A Fully Dynamic Algorithm for Recognizing and Representing Proper Interval Graphs *

Pavol Hell † Ron Shamir ‡ Roded Sharan ‡

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 $O(\log n)$ worst-case time 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. As a byproduct of our algorithm, we solve in $O(\log n)$ worst-case time the problem of maintaining connectivity in a dynamically changing proper

*Portions of this paper appeared in the Proceedings of the Seventh Annual European Symposium on Algorithms [9].

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

‡School of Computer Science, Tel Aviv University, Tel Aviv, Israel. {rshamir,roded}@post.tau.ac.il.

interval graph.

Keywords: Fully Dynamic Algorithms, Graph Algorithms, Proper Interval Graphs, Lower Bounds.

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 if and only if their assigned 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. [8, 16]), and several linear time algorithms are known for their recognition and realization [3, 4].

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 set of operations includes 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 similar lengths (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 if and only if 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 [18]). These translate into addition or deletion of vertices in the corresponding graph. Thus, 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 various graph families. Hsu [12] has given an $O(m + n \log n)$-time incremental algorithm for recognizing interval graphs. (Throughout, we denote the number of vertices and edges in a graph by n and m, respectively.) Deng, Hell and Huang [4] 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 [12, 4] only vertex additions are handled. Recently, Ibarra [13] devised a fully dynamic algorithm for recognizing chordal graphs, which handles each edge operation in $O(n)$ time. He also gave an optimal fully dynamic algorithm for recognizing split graphs, which handles each edge operation in $O(1)$ 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 [4]. We prove a close lower bound 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 [20]. It follows that our algorithm is nearly optimal (up to a factor of $O(\log\log n)$). We also give a fast $O(n)$ time algorithm for computing a realization of a proper interval graph given its representation, improving the $O(m+n)$ bound of [4].

For the incremental version of the problem we give an optimal algorithm (up to a constant factor) which handles each operation in time $O(d)$. This generalizes the result of [4] to arbitrary instances. The same bound is achieved for the decremental problem.

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 $O(d+\log n)$ worst-case time per operation involving d edges. In comparison, the best known algorithms for fully dynamic connectivity in general graphs require $O(\log n(\log\log n)^3)$ expected amortized time per edge operation [17], or $O(\log^2 n)$ amortized time per edge operation [11], or $O(\sqrt{n})$ worst-case time per edge operation [5]. Furthermore, we show that the lower bound of Fredman and Henzinger [10] of $\Omega(\log n/(\log\log n+\log b))$ amortized time per edge operation (in the cell probe model with word-size b) for fully dynamic connectivity in general graphs, applies also 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 lower bounds on

the amortized time per edge operation of fully dynamic algorithms for recognizing proper interval graphs, and for maintaining connectivity in proper interval graphs.

2 Preliminaries

Let $G = (V, E)$ be a graph. We denote its set of vertices also by $V(G)$ and its set of edges also by $E(G)$. For a vertex $v \in V$ we define $N(v) \equiv \{u \in V : (u, v) \in E\}$ and $N[v] \equiv N(v) \cup \{v\}$. Similarly, for a set $S \subseteq V$ we define $N(S) \equiv \cup_{v \in S} N(v)$ and $N[S] = N(S) \cup S$. Let R be an equivalence relation on V defined by uRv if and only if $N[u] = N[v]$. 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 *adjacent*, or *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 a linear 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 in Φ. The *out-degree* of a block B with respect to Φ, denoted by $o(B)$, is the number of neighbors of B which are ordered to its right in Φ. A *chordless cycle* is an induced cycle of length greater than 3. A *claw* is an induced $K_{1,3}$ (a 3-degree vertex connected to three 1-degree vertices). A graph is called *claw-free* if it contains no induced claw. For other definitions in graph theory see, e.g., [8].

We now quote some well-known properties of proper interval graphs that will be used in the sequel.

Theorem 2.1 *([14]) An interval graph (and in particular a proper interval graph) contains no chordless cycle.*

Theorem 2.2 *([19]) A graph is a proper interval graph if and only if it is an interval graph and is claw-free.*

Theorem 2.3 *([4]) A graph is a proper interval graph if and only if it has a straight enumeration.*

Lemma 2.4 ("The umbrella property") *([15]) 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).*

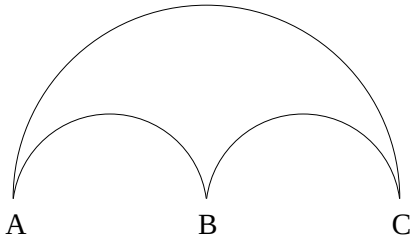

Figure 1: The umbrella property.

It is shown in [4] that a connected proper interval graph has a unique straight enumeration up to reversal. This motivates our representation of proper interval graphs: 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 in Section 3.1.

