*, which is the representative for all elements in its set. Union-find supports the following operations.

- $\text{MakeSet}(x)$: creates a new set containing only x .
- $\text{Find}(x)$: returns the leader of the set containing x .
- $\text{Union}(x, y)$: unions the sets containing x and y and returns the single set's new leader.

A MakeSet operation on n elements, followed by m union and find operations on them, takes $O(n + m\alpha(m, n))$ time, where α is an extremely slow-growing but unbounded functional inverse of Ackermann's function. Full details can be found in [11].

However, there is a special case of the general union-find data structure developed by Gabow and Tarjan [12]. Their data structure requires initializing the structure with an unrooted tree on the n elements, and performing unions in any order that maintains the invariant that each union find class induces a connected subtree of the initializing tree. In our application, we are able to initialize the Gabow-Tarjan structure, and this is critical to obtaining a true linear time bound.

3. Representing a Chordal Comparability Graph with Four Linear Orders

Details of the following properties of chordal graphs are well-known, and can be found in the text by Golumbic [3]. A graph is chordal if and only if it has a *subtree intersection model*, which consists of the following:

1. A tree T that has $O(n)$ vertices;
2. A connected subtree T_v associated with each vertex v of G such that two vertices x and y are adjacent in G if and only if T_x and T_y contain a common node in T , and that the sum of cardinalities of the vertex sets in the subtrees is $O(m)$.

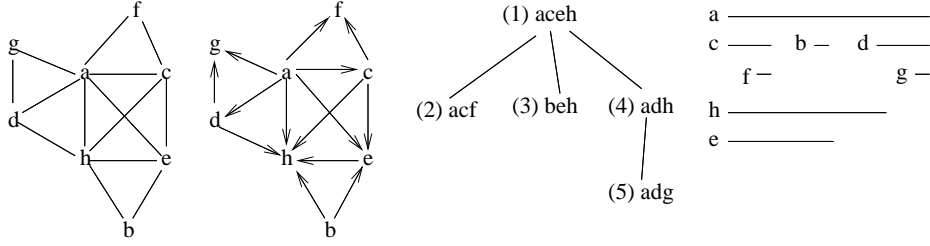


Figure 1: A chordal comparability graph G , a transitive orientation P , a clique tree of G , and, the intervals defined by a depth-first search of the clique tree. A vertex's interval is the time interval between the discovery of the first clique that contains it and the discovery of the last.

Such a tree is often called a *clique tree*. An example is given in Figure 1.

Following the approach of Ma and Spinrad, we perform an arbitrary depth-first search on the clique tree, labeling the vertices of the tree in ascending order of their discovery time. The first and last discovery time i and j of nodes in a subtree T_x defines an interval $I_x = [i, j]$ on the sequence $(1, 2, \dots, n)$. Because the sum of cardinalities of the cliques of a chordal graph is $O(n + m)$ [3], the time to find the intervals is $O(n + m)$.

It is easy to verify that x and y are adjacent in G if I_x and I_y properly overlap, and that they are nonadjacent if I_x and I_y are disjoint.

Suppose I_y is contained in I_x . Then it is possible that they are adjacent. If this were always the case, then G would be not just a chordal comparability graph, but an interval comparability graph. However, it is also possible that they are not adjacent. In this case, the DFS discovered a vertex in T_x , and sometime during the interval I_x , it left T_x to visit a set of vertices below T_x that contain T_y , before returning upward to T_x to finish traversing it. This shows that it is not necessary for T_x and T_y to intersect for I_y to be a subinterval of I_x . For example, in Figure 1, T_a consists of the subtree induced by nodes $\{1, 2, 4, 5\}$. The DFS leaves this tree to visit T_b , which is the subtree consisting of node 3, and then returns to T_a . The resulting intervals for a and b in the righthand figure intersect even though T_a and T_b do not.

We may find a transitive orientation P of the chordal comparability graph $G = (V, E)$ in linear time using the transitive orientation algorithm of [2]. Ma and Spinrad define three disjoint partial orders, any transitive orientation P of G , the set R_1 of ordered pairs of the form $\{(x, y) | I_x \text{ is left of } I_y\}$, and the set R_2 of ordered pairs of of the form $\{(x, y) | I_y \subset I_x \text{ and } x \text{ and } y$

