A Fully Dynamic Algorithm for Recognizing and Representing Proper Interval Graphs

Pavol Hell[1], Ron Shamir[2], and Roded Sharan[2]

[1] School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada V5A1S6.
pavol@cs.sfu.ca.

[2] Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.
{shamir,roded}@math.tau.ac.il.

Abstract. In this paper we study the problem of recognizing and representing dynamically changing proper interval graphs. The input to the problem consists of a series of modifications to be performed on a graph, where a modification can be a deletion or an addition of a vertex or an edge. The objective is to maintain a representation of the graph as long as it remains a proper interval graph, and to detect when it ceases to be so. The representation should enable one to efficiently construct a realization of the graph by an inclusion-free family of intervals. This problem has important applications in physical mapping of DNA.

We give a near-optimal fully dynamic algorithm for this problem. It operates in time $O(\log n)$ per edge insertion or deletion. We prove a close lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation in the cell probe model with word-size b. We also construct optimal incremental and decremental algorithms for the problem, which handle each edge operation in $O(1)$ time.

1 Introduction

A graph G is called an *interval graph* if its vertices can be assigned to intervals on the real line so that two vertices are adjacent in G iff their intervals intersect. The set of intervals assigned to the vertices of G is called a *realization* of G. If the set of intervals can be chosen to be inclusion-free, then G is called a *proper interval graph*. Proper interval graphs have been studied extensively in the literature (cf. [7, 13]), and several linear time algorithms are known for their recognition and realization [2, 3].

This paper deals with the problem of recognizing and representing dynamically changing proper interval graphs. The input is a series of operations to be performed on a graph, where an operation is any of the following: Adding a vertex (along with the edges incident to it), deleting a vertex (and the edges incident to it), adding an edge and deleting an edge. The objective is to maintain a representation of the dynamic graph as long as it is a proper interval graph, and to detect when it ceases to be so. The representation should enable one to efficiently construct a realization of the graph. In the *incremental* version of the problem, only addition operations are permitted, i.e., the operations include only the addition of a vertex and the addition of an edge. In the *decremental* version of the problem only deletion operations are allowed.

The motivation for this problem comes from its application to *physical mapping* of DNA [1]. Physical mapping is the process of reconstructing the relative position of DNA fragments, called *clones*, along the target DNA molecule, prior to their sequencing, based on information about their pairwise overlaps. In some biological frameworks the set of clones is virtually inclusion-free - for example when all clones have a similar length (this is the case for instance for cosmid clones). In this case, the physical mapping problem can be modeled using proper interval graphs as follows. A graph G is built according to the biological data. Each clone is represented by a vertex and two vertices are adjacent iff their corresponding clones overlap. The physical mapping problem then translates to the problem of finding a realization of G, or determining that none exists.

Had the overlap information been accurate, the two problems would have been equivalent. However, some biological techniques may occasionally lead to an incorrect conclusion about whether two clones intersect, and additional experiments may change the status of an intersection between two clones. The resulting changes to the corresponding graph are the deletion of an edge, or the addition of an edge. The set of clones is also subject to changes, such as adding new clones or deleting 'bad' clones (such as chimerics [14]). These translate into addition or deletion of vertices in the corresponding graph. Therefore, we would like to be able to dynamically change our graph, so as to reflect the changes in the biological data, as long as they allow us to construct a map, i.e., as long as the graph remains a proper interval graph.

Several authors have studied the problem of dynamically recognizing and representing certain graph families. Hsu [10] has given an $O(m + n \log n)$-time incremental algorithm for recognizing interval graphs. (Throughout, we denote the number of vertices in the graph by n and the number of edges in it by m.) Deng, Hell and Huang [3] have given a linear-time incremental algorithm for recognizing and representing connected proper interval graphs This algorithm requires that the graph will remain connected throughout the modifications. In both algorithms [10, 3] only vertex increments are handled. Recently, Ibarra [11] found a fully dynamic algorithm for recognizing chordal graphs, which handles each edge operation in $O(n)$ time, or alternatively, an edge deletion in $O(n \log n)$ time and an edge insertion in $O(n/\log n)$ time.

Our results are as follows: For the general problem of recognizing and representing proper interval graphs we give a fully dynamic algorithm which handles each operation in time $O(d + \log n)$, where d denotes the number of edges involved in the operation. Thus, in case a vertex is added or deleted, d equals its degree, and in case an edge is added or deleted, $d = 1$. Our algorithm builds on the representation of proper interval graphs given in [3]. We also prove a lower bound for this problem of $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation in the cell probe model of computation with word-size b [16]. It follows that our algorithm is nearly optimal (up to a factor of $O(\log\log n)$).

For the incremental and the decremental versions of the problem we give optimal algorithms (up to a constant factor) which handle each operation in time $O(d)$. For the incremental problem this generalizes the result of [3] to arbitrary instances.