This information implicitly defines a realization of the dynamic graph (cf. [4]) as follows: Assign to each vertex in block B_i the interval $[i, i + o(B_i) + 1 - \frac{1}{i}]$. We show in Section 3.1 how to compute a realization of the dynamic graph 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. (Thus, $d = 1$ in case of an edge addition, and d is the degree in case of a vertex addition.) It is assumed that initially the graph is empty, or alternatively, that the representation of the initial graph is known. We also show how to compute in $O(n)$ time a realization of a graph given its representation.

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*, and their vertices are called *end-vertices*. The rest of the blocks are called *inner-blocks*.

As mentioned above, each connected component of the dynamic graph has exactly two contigs (which are 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

We now describe the details of how we keep our representation. 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) The *size* of the block.
   (b) Left and right *near pointers*, pointing to nearest neighbor blocks on the left and on the right respectively.
   (c) Left and right *far pointers*, pointing to farthest neighbor blocks on the left and on the right respectively.
   (d) Left and right *self pointers*, pointing to the block itself.

(e) 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.

In the following we shall omit details about the obvious updates to the names of the blocks containing each of the vertices (item 1), and to the block sizes (item 2a).

We introduce self pointers due to the possible need in the course of the algorithm 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 a block 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; and (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 $\&B$. $\&\emptyset$ denotes the null pointer. When we set a far pointer to point to a left or to a right self-pointer of B we shall 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 $N_r(A)$. We define $N_r(\emptyset) = N_l(\emptyset) = F_r(\emptyset) = F_l(\emptyset) = \&\emptyset$. 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'.

Given this data structure we can compute a realization of a contig C of G as follows: We first rank the blocks of C, starting with the leftmost block. This is done by choosing an arbitrary block of C, and marching up the enumeration of blocks of C using left near pointers, until we reach an end-block. We then set the rank of this block to 1, and march down the enumeration of blocks using right near pointers, until we reach the other end-block. We rank all the blocks of C along the way. Let us denote by $r(B)$ the rank of a block B. Then the out-degree of B is simply $o(B) = r(F_r(B)) - r(B)$, and the interval that we assign to the vertices of B is $[r(B), r(F_r(B)) + 1 - 1/r(B)]$. We conclude:

Theorem 3.1 *A realization of a proper interval graph which is represented using the data structure described above, can be computed in time $O(n)$.*

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 a proper interval graph.

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 the adjacencies of the new vertex, assuming that G' is a proper interval graph.

Lemma 3.2 *If G' is a proper interval graph then v can have neighbors in at most two connected components of G.*

Proof: Suppose to the contrary that x, y and z are neighbors of v in three distinct components of G. Then v, x, y and z induce a claw in G', a contradiction. ■

Lemma 3.3 *[4] Let C be a connected component of G containing neighbors of v. Let $B_1 < \ldots < B_k$ be a contig of C. Suppose that G' is a proper interval graph 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 $x <_\Phi y$ if and only if 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 either for every $x, y \in V(C)$, $x <_\Phi y$ implies $x <_{\Phi'} y$; or for every $x, y \in V(C)$, $x >_\Phi y$ implies $x <_{\Phi'} y$.

Lemma 3.4 *If H is a connected induced subgraph of a proper interval graph H', Φ is a contig of H, and Φ' is a straight enumeration of H', then Φ' is a refinement of Φ.*

Proof: By induction on the number of additional vertices in H': If $H' = H$ then the claim is obvious. Let $k = |V(H') \setminus V(H)|$. By the induction hypothesis, for a proper interval graph H'' which contains H (as an induced subgraph) and is contained in H', and for which $|V(H'') \setminus V(H)| = k - 1$, every straight enumeration is a refinement of Φ. Let C be a connected component of H'' for which $V(C) \supseteq V(H)$, and let Φ''_C be a contig of C. Let C' be a connected component of H' for which $V(C') \supseteq V(H)$ (and therefore $V(C') \supseteq V(C)$), and let Φ'_C be a contig of C'. In [4] it is constructively shown how Φ'_C is obtained as a refinement of Φ''_C (see also Section 3.3). Since Φ''_C is a refinement of Φ, the claim follows. ■

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 one of B or $B \cup \{v\}$ is a block of G'.

Corollary 3.5 *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 3.6 *Let C be a connected component of G containing neighbors of v. Let $\{B_1, \ldots, B_k\}$ denote the set of blocks in C which are adjacent to v, such that in a contig of C, $B_1 < \ldots < B_k$. If G' is a proper interval graph then the following properties are satisfied:*

1. *$B_1, \ldots, B_k$ are consecutive in a contig of 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 3.3. Claim 3 follows from part 3 of Lemma 3.3. 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 a proper interval graph since it is an induced subgraph of G'. It is composed of three types of blocks: Blocks whose vertices are from $V(C)$, which we will henceforth call C-blocks; blocks whose vertices are from $V(D)$, which we will henceforth call 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, each of which may split into $B_j \setminus N(v)$ and $B_j \cap N(v)$, $j = 1, k$.