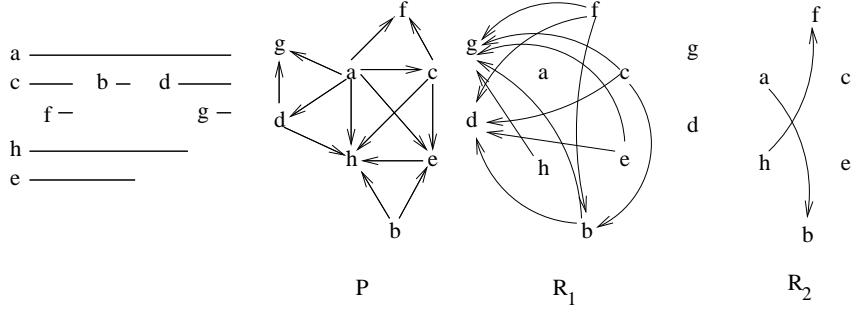


Figure 2: P is a transitive orientation of G , a pair (x, y) is in R_1 if x 's interval precedes y 's, and a pair (u, v) is in R_2 if v 's interval is contained in u' and uv is not an edge of G .

are non-neighbors}. Clearly, $P \cup R_1 \cup R_2$ is an orientation of the complete graph, $\{P, R_1, R_2\}$ is a partition of the edge set into three poset relations, and $\{R_1, R_2\}$ is a partition of an orientation of \overline{G} into two poset relations. We will refer to the ordered pairs in P , R_1 , and R_2 as *edges*.

In general, the union of two disjoint partial orders is not necessarily a partial order, or even acyclic. However, Ma and Spinrad show that $E_1 = P \cup R_1$, $E_2 = P \cup R_1^T$, where R_1^T denotes the reversal of all edges in R_1 , $E_3 = P \cup R_2$ and $E_4 = P \cup R_2^T$ are each acyclic. This shows that P is a four-dimensional partial order, as follows. Let L_1, L_2, L_3 , and L_4 be arbitrary topological sorts of E_1, E_2, E_3 , and E_4 , respectively. It must be the case that for $(u, v) \in P$, u precedes v in all four topological sorts, since it is a directed edge in each of E_1, E_2, E_3 , and E_4 . Every edge of P is *conserved* in the intersection of L_1 through L_4 .

For any edge (u, v) in R_1 , $(u, v) \in E_1$ and $(v, u) \in E_2$. Therefore u precedes v in L_1 and follows it in L_2 and (u, v) . It follows that neither (u, v) nor (v, u) is in the intersection $L_1 \cap L_2$ of the topological sorts of E_1 and E_2 . The act of reversing R_1 in E_1 and E_2 *deletes* the edges of R_1 from the intersection $L_1 \cap L_2$ hence from the intersection of L_1 through L_4 .

Similarly, for any edge (x, y) in R_2 , x precedes y in L_3 and follows it in L_4 , so the act of reversing R_2 in E_3 and E_4 ensures that x precedes y in L_3 and follows it in L_4 . Therefore, neither (x, y) nor (y, x) is in the intersection $L_3 \cap L_4$ of the topological sorts of E_3 and E_4 . Reversing R_2 in E_3 and E_4 *deletes* R_2 from the intersection.

Together, these observations prove that the intersection of L_1 through L_4 is exactly P : all elements of P are conserved and no elements of R_1, R_2, R_1^T , or R_2^T are conserved. The constructive proof gives the basis of Ma and Spin-

rad's algorithm, which finds a transitive orientation of P , finds a clique tree, performs a DFS on it to identify R_1 and R_2 , and then returns the topological sorts of E_1 through E_4 in $O(n^2)$ time.

On the surface, it seems impossible to improve on this time bound without resorting to an entirely different algorithm, since the topological sorts reference all edges in $P \cup R_1 \cup R_2$, and there are $n(n-1)/2$ of them.

Our approach is similar to Ma and Spinrad's, but we are able to use the properties of partial orders, chordal graphs, and a number of data structure tricks to avoid touching all of the edges in R_1 and R_2 directly, thereby obtaining an $O(n+m)$ bound.

3.1. Finding a topological sort of L_1 of $P \cup R_1$

In this section, we describe a procedure for finding the topological sort L_1 of $E_1 = P \cup R_1$. To obtain L_1 , we perform a depth-first search on E_1 , prepending each vertex to L_1 when DFS retreats from it because all of its neighbors have been marked visited. It is well-known that prepending vertices to a list as they finish during DFS results in a topological sort of any DAG [13]. The edges of P can be handled during the DFS in the standard way with an adjacency-list representation. However, when all neighbors in P of a vertex have been marked visited, it is not necessarily the case that all neighbors in E_1 have been marked visited, since it may have neighbors in R_1 . The problem is that $|R_1|$ is not $O(n+m)$, so touching all the members of R_1 would ruin our time bound. To get around this, we create a data structure that supports the following operation in $O(1)$ time:

- **Find next R_1 neighbor:** Given a vertex v , return an unmarked neighbor in R_1 if there is one, or else report that it has no unmarked neighbors in R_1 .