As a part of our general algorithm we give a fully dynamic procedure for maintaining connectivity in proper interval graphs. The procedure receives as input a sequence of operations each of which is a vertex addition or deletion, an edge addition or deletion, or a query whether two vertices are in the same connected component. It is assumed that the graph remains proper interval throughout the modifications, since otherwise our main algorithm detects that the graph is no longer a proper interval graph and halts. We show how to implement this procedure in $O(\log n)$ time per operation. In comparison, the best known algorithms for maintaining connectivity in general graphs require $O(\log^2 n)$ amortized time per operation [9], or $O(\sqrt{n})$ worst-case (deterministic) time per operation [4]. We also show that the lower bound of Fredman and Henzinger [5] of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation (in the cell probe model with word-size b) for maintaining connectivity in general graphs, applies to the problem of maintaining connectivity in proper interval graphs.

The paper is organized as follows: In section 2 we give the basic background and describe our representation of proper interval graphs and the realization it defines. In sections 3 and 4 we present the incremental algorithm. In section 5 we extend the incremental algorithm to a fully dynamic algorithm for proper interval graph recognition and representation. We also derive an optimal decremental algorithm. In section 6 we give a fully dynamic algorithm for maintaining connectivity in proper interval graphs. Finally, in section 7 we prove a lower bound on the amortized time per operation of a fully dynamic algorithm for recognizing proper interval graphs. For lack of space, some of the proofs and some of the algorithmic details are omitted.

2 Preliminaries

Let $G = (V, E)$ be a graph. We denote its set V of vertices also by $V(G)$ and its set E of edges also by $E(G)$. For a vertex $v \in V$ we define $N(v) := \{u \in V : (u, v) \in E\}$ and $N[v] := N(v) \cup \{v\}$. Let R be an equivalence relation on V defined by uRv iff $N[u] = N[v]$. Each equivalence class of R is called a *block* of G. Note that every block of G is a complete subgraph of G. The *size* of a block is the number of vertices in it. Two blocks A and B are *neighbors* in G if some (and hence all) vertices $a \in A, b \in B$, are adjacent in G. A *straight enumeration* of G is a linear ordering Φ of the blocks in G, such that for every block, the block and its neighboring blocks are consecutive in Φ.

Let $\Phi = B_1 < \ldots < B_l$ be an ordering of the blocks of G. For any $1 \leq i < j \leq l$, we say that B_i is ordered *to the left of* B_j, and that B_j is ordered *to the right of* B_i. A *chordless cycle* is an induced cycle of length greater than 3. A *claw* is an induced $K_{1,3}$. A graph is *claw-free* if it does not contain an induced claw. For basic definitions in graph theory see, e.g., [7].

The following are some useful facts about interval and proper interval graphs.

Theorem 1. *([12]) An interval graph contains no chordless cycle.*

Theorem 2. *([15]) A graph is a proper interval graph iff it is interval and claw-free.*

Theorem 3. *([3]) A graph is a proper interval graph iff it has a straight enumeration.*

Lemma 1 ("The umbrella property"). *Let Φ be a straight enumeration of a connected proper interval graph G. If A, B and C are blocks of G, such that $A < B < C$ in Φ and A is adjacent to C, then B is adjacent to A and to C (see figure 1).*

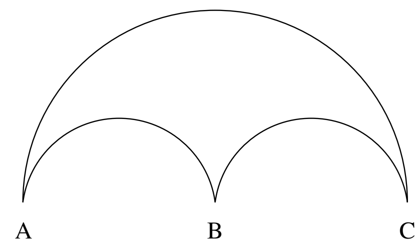

Fig. 1. The umbrella property

Let G be a connected proper interval graph and let Φ be a straight enumeration of G. It is shown in [3] that a connected proper interval graph has a unique straight enumeration up to its full reversal. Define the *out-degree* of a block B w.r.t. Φ, denoted by $o(B)$, as the number of neighbors of B which are ordered to its right in Φ.

We shall use the following representation: For each connected component of the dynamic graph we maintain a straight enumeration (in fact, for technical reasons we shall maintain both the enumeration and its reversal). The details of the data structure containing this information will be described below.

This information implicitly defines a realization of the dynamic graph (cf. [3]) as follows: Assign to each vertex in block B_i the interval $[i, i + o(B_i) + 1 - \frac{1}{i}]$. The out-degrees and hence the realization of the graph can be computed from our data structure in time $O(n)$.

3 An Incremental Algorithm for Vertex Addition

In the following two sections we describe an optimal incremental algorithm for recognizing and representing proper interval graphs. The algorithm receives as input a series of addition operations to be performed on a graph. Upon each operation the algorithm updates its representation of the graph and halts if the current graph is no longer a proper interval graph. The algorithm handles each operation in time $O(d)$, where d denotes the number of edges involved in the operation. It is assumed that initially the graph is empty, or alternatively, that the representation of the initial graph is known.