Surely, in a contig of H all C-blocks must be ordered completely before or completely after all 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 3.4, 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.

Without loss of generality, $X \subseteq B_k$. Suppose to the contrary that v is not fully adjacent to B_k. Then by Lemma 3.4 we have $B_{k-1} \cap N(v) < B_k \setminus N(v) < \{v\}$ in Φ (note that these blocks are not consecutive), contradicting the umbrella property. We conclude that v is fully adjacent to B_k. Furthermore, B_1 must be adjacent to B_k, or else G' contains a claw consisting of v and one vertex from each of B_1, B_k, and $V(D) \cap N(v)$. It remains to show that B_1 is an inner-block in C. Suppose it is an end block. Since B_1 and B_k are adjacent, C consists of a single block, a contradiction. Thus, claim 4 is proved. ■

3.3 The DHH Algorithm

In our algorithm we rely on the incremental algorithm of Deng, Hell and Huang [4]. This algorithm handles the insertion of a new vertex into a connected proper interval graph in $O(d)$ time, changing its straight enumeration appropriately, or determining that the new graph is not a proper interval graph. We describe it briefly below. For simplicity, we assume throughout that the modified graph is a proper interval graph.

Let H be a connected proper interval graph, and let v be a vertex to be added, which is adjacent to d vertices in H. Let $\Phi = B_1 < \ldots < B_p$ denote a contig of H. By Lemma 3.6, the blocks to which v is fully adjacent occur consecutively in Φ. Assume that v is fully adjacent to $B_l, \ldots, B_r$, and for clarity we shall consider only the case where $1 < l < r < p$. Let $a = l - 1$ and $c = r + 1$. By Lemma 3.3(2) B_a and B_c are non-adjacent. Let $b > a$ be the largest index such that B_b is adjacent to B_a, and let $d < c$ be the smallest integer such that B_d is adjacent to B_c. It is shown in [4] that $a < b < d < c$.

In order to construct a straight enumeration of the new graph we have to distinguish between two cases:

1. If v is adjacent either to B_a or to B_c, then a straight enumeration of the new graph can be obtained as follows: If v is adjacent to B_a, we split B_a into $B_a \setminus N(v), B_a \cap N(v)$, list them in this order, and add $\{v\}$ as a block just after B_b. If v is adjacent to B_c, we split B_c into $B_c \cap N(v), B_c \setminus N(v)$ in this order, and add $\{v\}$ as a block just before B_d. In case v is adjacent to both B_a and B_c then these two instructions coincide, as shown in [4].

2. If v is adjacent neither to B_a nor to B_c then there are two possibilities: If there exists a block B_j, $b < j < d$, such that B_j is adjacent to both B_l and B_r, then a straight enumeration is obtained by adding v to B_j. Otherwise, let u be the least integer greater than b such that B_u is adjacent to B_r. Then a straight enumeration is obtained by inserting a new block $\{v\}$ just before B_u.

In Section 3.4 we show how to find the sequence of blocks $B_l, \ldots, B_r$ from our data structure in $O(d)$ time. Using near and far pointers we can find, in $O(1)$ time, $B_a = N_l(B_l), B_c = N_r(B_r), B_b = F_r(B_a)$ and $B_d = F_l(B_c)$. If v is adjacent to B_a or to B_c then updating the straight enumeration can be done in $O(1)$ time. Otherwise, finding B_j (if such exists) can be done in $O(d)$ time, and alternatively, finding $B_u = F_l(B_r)$ can be done in $O(1)$ time. Again in this case we can update the straight enumeration in $O(1)$ time. Hence, our data structure supports the insertion of a vertex of degree d in $O(d)$ time, when all its neighbors are in the same connected component.

3.4 Our Algorithm

We perform the following upon a request for adding a new vertex v: We make two passes over the neighbors of v. In the first pass we discover all blocks adjacent to v, and for each such block we allocate a *counter* and initialize it to zero. In the second pass, for each neighbor u of v we add one to the counter 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 up

the enumeration of blocks to the left using the left near 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 3.6(2)).

If the segment we found contains all the 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 3.2 and 3.6(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 3.6(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 3.6 are satisfied for both segments. 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 $\Phi = B_1 < \ldots < B_k$ and Φ^R be the two contigs of C. Let $\Psi = B'_1 < \ldots < B'_l$ and Ψ^R be the two contigs of D. The way the merge is performed depends on the identity of the end-blocks to which v is adjacent in each segment. If v is adjacent to B_k and 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 internal data structure in case these are the new contigs. The three other cases (e.g., v is adjacent to B_1 and B'_1, etc.) 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 which is 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 $B_0 = \emptyset$. If B_i was split we set $N_r(\hat{B}_i) = \&B_i, N_l(B_i) = \&\hat{B}_i, N_l(\hat{B}_i) = \&B_{i-1}$ and $N_r(B_{i-1}) = \&\hat{B}_i$. Analogous updates are made to the near pointers of $B'_j, \hat{B}'_j$ and B'_{j+1}, in case B'_j was split.