To create this data structure, we radix sort all endpoints of intervals using the position of the endpoint as primary sort key, and whether it is a left or right endpoint as a secondary sort key with left endpoints going earlier. We assign a pointer from each endpoint to its corresponding vertex; each vertex is pointed to by a left endpoint and a right endpoint. We then label each vertex v with a pointer to its *parent*, which is the vertex w with the first left endpoint that follows the right endpoint of v . The parent pointer is null if no such vertex exists. Finally, we create a list L of the vertices sorted by left endpoint, which can be obtained by listing the vertices in the order in which

pointers to them are found at left endpoints during a traversal of the list of endpoints.

When we run the DFS, we maintain a set of union-find classes on elements of L , using the following invariant:

- Initially all elements are unmarked, and elements are marked as union-find classes are merged. An element is marked when it is discovered during DFS, and can be marked either *discovered* or *finished*, depending on whether a recursive call on it is in progress or has returned. Each union-find class starts either at the beginning of the list or at the first element following an unmarked element, and contains all elements either up through the end of the list or through the next unmarked element. Each union-find class has at most one unmarked element, and is labeled with a pointer to its unmarked element, if it has one.

Initially, every element of L is in its own union-find class. At all times, every union-find class except the rightmost one has exactly one unmarked element, and the unmarked element in a class is the rightmost element in the class. In addition, every union-find class is consecutive in L , which allows us to use the path represented by L as the initializing tree for the Gabow-Tarjan data structure. The `find next R_1 neighbor` for a vertex v operation can be implemented by performing a find operation on the parent of v . As v 's parent is the first disjoint vertex right of v , its union-find class points to the first unmarked vertex right of v . Therefore, it takes $O(1)$ time to locate the next unmarked vertex right of v , namely, the next unmarked R_1 neighbor of v .

When a vertex v is visited, it is marked discovered. This requires the following:

- **Mark a vertex v as discovered:** Unless v is the last element of L , merge its union-find class and the union-find class that contains the successor w of v in L . Let the new class point to the unmarked element of w 's old class.

To perform the DFS, we make recursive calls on all neighbors of v in P . When the last of these returns, we can find an unmarked neighbor in R_1 in $O(1)$ time executing the `find next R_1 neighbor` operation. Either a recursive call is made on the result, marking it as discovered, or, if v has

no remaining neighbors in R_1 , v can be marked as finished. Each vertex is marked once as discovered and once as finished, so these marking operations can take place $O(n)$ times during the entire DFS, each at a cost of $O(1)$. Therefore, the inclusion of R_1 along with P in the DFS ends up costing $O(n)$ time, even though $|R_1|$ can be $\Theta(n^2)$ in the worst case. The final bound on the DFS to obtain L_1 is $O(n + m)$, where m is the number of edges in G .

By left-right symmetry, a similar algorithm applies to finding L_2 .

3.2. Finding a topological sort L_3 of $P \cup R_2$

The approach for finding L_3 is similar in spirit to the one for finding L_1 and L_2 , except that the charging argument to obtain the time bound is more complicated. We again handle DFS on P using an adjacency-list representation. When a vertex has no more unmarked neighbors in P , it may still have neighbors in R_2 . We must define an operation analogous to **find next R_1 neighbor**:

- **Find next R_2 neighbor:** Given a vertex v that has no undiscovered neighbor in P , return an undiscovered neighbor in R_2 if there is one, or else report that it has no unmarked neighbors in R_2 .

However, because of additional difficulties posed by R_2 edges, we use an amortizing argument that shows that all calls to **find next R_2 neighbor** made during a DFS take a total of $O(n)$ time.

As with **find next R_1 neighbor**, we create a data structure to support the operation by sorting vertices by left endpoint of their interval to obtain a list L , and we maintain union-find classes with one unmarked element in each class except the rightmost class.

All R_2 neighbors of v have both endpoints in the interior of I_v instead of to the right of I_v . Instead of making the parent be the first vertex whose interval lies strictly to the right of v 's interval, we make it be the first interval whose left endpoint is to the right of v 's left endpoint. We perform a **find** operation on this vertex to find the first unmarked vertex w that follows it in L . If I_w is to the right of I_v , then we may mark v as *finished*, since I_v has no unmarked vertices beginning in its interior. If $I_w \subset I_v$, then w is the next R_2 neighbor of v in the list, and we can make a recursive call on w , marking it as discovered.