A *contig* of a connected proper interval graph G is a straight enumeration of G. The first and the last blocks of a contig are called *end-blocks*. The rest of the blocks are called *inner-blocks*.

As mentioned above, each component of the dynamic graph has exactly two contigs (which are full reversals of each other) and both are maintained by the algorithm. Each operation involves updating the representation. (In the sequel we concentrate on describing only one of the two contigs for each component. The second contig is updated in a similar way.)

3.1 The Data Structure

The following data is kept and updated by the algorithm:

1. For each vertex we keep the name of the block to which it belongs.
2. For each block we keep the following:
   (a) An *end* pointer which is null if the block is not an end-block of its contig, and otherwise points to the other end-block of that contig.
   (b) The *size* of the block.
   (c) Left and right *near* pointers, pointing to nearest neighbor blocks on the left and on the right respectively.
   (d) Left and right *far* pointers, pointing to farthest neighbor blocks on the left and on the right respectively.
   (e) Left and right *self* pointers, pointing to the block.
   (f) A counter.

In the following we shall omit details about the obvious updates to the name of the block of a vertex and to the size of a block.

During the execution of the algorithm we may need to update many far pointers pointing to a certain block, so that they point to another block. In order to be able to do that in $O(1)$ time we use the technique of *nested pointers*: We make the far pointers point to a *location* whose content is the address of the block to which the far pointers should point. The role of this special location will be served by our self-pointers. The value of the left and right self-pointers of B is always the address of B. When we say that a certain left (right) far pointer points to B, we mean that it points to a left (right) self-pointer of B. Let A and B be blocks. In order to change all left (right) far pointers pointing to A so that they point to B, we require that no left (right) far pointer points to B. If this is the case, we simply *exchange* the left (right) self-pointer of A with the left (right) self-pointer of B. This means that: (1) The previous left (right) self-pointer of A is made to point to B, and the algorithm records it as the new left (right) self-pointer of B; (2) The previous left (right) self-pointer of B is made to point to A, and the algorithm records it as the new left (right) self-pointer of A.

We shall use the following notation: For a block B we denote its address in the memory by $\&B$. When we set a far pointer to point to a left or to a right self-pointer of B we will abbreviate and set it to $\&B$. We denote the left and right near pointers of B by $N_l(B)$ and $N_r(B)$ respectively. We denote the left and right far pointers of B by $F_l(B)$ and $F_r(B)$ respectively. We denote its end pointer by $E(B)$. In the sequel we often refer to blocks by their addresses. For example, if A and B are blocks, and $N_r(A) = \&B$, we sometimes refer to B by $N_r(A)$. When it is clear from the context, we also use a name of a block to denote any vertex in that block. Given a contig Φ we denote its reversal by Φ^R. In general when performing an operation, we denote the graph before the operation is carried out by G, and the graph after the operation is carried out by G'.

3.2 The Impact of a New Vertex

In the following we describe the changes made to the representation of the graph in case G' is formed from G by the addition of a new vertex v of degree d. We also give some necessary and some sufficient conditions for deciding whether G' is proper interval.

Let B be a block of G. We say that v is *adjacent* to B if v is adjacent to some vertex in B. We say that v is *fully adjacent* to B if v is adjacent to *every* vertex in B. We say that v is *partially adjacent* to B if v is adjacent to B but not fully adjacent to B.

The following lemmas characterize, assuming that G' is proper interval, the adjacencies of the new vertex.

Lemma 2. *If G' is a proper interval graph then v can have neighbors in at most two connected components of G.*

Lemma 3. *[3] Let C be a connected component of G containing neighbors of v. Let $B_1 < \ldots < B_k$ be a contig of C. Assume that G' is proper interval and let $1 \leq a < b < c \leq k$. Then the following properties are satisfied:*

1. *If v is adjacent to B_a and to B_c, then v is fully adjacent to B_b.*
2. *If v is adjacent to B_b and not fully adjacent to B_a and to B_c, then B_a is not adjacent to B_c.*
3. *If $b = a + 1, c = b + 1$ and v is adjacent to B_b, then v is fully adjacent to B_a or to B_c.*

One can view a contig Φ of a connected proper interval graph C as a weak linear order $<_\Phi$ on the vertices of C, where $x <_\Phi y$ iff the block containing x is ordered in Φ to the left of the block containing y. We say that Φ' is a *refinement* of Φ if for every $x, y \in V(C)$, $x <_\Phi y$ implies $x <_{\Phi'} y$ (since a contig can be reversed, we also allow complete reversal of Φ).