- 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 $F_r(\{v\}) = \&B'_j$.

The algorithm is summarized in Figure 2.

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 a proper interval graph and show how to efficiently detect them, and how to update our representation of the graph.

Lemma 4.1 *If u and v are in distinct connected components in G, then G' is a proper interval graph if and only if u and v are end-vertices in a straight enumeration of G.*

Input: A representation of the current graph G and a list of neighbors in G of a new vertex v.

Output: A representation of $G \cup \{v\}$ or a *False* value indicating that $G \cup \{v\}$ is not a proper interval graph.

1. Find the number s of segments of blocks which are adjacent to v.
2. If $s \geq 3$ then return *False*.
3. If $s = 1$ then apply the DHH algorithm.
4. Otherwise, proceed as follows ($s = 2$):
   (a) Check that exactly one extreme block in each segment is an end-block to which v is fully adjacent, and that the two extreme blocks in each segment are adjacent. Otherwise, return *False*.
   (b) Check that the two segments are in distinct contigs. Otherwise, return *False*.
   (c) Update the representation of the graph as described above.

Figure 2: An incremental algorithm for vertex addition.

Proof: To prove the 'only if' part let us examine the graph $H = G' \setminus \{u\} = G \setminus \{u\}$. H is a proper interval graph as it is an induced subgraph of G. If G' is also a proper interval graph, then by Lemma 3.6(3) v must be an end-vertex in a straight enumeration of G, 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 the required straight enumeration. ■

By the previous lemma if u and v are in distinct components in G, and G' is a proper interval graph, then they must reside in end-blocks of distinct contigs. We can check that in $O(1)$ time. In case u and v are end-vertices of two distinct contigs, we update our internal data structure as follows:

- Block enumeration: Given in the proof of Lemma 4.1.
- 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.
- Notation: Let $B_0 = \emptyset$ and $B'_{l+1} = \emptyset$. Let $B_k = B_k \setminus \{u\}$ and $B'_1 = B'_1 \setminus \{v\}$. If $B_k \neq \emptyset$ let $x = k$, and otherwise, let $x = k - 1$. If $B'_1 \neq \emptyset$ let $y = 1$, and otherwise, let $y = 2$.
- Near pointers: We set $N_r(\{u\}) = \&\{v\}, N_l(\{u\}) = \&B_x, N_l(\{v\}) = \&\{u\}$, and $N_r(\{v\}) = \&B'_y$. We also update $N_r(B_x) = \&\{u\}$ and $N_l(B'_y) = \&\{v\}$.
- Far pointers: We set $F_l(\{u\}) = F_l(B_k)$ and $F_r(\{v\}) = F_r(B'_1)$. We exchange the right self-pointer of B_k with the right self-pointer of $\{u\}$, and the left self-pointer of B'_1 with the left self-pointer of $\{v\}$. Finally, we set $F_r(\{u\}) = \&\{v\}$ and $F_l(\{v\}) = \&\{u\}$.

It remains to handle the case where u and v are 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 a proper interval graph and updates to the internal data structure are trivial. The remaining case is analyzed in the following lemma.

Lemma 4.2 *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 a proper interval graph if and only if $F_r(B_i) = B_{j-1}$ and $F_l(B_j) = B_{i+1}$ in G.*

Proof: Let G' be a proper interval graph. Since B_i and B_j are non-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 by the

umbrella property $N[v] \supset N[z]$ (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 and u induce a claw in G', a contradiction. Hence, $F_l(B_j) > B_{i+1}$ and therefore $F_r(B_{i+1}) < B_j$. Let $x \in B_{i+1}$ and let $y \in F_r(B_{i+1})$. As u and x are in distinct blocks, we have either $(u, y) \notin E(G)$ or there exists a vertex $a \in N[u] \setminus N[x]$ (or both). In the first case, v, u, x, y and the vertices on a shortest path from y to v induce a chordless cycle in G'. In the second case u, a, x and v induce a claw in G'. Hence, in both cases we arrive at a contradiction. By a symmetric argument we deduce that $F_l(B_j) = B_{i+1}$.