A new problem arises when I_w properly overlaps I_v , since we have now spent $O(1)$ time finding w , but w 's interval is not contained in v 's, so it is

not an R_2 neighbor of v and we cannot mark it. Fortunately, if I_v and I_w properly overlap, they are neighbors in G . Since v has no unmarked neighbors in P and w is unmarked, it must be the case that (w, v) is an edge in P . We therefore charge the cost of touching w to the edge wv in G . We then continue by performing a `find` operation on the successor of w in L to find the next unmarked vertex. We iterate this operation, each time charging the $O(1)$ cost of finding the next unmarked vertex to an edge of G , halting when we reach the right endpoint of I_v , in which case v can be marked as finished, or else find an unmarked R_2 neighbor z , which we can then make a recursive call to DFS on, marking z discovered. During the recursive call, we retain a pointer to z so that when it returns, we may resume the DFS from v by performing a `find` operation on z .

Each vertex is again marked as discovered once, finished once, and each edge of P directed into a vertex v is charged once. Since $|P| = m$, The additional cost incurred in including R_2 with P in the DFS is $O(n + m)$.

3.3. Finding L_4

Unlike the case of L_1 and L_2 , the cases of L_3 and L_4 are not symmetric, so we cannot use the procedure for finding L_3 to find L_4 . In the case of L_3 , the `find next R_2 neighbor` found all unmarked vertices whose left endpoint was interior to I_v . Those that were not R_2 neighbors were neighbors in G , which allowed us to charge the cost of finding them to edges of G .

L_4 needs to be a topological sort of $P \cup R_2^T$. Consider what happens when we reverse R_2 to get R_2^T . The R_2^T neighbors of a vertex v are those non-neighbors w in G such that $I_v \subset I_w$. Such a neighbor has a left endpoint to the left of I_v 's left endpoint and a right endpoint to the right of I_v 's right endpoint. Using a Gabow-Tarjan data structure as we did above can identify unmarked neighbors whose left endpoint is to the left of I_v 's. The insurmountable problem is that such a vertex may also have a right endpoint to the left of I_v 's left endpoint. This means it is a non-neighbor in G . We have spent $O(1)$ time touching it, but we have no edge of G to charge the cost to.

We therefore abandon the union-find approach and instead adopt a strategy that involves partitioning sets into neighbors and non-neighbors in P , and takes advantage of the fact that we already have a topological sort L_3 of $P \cup R_2$.

We begin with L_3^T , which is a topological sort of $P^T \cup R_2^T$. We then modify this list to reverse the relative order of every pair (a, b) such that $(a, b) \in P$

without affecting the relative order of any pair $(c, d) \in R_2$. This yields an ordering L_4 such that all edges of P are in the intersection $L_3 \cap L_4$ and no edge of R_2 is in $L_3 \cap L_4$. $L_1 \cap L_2 \cap L_3 \cap L_4 = P$ will then be the four linear orders representing P , hence G .

Given a subset S of elements of a linear order L , let the *subsequence of L' induced by S* denote the result of deleting all elements from L' that are not in S . This is just the ordering of S that is consistent with their relative order in L .

Let us initially number the vertices in L_3^T in order from 1 to n in the order in which they appear in L_3^T . By radix sorting all edges of P using vertex of origin as the primary sort key and destination vertex as the secondary sort key, we may obtain, for each vertex v , an adjacency list that is sorted in left-to-right order as the vertices appear in L_3^T . This gives the subsequence L' of L_3^T defined by neighbors of v . We let L_3^T be a doubly-linked list, so that, given a pointer to a vertex in L_3^T , it can be removed from in $O(1)$ time.

We now give the algorithm for turning L_3^T into L_4 :

Reordering L_3^T to obtain L_4

Let v be the first vertex in L_3^T . As a base case, if $|L_3^T| \leq 1$, there is nothing to be done. Otherwise, remove the neighbors of v in P from L_3^T , leaving the subsequence L_n of non-neighbors of v in L_3^T . Since the adjacency list of v is in sorted left-to-right order in L_3^T , we can put them into a doubly-linked list that gives the subsequence L_a of vertices that are adjacent to v in L_3^T . We recursively reorder L_n and L_a to obtain L'_n and L'_a , and return the concatenation $L'_n \cdot v \cdot L'_a$.