Lemma 4. *If G is a connected induced subgraph of a proper interval graph G', Φ is a contig of G and Φ' is a straight enumeration of G', then Φ' is a refinement of Φ.*

Note, that whenever v is partially adjacent to a block B in G, then the addition of v will cause B to split into two blocks of G', namely $B \setminus N(v)$ and $B \cap N(v)$. Otherwise, if B is a block of G to which v is either fully adjacent or not adjacent, then B is also a block of G'.

Corollary 1. *If B is a block of G to which v is partially adjacent, then $B \setminus N(v)$ and $B \cap N(v)$ occur consecutively in a straight enumeration of G'.*

Lemma 5. *Let C be a connected component of G containing neighbors of v. Let the set of blocks in C which are adjacent to v be $\{B_1, \ldots, B_k\}$. Assume that in a contig of C, $B_1 < \ldots < B_k$. If G' is proper interval then the following properties are satisfied:*

1. *$B_1, \ldots, B_k$ are consecutive in C.*
2. *If $k \geq 3$ then v is fully adjacent to $B_2, \ldots, B_{k-1}$.*
3. *If v is adjacent to a single block B_1 in C, then B_1 is an end-block.*
4. *If v is adjacent to more than one block in C and has neighbors in another component, then B_1 is adjacent to B_k, and one of B_1 or B_k is an end-block to which v is fully adjacent, while the other is an inner-block.*

Proof. Claims 1 and 2 follow directly from part 1 of Lemma 3. Claim 3 follows from part 3 of Lemma 3. To prove the last part of the lemma let us denote the other component containing neighbors of v by D. Examine the induced connected subgraph H of G whose set of vertices is $V(H) = \{v\} \cup V(C) \cup V(D)$. H is proper interval as an induced subgraph of G. It is composed of three types of blocks: Blocks whose vertices are from $V(C)$, which we will call henceforth C-blocks; blocks whose vertices are from $V(D)$, which we will call henceforth D-blocks; and $\{v\}$ which is a block of H since $H \setminus \{v\}$ is not connected. All blocks of C remain intact in H, except B_1 and B_k which might split into $B_j \setminus N(v)$ and $B_j \cap N(v)$, for $j = 1, k$.

Surely in a contig of H, C-blocks must be ordered completely before or completely after D-blocks. Let Φ denote a contig of H, in which C-blocks are ordered before D-blocks. Let X denote the rightmost C-block in Φ. By the umbrella property, $X < \{v\}$ and moreover, X is adjacent to v. By Lemma 4, Φ is a refinement of a contig of C. Hence, $X \subseteq B_1$ or $X \subseteq B_k$ (more precisely, $X = B_1 \cap N(v)$ or $X = B_k \cap N(v)$). Therefore, one of B_1 or B_k is an end-block.

W.l.o.g. $X \subseteq B_k$. Suppose to the contrary that v is not fully adjacent to B_k. Then by Lemma 4 we have $B_{k-1} \cap N(v) < B_k \setminus N(v) < \{v\}$ in Φ, contradicting the umbrella property. B_1 must be adjacent to B_k, or else G' contains a claw consisting of v, B_1, B_k and a vertex from $V(D) \cap N(v)$. It remains to show that B_1 is an inner-block. Suppose it is an end block. Since B_1 and B_k are adjacent, C contains a single block B_1, a contradiction. Thus, claim 4 is proved.■

3.3 The Algorithm

In our algorithm we rely on the incremental algorithm of Deng, Hell and Huang [3], which we call henceforth the *DHH algorithm*. This algorithm handles the insertion of a new vertex into a graph in $O(d)$ time, provided that all its neighbors are in the same connected component, changing the straight enumeration of this component appropriately. We refer the reader to [3] for more details.

We perform the following upon a request for adding a new vertex v. For each neighbor u of v we add one to the count of the block containing u. We call a block *full* if its counter equals its size, *empty* if its counter equals zero, and *partial* otherwise. In order to find a set of consecutive blocks which contain neighbors of v, we pick arbitrarily a neighbor of v and march down the enumeration of blocks to the left using the left near neighbor pointers. We continue till we hit an empty block or till we reach the end of the contig. We do the same to the right and this way we discover a maximal sequence of nonempty blocks in that component which contain neighbors of v. We call this maximal sequence a *segment*. Only the two extreme blocks of the segment are allowed to be partial or else we fail (by Lemma 5(2)).

If the segment we found contains all neighbors of v then we can use the DHH algorithm in order to insert v into G, updating our internal data structure accordingly. Otherwise, by Lemmas 2 and 5(1) there could be only one more segment which contains neighbors of v. In that case, exactly one extreme block in each segment is an end-block to which v is fully adjacent (if the segment contains more than one block), and the two extreme blocks in each segment are adjacent, or else we fail (by Lemma 5(3,4)).

We proceed as above to find a second segment containing neighbors of v. We can make sure that the two segments are from two different contigs by checking that their end-blocks do not point to each other. We also check that conditions 3 and 4 in Lemma 5 are satisfied. If the two segments do not cover all neighbors of v, we fail.