To prove the 'if' part we 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 contained vertices other than u, we split B_i into $B_i = B_i \setminus \{u\}, \{u\}$ in this order. If v was not moved and B_j contained vertices other than v, we split B_j into $\{v\}, B_j = B_j \setminus \{v\}$ in this order. It is easy to see that the result is a straight enumeration of $C \cup \{(u, v)\}$. ■

If u and v are neither end-vertices of distinct contigs, nor end-vertices of a three-block contig, then assuming that G' is a proper interval graph, the condition of Lemma 4.2 must hold. We can check that in time $O(1)$, and if it is the case, change our data structure so as to reflect the new straight enumeration of blocks given in the proof of Lemma 4.2. We describe below the changes to our data structure.

- Block enumeration: Given in the proof of Lemma 4.2.
- Near pointers: Let $B_{k+1} = \emptyset$. If u was moved into B_{i+1} then no change is necessary with respect to u. If $B_i \supset \{u\}$, u forms a new block and we set $N_l(\{u\}) = \&B_i, N_r(B_i) = \&\{u\}, N_r(\{u\}) = \&B_{i+1}$ and $N_l(B_{i+1}) = \&\{u\}$. Analogous updates are made with respect to v.
- Far pointers: If u was moved into B_{i+1}, then no change is necessary with respect to u. If $B_i \supset \{u\}$, we exchange the right self-pointer of B_i with the right self-pointer of (the new block) $\{u\}$. Let B denote the

block containing v in G'. We also set $F_l(\{u\}) = F_l(B_i)$ and $F_r(\{u\}) = \&B$. Analogous updates are made with respect to v.

The following theorem summarizes the results of Sections 3 and 4.

Theorem 4.3 *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 an operation involving d edges in $O(d + \log n)$ time. It supports four types of operations: Adding a vertex, adding an edge, deleting a vertex and deleting an edge. It is based on 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 both insertions and deletions are allowed. To bypass this problem we (implicitly) keep the identity of each block as an end/non-end block, but 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 provide a fully dynamic algorithm for maintaining the connected components of a proper interval graph. This algorithm handles a modification request involving d edges in $O(d + \log n)$ time, and determines whether two blocks are in the same connected component in $O(\log n)$ time. We describe below how each operation is handled by the fully dynamic proper interval graph representation algorithm.

5.1 The Addition of a Vertex

This operation is handled in essentially the same way as done by the incremental algorithm. However, in order to check if the end-blocks of two distinct segments are in distinct components, we query our data structure of

connected components (in $O(\log n)$ time), rather than checking if the end pointers of these blocks do not point to each other.

5.2 The Addition of an Edge

Again, handling this operation is similar to its handling by the incremental algorithm, with the exception that in order to check if the endpoints of an edge are in distinct components, we query our data structure of connected components (in $O(\log n)$ time).

5.3 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 in this case G' is an induced subgraph of G, and hence, also a proper interval graph.

Denote by X the block containing v. If X contains vertices other than v then the data structure is simply updated by deleting v. Hence, we concentrate on the case that $X = \{v\}$. In time $O(d)$ we can find 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 $1 \leq i \leq l \leq j \leq k$. The following updates should be performed:

- 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 these blocks by moving all vertices from B_i to B_{i-1} (in $O(d)$ time) and deleting B_i. If $l < j < k$ we deal similarly with 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 the other beginning at B_{l+1}.

- Near pointers: Let $B_0 = \emptyset, B_{k+1} = \emptyset$. 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 are made with respect to 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 with respect to 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.

5.4 The Deletion of an Edge

Let (u, v) be an edge of G to be deleted. Let C be the connected component of G containing u and v. Let B_i and B_j be the blocks containing u and v, respectively, in a contig $B_1 < \ldots < B_k$ of C. If $i = j = k = 1$ then B_1 is split into $\{u\}, B_1 \setminus \{u, v\}$ and $\{v\}$, in this order, resulting in a straight enumeration of G'. Updates are trivial in this case. Henceforth we assume that $k > 1$. We first observe that $i \neq j$, i.e., $N[u] \neq N[v]$:

Lemma 5.1 *If $N[u] = N[v]$ then G' is a proper interval graph if and only if C is a clique.*

Proof: To prove the 'only if' part, we first show that every vertex $x \in C \setminus \{u, v\}$ is adjacent to both u and v. Suppose to the contrary that there exists a vertex $x \in C \setminus \{u, v\}$ which is not adjacent to u. Let $x = x_1, \ldots, x_k = u$ be a path in C from x to u. Let x_i be the first vertex on the path which is adjacent to u (and therefore also to v). Then $\{x_i, x_{i-1}, u, v\}$ induce a claw in G', a contradiction. Finally, if a and b are two non-adjacent vertices in $C \setminus \{u, v\}$ then $\{a, u, b, v\}$ induce a chordless cycle in G', a contradiction.