The following establishes the correctness:

Lemma 1. $P \subseteq L_4$ and $R_2 \cap L_4 = \emptyset$.

Proof. In L_4 , all of its edges to elements L_a must point to the right. This is satisfied when L_a is moved to the right of v . Since P is transitive, L_a is not just the neighbors of v , but the set of all nodes reachable from v on a directed path. Therefore, there is no directed edge of P from L_a to L_n . There are no edges of P that go between v and L_n . All edges of P that go between these three sets are now directed to the right, as they are supposed to in L_4 .

Now we show that no edge of R_2 between these three sets points to the right. Suppose by contradiction that there is an edge $(x, v) \in R_2$ such that

$x \in L_n$. This would contradict the fact that v is the first element of L_3^T , which means it is a source in R_2^T , hence a sink in R_2 . Similarly, an edge (v, y) such that $y \in L_a$ is an edge of G , it is not an edge of R_2 .

Suppose by contradiction that (x, y) is an edge of R_2 such that $x \in L_n$ and $y \in L_a$. Then y is a neighbor of v in G and x is not. Moreover, $(x, y) \in R_2$ implies that $I_y \subset I_x$. Since v is a neighbor of y , I_v intersects I_y , which means I_v also intersects I_x . If I_v properly overlaps I_x , then v and x are neighbors in G , contradicting x 's membership in L_n . Therefore, $I_v \subset I_x$, and, since v and x are non-neighbors in G , this implies that $(v, x) \in R_2$. However, this contradicts the fact that v is a sink in R_2 .

We conclude that all edges of P that go between $L_n, \{v\}, L_a$ point to the right after the reordering, and all edges of R_2 that go between these sets point to the left.

It remains to show that after the recursive calls return, this is true of edges that are internal to L_n and L_a . None of the vertices internal to L_n or L_a have been reordered. They therefore satisfy the properties of L_3^T for the subgraph induced by the vertices passed to their recursive call and they are in doubly-linked lists, so the preconditions of the recursive calls have been met. By induction on the length of the subsequence passed into a recursive call, the recursive calls on L_n and L_a reorder these sets so that all edges of P internal to them point to the right in the final order, and all edges of R_2 internal to them point to the left in the final order. \square

Since L_3^T is a doubly-linked list, it takes time proportional to the degree of v to remove L_a out of L_3^T and concatenate it to the front. We charge this cost to edges of G out of v . At each level of the recursion, a distinct vertex serves in the role of v , so each edge of G is charged at most once. The running time is therefore $O(n + m)$.

Since $L_1 \cap L_2$ includes all of P and excludes every edge of R_1 , and since $L_3 \cap L_4$ includes all of P and excludes every edge of R_2 , we get the following:

Theorem 1. *Four linear orders that realize a chordal comparability graph G can be found in $O(n + m)$ time.*

References

- [1] R. M. McConnell, J. P. Spinrad, Modular decomposition and transitive orientation, *Discrete Mathematics* 201 (1-3) (1999) 189–241.

- [2] W. Hsu, T. Ma, Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs, *SIAM J. Comput.* 28 (1999) 1004–1020.
- [3] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
- [4] J. Spinrad, *Efficient Graph Representations*, American Mathematical Society, Providence RI, 2003.
- [5] S. Kannan, M. Naor, S. Rudich, Implicit representation of graphs, *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1988) 334–343.
- [6] B. Dushnik, E. W. Miller, Partially ordered sets, *Amer. J. Math.* 63 (1941) 600–610.
- [7] M. Yannakakis, The complexity of the partial order dimension problem, *SIAM J. Algebraic and Discrete Methods* 3 (1982) 303–322.
- [8] T. Ma, J. P. Spinrad, Cycle-free partial orders and chordal comparability graphs, *Order* 8 (1991) 49–61.
- [9] H. Kierstead, , W. T. Trotter, J. Qin, The dimension of cycle-free orders, *Order* 9 (1992) 103–110.
- [10] A. Curtis, C. Izurieta, B. Joeris, S. Lundberg, R. M. onnell, An implicit representation of chordal comparability graphs in linear time, *The 32nd International Conference on Graph-Theoretic Concepts in Computer Science LNCS4271* (2006) 168–178.
- [11] R. E. Tarjan, *Data structures and network algorithms*, Society for Industrial and Applied Math., Philadelphia, 1983.
- [12] H. N. Gabow, R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *Journal of Computer and System Sciences* 30 (1985) 209–221.
- [13] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, McGraw Hill, Boston, 2001.