If v is adjacent to vertices in two distinct components C and D, then we should merge their contigs. Let $\Phi = B_1 < \ldots < B_k$, Φ^R be the two contigs of C. Let

$\Psi = B'_1 < \ldots < B'_l$, Ψ^R be the two contigs of D. The way the merge is performed depends on the blocks to which v is adjacent. If v is adjacent to B_k and to B'_1, then by the umbrella property the two new contigs (up to refinements described below) are $\Phi < \{v\} < \Psi$ and $\Psi^R < \{v\} < \Phi^R$. In the following we describe the necessary changes to our data structure in case these are the new contigs. The three other cases are handled similarly.

- Block enumeration: We merge the two enumerations of blocks and put a new block $\{v\}$ in-between the two contigs. Let the leftmost block adjacent to v in the new ordering $\Phi < \{v\} < \Psi$ be B_i and let the rightmost block adjacent to v be B'_j. If B_i is partial we split it into two blocks $\hat{B}_i = B_i \setminus N(v)$ and $B_i = B_i \cap N(v)$ in this order. If B'_j is partial we split it into two blocks $B'_j = B'_j \cap N(v)$ and $\hat{B}'_j = B'_j \setminus N(v)$ in this order.
- End pointers: We set $E(B_1) = E(B'_1)$ and $E(B'_l) = E(B_k)$. We then nullify the end pointers of B_k and B'_1.
- Near pointers: We update $N_l(\{v\}) = \&B_k, N_r(\{v\}) = \&B'_1, N_r(B_k) = \&\{v\}$ and $N_l(B'_1) = \&\{v\}$. Let $B_0 = \emptyset$. In case B_i was split we update $N_r(\hat{B}_i) = \&B_i, N_l(B_i) = \&\hat{B}_i, N_l(\hat{B}_i) = \&B_{i-1}$ and $N_r(B_{i-1}) = \&\hat{B}_i$. Similar updates are made in case B'_j was split to the near pointers of $B'_j, \hat{B}'_j$ and B'_{j+1}.
- Far pointers: If B_i was split we set $F_l(\hat{B}_i) = F_l(B_i), F_r(\hat{B}_i) = \&B_k$ and exchange the left self-pointer of B_i with the left self-pointer of $\hat{B}_i$. If B'_j was split we set $F_r(\hat{B}'_j) = F_r(B'_j), F_l(\hat{B}'_j) = \&B'_1$ and exchange the right self-pointer of B'_j with the right self-pointer of $\hat{B}'_j$. In addition, we set all right far pointers of $B_i, B_{i+1} \ldots, B_k$ and all left far pointers of $B'_1, \ldots, B'_{j-1}, B'_j$ to $\&\{v\}$ (in $O(d)$ time). Finally, we set $F_l(\{v\}) = \&B_i$ and $F_r(\{v\}) = \&B'_j$.

4 An Incremental Algorithm for Edge Addition

In this section we show how to handle the addition of a new edge (u, v) in $O(1)$ time. We characterize the cases for which $G' = G \cup \{(u, v)\}$ is proper interval and show how to efficiently detect them, and how to update our representation of the graph.

Lemma 6. *If u and v are in distinct components in G, then G' is proper interval iff u and v were in end-blocks of their respective contigs.*

Proof. To prove the 'only if' part let us examine the graph $H = G' \setminus \{u\} = G \setminus \{u\}$. H is proper interval as an induced subgraph of G. If G' is proper interval, then by Lemma 5(3) v must be in an end-block of its contig, since u is not adjacent to any other vertex in the component containing v. The same argument applies to u.

To prove the 'if' part we give a straight enumeration of the new connected component containing u and v in G'. Denote by C and D the components containing u and v respectively. Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_k$. Let $B'_1 < \ldots < B'_l$ be a contig of D, such that $v \in B'_1$. Then $B_1 < \ldots < B_k \setminus \{u\} < \{u\} < \{v\} < B'_1 \setminus \{v\} < \ldots < B'_l$ is a straight enumeration of the new component.■

We can check in $O(1)$ time if u and v are in end-blocks of distinct contigs. If this is the case, we update our data structure according to the straight enumeration given in the proof of Lemma 6 in $O(1)$ time.

It remains to handle the case where u and v were in the same connected component C in G. If $N(u) = N(v)$ then by the umbrella property it follows that C contains only three blocks which are merged into a single block in G'. In this case G' is proper interval and updates to the internal data structure are trivial. The following lemma analyses the case where $N(u) \neq N(v)$.