To prove the 'if' part, notice that since C is a clique, it is a block in G, and therefore, $\{u\}, C \setminus \{u, v\}, \{v\}$ is a straight enumeration of $C \setminus \{(u, v)\}$. ■

Since by our assumptions $k > 1$, we conclude that $N[u] \neq N[v]$, and therefore, $N(u) \neq N(v)$. Without loss of generality, $i < j$. The updates to the straight enumeration of $C \setminus \{(u, v)\}$ are derived from the following lemma.

Lemma 5.2 *Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_i$ and $v \in B_j$ for some $1 \leq i < j \leq k$. Then G' is a proper interval graph if and only if $F_r(B_i) = B_j$ and $F_l(B_j) = B_i$ in G.*

Proof: Suppose that G' is a proper interval graph. We prove that $F_r(B_i) = B_j$. A symmetric argument shows that $F_l(B_j) = B_i$. 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 $x \in F_r(B_i)$. By the umbrella property $(x, v) \in E(G)$. Since x and v are in distinct blocks in G, either there exists a vertex $a \in N[v] \setminus N[x]$ or there exists a vertex $b \in N[x] \setminus N[v]$ (or both). In the first case, by the umbrella property $(a, u) \in E(G)$. Therefore, u, x, v and a induce a chordless cycle in G'. In the second case, x, b, u and v induce a claw in G'. Hence in both cases we arrive at a contradiction.

To prove the converse implication we give a straight enumeration of $C \setminus \{(u, v)\}$. If $B_i = \{u\}$, $B_j = \{v\}$, and $j = i + 1$, we have to split the contig into two contigs, one ending at B_i and the other beginning at B_j. 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 contains vertices other than u, then B_i is split into $\{u\}, B_i = B_i \setminus \{u\}$ in this order. If v was not moved and B_j contains vertices other than v, then B_j is split into $B_j = B_j \setminus \{v\}, \{v\}$ in this order. The result is a straight enumeration of $C \setminus \{(u, v)\}$. ■

If the conditions of Lemma 5.2 are fulfilled, then the following updates should be made:

- Block enumeration: Given in the proof of Lemma 5.2.

- Near pointers: Let $B_0 = \emptyset, B_{k+1} = \emptyset$. If $B_i = \{u\}$, $B_j = \{v\}$, and $j = i + 1$, we nullify $N_r(u)$. If B_i was split, we set $N_r(\{u\}) = \&B_i$, $N_l(B_i) = \&\{u\}$, $N_l(\{u\}) = \&B_{i-1}$ and $N_r(B_{i-1}) = \&\{u\}$. If B_i contained only u, and u was moved into B_{i-1}, we update $N_r(B_{i-1}) = \&B_{i+1}$ and $N_l(B_{i+1}) = \&B_{i-1}$. Analogous updates are made with respect to v.

- Far pointers: If $B_i = \{u\}$, $B_j = \{v\}$, and $j = i + 1$, we nullify $F_r(u)$. If B_i was split, we exchange the left self-pointer of B_i with the left self-pointer of $\{u\}$. We also set $F_l(\{u\}) = F_l(B_i)$ and $F_r(\{u\}) = \&B_y$, where $y = j$ in case v is no longer in B_j (that is, v was moved into B_{j+1} or B_j was split), and otherwise, $y = j - 1$. If B_i contained only u, and u was moved into B_{i-1}, we exchange the right self-pointer of B_i with the right self-pointers of B_{i-1}, and delete B_i. Analogous updates are made with respect to v.

Note that these updates take $O(1)$ time and require no knowledge about the connected components of G. Hence, from Sections 5.3 and 5.4 there follows an optimal algorithm for the decremental proper interval graph representation problem. The following theorem summarizes this result:

Theorem 5.3 *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 $O(d + \log n)$ time per operation involving d edges. In Section 7 we shall establish a lower bound 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) for this problem.

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 vertices

are in the same connected component. It operates on the blocks of the graph rather than on its vertices. 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. We provide a data structure of connected components which performs each operation in $O(\log n)$ time.

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 if and only if their corresponding blocks are adjacent in G. The algorithm maintains a spanning forest F of $B(G)$. When a modification in the graph occurs, the spanning forest is updated accordingly. 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 $B(G)$, $(u,v) \in E(F)$ if and only if their corresponding blocks are consecutive in a contig of G (or equivalently, if the near pointers of these blocks point to each other in our representation). 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 a data structure for storing F that allows to query for each vertex to which path it belongs, and that enables splitting a path upon a deletion of an edge in F, and linking two paths upon an addition of an edge to F. If we store the vertices of each path of F in a balanced binary tree, then each of these operations can be supported in $O(\log n)$ time (cf. [2]).