Lemma 7. *Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_i$ and $v \in B_j$ for some $1 \leq i < j \leq k$. Assume that $N(u) \neq N(v)$. Then G' is proper interval iff $F_r(B_i) = B_{j-1}$ and $F_l(B_j) = B_{i+1}$ in G.*

Proof. To prove the 'only if' part assume that G' is proper interval. Since B_i and B_j are not adjacent, $F_r(B_i) \leq B_{j-1}$ and $F_l(B_j) \geq B_{i+1}$. Suppose to the contrary that $F_r(B_i) < B_{j-1}$. Let $z \in B_{j-1}$. If in addition $F_l(B_j) = B_{i+1}$ then $N[v] \supset N[z]$ (this is a strict containment). As v and z are in distinct blocks, there exists a vertex $b \in N[v] \setminus N[z]$. But then, v, b, z, u induce a claw in G', a contradiction. Hence, $F_l(B_j) > B_{i+1}$ and so $F_r(B_{i+1}) < B_j$. Let $x \in B_{i+1}$ and let $y \in F_r(B_{i+1})$. Since u and x are in distinct blocks, either $(u, y) \notin E(G)$ or there is a vertex $a \in N[u] \setminus N[x]$ (or both). In the first case, v, u, x, y and the vertices of the shortest path from y to v induce a chordless cycle in G'. In the second case u, a, x, v induce a claw in G'. Hence, in both cases we arrive at a contradiction. The proof that $F_l(B_j) = B_{i+1}$ is symmetric.

To prove the 'if' part we shall provide a straight enumeration of $C \cup \{u, v\}$. If $B_i = \{u\}$, $F_r(B_{j-1}) = F_r(B_j)$ and $F_l(B_{j-1}) = B_i$ (i.e., $N[v] = N[B_{j-1}]$ in G'), we move v from B_j to B_{j-1}. Similarly, if B_j contained only v, $F_l(B_{i+1}) = F_l(B_i)$ and $F_r(B_{i+1}) = B_j$ (i.e., $N[u] = N[B_{i+1}]$ in G'), we move u from B_i to B_{i+1}. If u was not moved and $B_i \supset \{u\}$, we split B_i into $B_i \setminus \{u\}, \{u\}$ in this order. If v was not moved and $B_j \supset \{v\}$, we split B_j into $\{v\}, B_j \setminus \{v\}$ in this order. It is easy to see that the result is a straight enumeration of $C \cup \{u, v\}$.■

We can check in $O(1)$ time if the condition in Lemma 7 holds. If this is the case, we change our data structure so as to reflect the new straight enumeration given in the proof of Lemma 7. This can be done in $O(1)$ time, in a similar fashion to the update technique described in Section 3.3. The details are omitted here. The following theorem summarizes the results of Sections 3 and 4.

Theorem 4. *The incremental proper interval graph representation problem is solvable in $O(1)$ time per added edge.*

5 The Fully Dynamic Algorithm

In this section we give a fully dynamic algorithm for recognizing and representing proper interval graphs. The algorithm performs each operation in $O(d + \log n)$ time, where d denotes the number of edges involved in the operation. It supports four types of operations: Adding a vertex, adding an edge, deleting a vertex and deleting an edge.

It is based on the same ideas used in the incremental algorithm. The main difficulty in extending the incremental algorithm to handle all types of operations, is updating the end pointers of blocks when deletions are allowed. To bypass this problem we do not keep end pointers at all. Instead, we maintain the connected components of G, and use this information in our algorithm. In the next section we show how to maintain the connected components of G in $O(\log n)$ time per operation. We describe below how each operation is handled by the algorithm.

5.1 The Addition of a Vertex or an Edge

These operations are handled in essentially the same way as done by the incremental algorithm. However, in order to check if the end-blocks of two segments are in distinct components, we query our data structure of connected components (in $O(\log n)$ time). Similarly, in order to check if the endpoints of an added edge are in distinct components, we check if their corresponding blocks are in distinct components (in $O(\log n)$ time).

5.2 The Deletion of a Vertex

We show next how to update the contigs of G after deleting a vertex v of degree d. Note that G' is proper interval as an induced subgraph of G. Denote by X the block containing v. If $X \supset \{v\}$, then the only change needed is to delete v. We hence concentrate on the case that $X = \{v\}$. We can find in $O(d)$ time the segment of blocks which includes X and all its neighbors. Let the contig containing X be $B_1 < \ldots < B_k$ and let the blocks of the segment be $B_i < \ldots < B_j$, where $X = B_l$ for some $1 \leq i \leq l \leq j \leq k$. Let $B_0 = \emptyset, B_{k+1} = \emptyset$. We make the following updates:

- Block enumeration: If $1 < i < l$, we check whether B_i can be merged with B_{i-1}. If $F_l(B_i) = F_l(B_{i-1}), F_r(B_i) = B_l$ and $F_r(B_{i-1}) = B_{l-1}$, we merge them by moving all vertices from B_i to B_{i-1} (in $O(d)$ time) and deleting B_i. If $l < j < k$ we act similarly w.r.t. B_j and B_{j+1}. Finally, we delete B_l. If $1 < l < k$ and B_{l-1}, B_{l+1} are non-adjacent, then by the umbrella property they are no longer in the same connected component, and the contig should be split into two contigs, one ending at B_{l-1} and one beginning at B_{l+1}.
- Near pointers: If B_i and B_{i-1} were merged, we update $N_r(B_{i-1}) = \&B_{i+1}$ and $N_l(B_{i+1}) = \&B_{i-1}$. Similar updates should be made w.r.t. B_{j-1} and B_{j+1} in case B_j and B_{j+1} were merged. If the contig is split, we nullify $N_r(B_{l-1})$ and $N_l(B_{l+1})$. Otherwise, we update $N_r(B_{l-1}) = \&B_{l+1}$ and $N_l(B_{l+1}) = \&B_{l-1}$.
- Far pointers: If B_i and B_{i-1} were merged, we exchange the right self-pointer of B_i with the right self-pointer of B_{i-1}. Similar changes should be made w.r.t. B_j and B_{j+1}. We also set all right far pointers previously pointing to B_l, to $\&B_{l-1}$; and all left far pointers previously pointing to B_l, to $\&B_{l+1}$ (in $O(d)$ time).

Note that these updates take $O(d)$ time and require no knowledge about the connected components of G.

5.3 The Deletion of an Edge

Let (u, v) be an edge of G to be deleted. Let C denote the connected component of G containing u and v, and let $B_1 < \ldots < B_k$ be a contig of C. If $k = 1$ then B_1 is split into $\{u\}, B_1 \setminus \{u, v\}$ and $\{v\}$, resulting in a straight enumeration of G'. Updates are trivial in this case. If $N[u] = N[v]$ then one can show that $G' = G \setminus \{(u, v)\}$ is a proper interval graph iff C was a clique, so again $k = 1$. We assume henceforth that $k > 1$ and $N(u) \neq N(v)$.

W.l.o.g. $i < j$. If $B_i = \{u\}, B_j = \{v\}, j = i + 1$ and B_i, B_j were far neighbors of each other, then we should split the contig into two contigs, one ending at B_i and the other beginning at B_j. Otherwise, updates to the straight enumeration are derived from the following lemma.

Lemma 8. *Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_i$ and $v \in B_j$ for some $1 \leq i < j \leq k$. Assume that $N(u) \neq N(v)$. Then G' is proper interval iff $F_r(B_i) = B_j$ and $F_l(B_j) = B_i$ in G.*

Proof. Assume that G' is proper interval. We will show that $F_r(B_i) = B_j$. The proof that $F_l(B_j) = B_i$ is symmetric. Since B_i and B_j are adjacent in G, $F_r(B_i) \geq B_j$. Suppose to the contrary that $F_r(B_i) > B_j$. Let $x \in F_r(B_i)$. Since x and v are in distinct blocks, either there is a vertex $a \in N[v] \setminus N[x]$ or there is a vertex $b \in N[x] \setminus N[v]$ (or both). In the first case, by the umbrella property $(a, u) \in E(G)$ and therefore u, x, v, a induce a chordless cycle in G'. In the second case, x, b, u, v induce a claw in G'. Hence, in both cases we arrive at a contradiction.

To prove the opposite direction we give a straight enumeration of $C \setminus \{(u, v)\}$. If $B_j = \{v\}$, $F_l(B_{i-1}) = F_l(B_i)$ and $F_r(B_{i-1}) = B_{j-1}$ (i.e., $N[u] = N[B_{i-1}]$ in G'), we move u into B_{i-1}. If B_i contained only u, $F_r(B_{j+1}) = F_r(B_j)$ and $F_l(B_{j+1}) = B_{i+1}$ (i.e., $N[v] = N[B_{j+1}]$ in G'), we move v into B_{j+1}. If u was not moved and $B_i \supset \{u\}$, then B_i is split into $\{u\}, B_i \setminus \{u\}$ in this order. If v was not moved and $B_j \supset \{v\}$, then B_j is split into $B_j \setminus \{v\}, \{v\}$ in this order. The result is a contig of $C \setminus \{(u, v)\}$.■

If the conditions of Lemma 8 are fulfilled, one has to update the data structure according to its proof. These updates require no knowledge about the connected components of G, and it can be shown that they take $O(1)$ time. Hence, from Sections 5.2 and 5.3 we obtain the following result:

Theorem 5. *The decremental proper interval graph representation problem is solvable in $O(1)$ time per removed edge.*

6 Maintaining the Connected Components

In this section we describe a fully dynamic algorithm for maintaining connectivity in a proper interval graph G in $O(\log n)$ time per operation. The algorithm receives as input a series of operations to be performed on a graph, which can be any of the following: Adding a vertex, adding an edge, deleting a vertex, deleting an edge or querying if two blocks are in the same connected component. The algorithm depends on a data structure which includes the blocks and the contigs of the graph. It hence interacts with the proper interval graph representation algorithm. In response to an update request, changes are made to the representation of the graph based on the structure of its connected components prior to the update. Only then are the connected components of the graph updated.