We are now ready to state our main result:

Theorem 6.1 *The fully dynamic proper interval graph representation problem is solvable in $O(d + \log n)$ time per modification involving d edges.*

We note, that the performance of our representation algorithm depends on the performance of a data structure of connected components of a graph, which is a union of disjoint paths, that supports the following operations: Linking two paths, splitting a path, and querying if two vertices belong to the same path. Given such a data structure which supports each operation in $O(f(n))$ time, for some function f, our representation algorithm can be implemented to run in $O(d + f(n))$ time per modification involving d edges.

7 The Lower Bounds

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 (see [20] for details about the model). Furthermore, we prove the same lower bound also for the problem of fully dynamic connectivity maintenance of a proper interval graph.

Fredman and Henzinger [10] have shown a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation (in the cell probe model of computation with word-size b) for fully dynamic connectivity, by reduction from the *helpful parity prefix sum* (HPPS) problem, which is defined below. We use similar constructions in our lower bound proofs.

The HPPS problem is a modified *parity prefix sum* problem (see [7] for definition of the latter problem). Its lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation follows from the work of Fredman and Saks [7]. It is defined as follows: Given an array $A[0], \ldots, A[n+1]$ of zeroes and ones such that initially all $A[i]$ are 0, except $A[0]$ and $A[n+1]$ which are 1, execute an arbitrary sequence of the following operations:

Add(t, i, j): If $0 \leq i < t < j \leq n+1, A[i] = 1, A[j] = 1$, and $A[k] = 0$ for all $i < k < j$, then

$A[t] = (A[t] + 1) \ mod \ 2$. Otherwise, do nothing.

Sum(t): Return $(\sum_{i=1}^{t} A[i]) \ mod \ 2$.

Theorem 7.1 *Fully dynamic proper interval graph recognition takes amortized time* $\Omega(\log n/(\log\log n + \log b))$ *per edge operation in the cell probe model of computation with word-size* b.

Proof: Given an instance of the HPPS problem (i.e., a sequence of Add and Sum operations) we construct an instance of the dynamic proper interval graph recognition problem, such that each Add operation corresponds to $O(1)$ edge modifications in the dynamic proper interval graph instance, and each Sum operation corresponds to $O(1)$ temporary modifications in the dynamic graph: Depending on whether the modification generates a proper interval graph we answer the Sum query and then reverse the modification. Thus, the lower bound for the HPPS problem shows that there exists a sequence of m operations for the dynamic proper interval recognition problem that takes $\Omega(m \log n/(\log\log n + \log b))$ time in the cell probe model of computation with word-size b.

Let $S_{-1} = 0$ and let $S_0 = 1$. Given an instance of the HPPS problem, define $S_t \equiv (\sum_{i=1}^{t} A[i]) \ mod \ 2$ for $1 \leq t \leq n$. The reduction is as follows: We construct a graph $G = (V, E)$ with $n + 2$ vertices labeled $-1, 0, 1, \ldots, n$, where each vertex v represents S_v. If $S_t = i$, for $i = 0, 1$, and $t' < t$ is the largest index such that $S_{t'} = i$, then G contains the edge (t', t). In words, vertices t for which $S_t = 1$ are connected in a chain, which we henceforth call the *odd chain*, and all other vertices are connected in a chain, which we henceforth call the *even chain*. Note, that the vertex labeled -1 lies on the even chain, and the vertex labeled 0 lies on the odd chain.

To answer a Sum(t) query ($1 \leq t \leq n$) we do the following:

1. If $(0, t) \in E$ or $(0, t'), (t', t) \in E$ for some vertex $t' \in V$, we output 1.
2. Otherwise, let t' be a vertex such that $t' > t$ and $(t, t') \in E$. If such a vertex exists, define $H \equiv G \setminus \{(t, t')\} \cup \{(0, t)\}$. Otherwise, let $H \equiv G \cup \{(0, t)\}$. If t is on the odd chain then this modification forms a chordless cycle. If t is on the even chain then the new graph is a single path or a union of two disjoint paths.

Hence, H is a proper interval graph if and only if Sum$(t) = 0$. Thus, if H is a proper interval graph we output 0, and otherwise, we output 1. Note, that G is not modified in this case.

To perform an Add(t, i, j) operation we do the following:

1. Let i_{odd} (i_{even}) be the largest vertex on the odd (even) chain with $i_{odd} < t$ ($i_{even} < t$). Let j_{odd} (j_{even}) be the smallest vertex on the odd (even) chain with $j_{odd} \geq t$ ($j_{even} \geq t$), if such a vertex exists.
2. Delete from G the edges (i_{odd}, j_{odd}) and (i_{even}, j_{even}).
3. Add to G the edges (i_{odd}, j_{even}) and (i_{even}, j_{odd}).