Let us denote by $B(G)$ the *block graph* of G, that is, a graph in which each vertex corresponds to a block of G and two vertices are adjacent iff their corresponding blocks are adjacent in G. The algorithm maintains a spanning forest F of $B(G)$. In order to decide if two blocks are in the same connected component, the algorithm checks if they belong to the same tree in F.

The key idea is to design F so that it can be efficiently updated upon a modification in G. We define the edges of F as follows: For every two vertices u and v in $B(G)$, $(u, v) \in E(F)$ iff their corresponding blocks are consecutive in a contig of G. Consequently, each tree in F is a path representing a contig. The crucial observation about F is that an addition or a deletion of a vertex or an edge in G induces $O(1)$ modifications to the vertices and edges of F. This can be seen by noting that each modification of G induces $O(1)$ updates to near pointers in our representation of G.

It remains to show how to implement a spanning forest in which trees may be cut when an edge is deleted from F, linked when an edge is inserted to F, and which allows to query for each vertex to which tree does it belong. All these operations are supported by the ET-tree data structure of [8] in $O(\log n)$ time per operation.

We are now ready to state our main result:

Theorem 6. *The fully dynamic proper interval graph representation problem is solvable in $O(d + \log n)$ time per modification involving d edges.*

7 The Lower Bound

In this section we prove a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation for fully dynamic proper interval graph recognition in the cell probe model of computation with word-size b [16].

Fredman and Saks [6] proved a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation for the following *parity prefix sum* (PPS) problem: Given an array of integers $A[1], \ldots, A[n]$ with initial value zero, execute an arbitrary sequence of Add(t) and Sum(t) operations, where an Add(t) increases $A[t]$ by 1, and Sum(t) returns $(\sum_{i=1}^{t} A[i])$ *mod* 2. Fredman and Henzinger [5] showed that the same lower bound applies to the problem of maintaining connectivity in general graphs, by showing a reduction from a modified PPS problem, called *helpful parity prefix sum*, for which they proved the same lower bound. A slight change to their reduction yields the same lower bound for the problem of maintaining connectivity in proper interval graphs, as the graph built in the reduction is a union of two paths and therefore proper interval. Using a similar construction we can prove the following result:

Theorem 7. *Fully dynamic proper interval recognition takes $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation in the cell probe model with word-size b.*

Acknowledgments

The first author gratefully acknowledges support from NSERC. The second author was supported in part by a grant from the Ministry of Science, Israel. The third author was supported by Eshkol scholarship from the Ministry of Science, Israel.

References

1. A. V. Carrano. Establishing the order of human chromosome-specific DNA fragments. In A. D. Woodhead and B. J. Barnhart, editors, *Biotechnology and the Human Genome*, pages 37–50. Plenum Press, New York, 1988.
2. D. Corneil, H. Kim, S. Natarajan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Inf. Proc. Letts.*, 55:99–104, 1995.
3. X. Deng, P. Hell, and J. Huang. Recognition and representation of proper circular arc graphs. In *Proc. 2nd Integer Programming and Combinatorial Optimization (IPCO)*, pages 114–121, 1992. Journal version: *SIAM Journal on Computing*, 25:390–403, 1996.
4. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. Foundations of Computer Science*, pages 60–69. IEEE, 1992.
5. M. Fredman and M. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*. to appear.
6. M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.
7. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
8. M. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the TwentySeventh Annual ACM Symposium on Theory of Computing*, pages 519–527, 1995.
9. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 79–89, New York, May 23–26 1998. ACM Press.
10. W.-L. Hsu. A simple test for interval graphs. In W.-L. Hsu and R. C. T. Lee, editors, *Proc. 18th Int. Workshop (WG '92), Graph-Theoretic Concepts in Computer Science*, pages 11–16. Springer-Verlag, 1992. LNCS 657.
11. L. Ibarra. Fully dynamic algorithms for chordal graphs. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, 1999.
12. C. G. Lekkerkerker and J. C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundam. Math.*, 51:45–64, 1962.
13. F. S. Roberts. Indifference graphs. In F. Harary, editor, *Proof Techniques in Graph Theory*, pages 139–146. Academic Press, New York, 1969.
14. J. Watson, M. Gilman, J. Witkowski, and M. Zoller. *Recombinant DNA*. W.H. Freeman, New York, 2nd edition, 1992.
15. G. Wegner. *Eigenschaften der Nerven homologisch einfacher Familien in* R^n. PhD thesis, Göttingen, 1967.
16. A. Yao. Should tables be sorted. *Assoc. Comput. Mach.*, 28(3):615–628, 1981.