By [10, Lemma 3.1] $i_{odd}, j_{odd}, i_{even}$ and j_{even} can be found by querying Sum(t) as follows: If Sum$(t) = 1$ then $i_{odd} = t - 1$, $j_{odd} = t$, $i_{even} = i - 1$ if $i > 1$, or $i_{even} = -1$ otherwise; and $j_{even} = j$ if $j <= n$, or j_{even} is undefined otherwise. If Sum$(t) = 0$ then $i_{odd} = i - 1$ if $i > 0$, or $i_{odd} = 0$ otherwise; $j_{odd} = j$ if $j \leq n$, or j_{odd} is undefined otherwise; $i_{even} = t - 1$ if $t > 1$, or $i_{even} = -1$ otherwise; and $j_{even} = t$. This completes the reduction. ■

Note, that since the key to the reduction above is the ability to detect cycles, similar arguments can be used to show that the same lower bound applies also to other problems, e.g., fully dynamic interval graph recognition and fully dynamic chordal graph recognition.

Theorem 7.2 *There is a lower bound 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 *for fully dynamic connectivity maintenance of a proper interval graph.*

Proof: We use the same reduction as in the proof of Theorem 7.1, with the exception that in order to answer a $Sum(t)$ query we check whether vertices 0 and t are connected. If the answer is positive we output 1, and

otherwise we output 0. The reduction is valid, since the graph G, which is constructed in the reduction, is a union of two disjoint paths, and therefore, is a proper interval graph. ■

Recently, a simpler reduction was given from the parity prefix sum problem to fully dynamic connectivity [6]. This reduction allows the repeated modification of $A[t]$ for the same argument t, and leads to alternative proofs of Theorems 7.1 and 7.2.

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 an Eshkol fellowship from the Ministry of Science, Israel.

References

[1] A. V. CARRANO, *Establishing the order of human chromosome-specific DNA fragments*, in Biotechnology and the Human Genome, A. D. Woodhead and B. J. Barnhart, eds., Plenum Press, New York, 1988, pp. 37–50.

[2] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.

[3] D. CORNEIL, H. KIM, S. NATARAJAN, S. OLARIU, AND A. P. SPRAGUE, *Simple linear time recognition of unit interval graphs*, Inf. Proc. Letts., 55 (1995), pp. 99–104.

[4] X. DENG, P. HELL, AND J. HUANG, *Recognition and representation of proper circular arc graphs*, in Proc. 2nd Integer Programming and Combinatorial Optimization (IPCO), 1992, pp. 114–121. Journal version: *SIAM Journal on Computing*, 25:390–403, 1996.

[5] D. ESTEIN, Z. GALIL, G. F. ITALIANO, AND A. NISSENZWEIG, *Sparsification — a technique for speeding up dynamic graph algorithms*, Journal of the ACM, 44 (1997), pp. 669–696.

[6] F. FICH. Private communication, 2000.

[7] 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, 1989, pp. 345–354.

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

[9] P. HELL, R. SHAMIR, AND R. SHARAN, *A fully dynamic algorithm for recognizing and representing proper interval graphs*, in Proceedings of the Seventh Annual European Symposium on Algorithms, LNCS 1643, 1999, pp. 527–539.

[10] M. HENZINGER AND M. FREDMAN, *Lower bounds for fully dynamic connectivity problems in graphs*, Algorithmica, 22 (1998), pp. 351–362.

[11] 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), New York, May 23–26 1998, ACM Press, pp. 79–89.

[12] W.-L. HSU, *A simple test for interval graphs*, in Proc. 18th Int. Workshop (WG '92), Graph-Theoretic Concepts in Computer Science, W.-L. Hsu and R. C. T. Lee, eds., Springer-Verlag, 1992, pp. 11–16. LNCS 657.

[13] L. IBARRA, *Fully dynamic algorithms for chordal graphs*, in Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), 1999, pp. 923–924.

[14] C. G. LEKKERKERKER AND J. C. BOLAND, *Representation of a finite graph by a set of intervals on the real line*, Fundam. Math., 51 (1962), pp. 45–64.

[15] P. LOOGES AND S. OLARIU, *Optimal greedy algorithms for indifference graphs*, Comput. Math. Appl., 25 (1993), pp. 15–25.

[16] F. S. ROBERTS, *Indifference graphs*, in Proof Techniques in Graph Theory, F. Harary, ed., Academic Press, New York, 1969, pp. 139–146.

[17] M. THORUP, *Near-optimal fully-dynamic graph connectivity*, in Proceedings of the 32th Annual ACM Symposium on Theory of Computing (STOC'00), 2000. To appear.

[18] J. WATSON, M. GILMAN, J. WITKOWSKI, AND M. ZOLLER, *Recombinant DNA*, W.H. Freeman, New York, 2nd ed., 1992.

[19] G. WEGNER, *Eigenschaften der Nerven homologisch einfacher Familien in* R^n, PhD thesis, Göttingen, 1967.

[20] A. YAO, *Should tables be sorted*, Assoc. Comput. Mach., 28(3) (1981), pp. 615–628.