A lexBFS Algorithm for Proper Interval Graph Recognition

Celina M. Herrera de Figueiredo
João Meidanis   *Célia Picinin de Mello*

Relatório Técnico DCC–04/93

Março de 1993

A lexBFS Algorithm for Proper Interval Graph Recognition

Celina M. Herrera de Figueiredo[*] João Meidanis[†]
Célia Picinin de Mello[‡]

Abstract

Interval graphs are the intersection graphs of families of intervals in the real line. If the intervals can be chosen so that no interval contains another, we obtain the subclass of proper interval graphs. In this paper, we show how to recognize proper interval graphs with one lexBFS. This algorithm runs in linear time, and produces as a by-product the clique partition of the input graph.

1 Introduction

Interval graphs, which are the intersection graphs of families of intervals in the real line, have countless applications and have been extensively studied since their inception [4]. If the intervals can be chosen so that no interval contains another, we obtain precisely the class of interval graphs that do not contain a forbidden induced subgraph [10]. Such graphs are called accordingly *proper interval graphs*. The names *indifference graph*

[*]Universidade Federal do Rio de Janeiro, Instituto de Matemática, Caixa Postal 68530, 21944 Rio de Janeiro, RJ, Brasil.

[†]Universidade Estadual de Campinas, Departamento de Ciência da Computação, Caixa Postal 6065, 13081-970 Campinas, SP, Brasil.

[‡]Universidade Estadual de Campinas, Departamento de Ciência da Computação, Caixa Postal 6065, 13081-970 Campinas, SP, Brasil.

and *time graph* are also used in the literature as synonyms of proper interval graph, and illustrate some of the several applications involving this subclass [10, 6].

The first linear-time algorithm for interval graph recognition appeared in 1975 [1]. This algorithm starts by determining the maximal cliques of the graph and then tries to find a particular kind of ordering of the maximal cliques which characterizes interval graphs (see Section 2 for details on this ordering). The maximal cliques of an interval graph can be found in linear time using a *lexicographic breadth-first search* (lexBFS). Special structures called PQ-trees are then used to find the correct clique orderings. However, in contrast with the nice theoretical appeal of PQ-trees, their manipulation is cumbersome, leading to intricate algorithms.

In 1989, Korte and Möhring introduced their MPQ-trees, a modified version of the original PQ-trees, and gave a simpler, incremental algorithm for interval graph recognition, still running in linear time [7].

More recently, in 1991, K. Simon presented an intriguing algorithm, in which four iterations of lexBFS suffice to recognize interval graphs [13]. No PQ-trees or related structures are needed. This is a hint that lexBFS may in fact be much more powerful than previously thought.

It is known that one run of lexBFS is enough to recognize triangulated graphs [12]. Simon's work proves that we can recognize interval graphs with four lexBFS's. In this paper, we show how to recognize proper interval graphs with one lexBFS. Our recognition algorithm for proper interval graphs runs in linear time, and requires an additional traversal of the graph after the lexBFS to check for specific conditions, but does not involve PQ-trees or MPQ-trees.

Although we are not aware of any published explicit recognition algorithm for proper interval graphs, we remark that it is possible to use Booth and Lueker's test of the consecutive ones property to solve the problem, due to the following interesting fact. If M is the vertices $\times$ maximal cliques incidence matrix for a graph G, then G is an interval graph if and only if M has the consecutive ones property for *columns* and G is a proper interval graph if and only if M has the consecutive ones property for *lines*.

However, this method has all the problems associated with the use of PQ-trees. The algorithm we present here uses a much simpler structure, which resembles the one used in lexBFS itself. The vertices are added to this structure in the order they are visited by the lexBFS, an idea already present in the MPQ-tree algorithm [7]. In addition, our algorithm produces as a by-product the *clique partition* (see section 2) of the input graph.

The rest of the paper is organized as follows. In Section 2 we give the basic definitions and notations used throughout. Section 3 contains an informal description of the algorithm, followed by a detailed one in Section 4. A proof of correctness is presented in Section 5, and implementation issues are discussed in Section 6. Finally, our concluding remarks and plans for future work appear in Section 7.

2 Definitions and Notations

In this paper, G denotes an undirected, finite, connected graph. $V(G)$ and $E(G)$ are the vertex and edge sets of G, respectively. A graph G is a *trivial* graph if $\mid V(G) \mid = 1$. A *clique* is a set of vertices pairwise adjacent in G. A *maximal clique* of G is a clique not properly contained in any other clique. For $V' \subset V(G)$, we denote by $G[V']$ the *subgraph induced by* V', that is, the vertex set of $G[V']$ is V' and the edge set of $G[V']$ consists of those edges of G having both ends in V'.

For each vertex v of a graph G, $Adj(v)$ denotes the set of vertices which are adjacent to v. In addition, $N(v)$ denotes the *neighborhood* of v, that is, $N(v) = Adj(v) \cup \{v\}$. We extend the domain of N to subsets V' of $V(G)$ by making $N(V') = \cup_{v \in V'} N(v)$.

Let $\mathcal{C} = \{C_1, ..., C_\ell\}$ be the set of maximal cliques of G and $Y \subseteq \mathcal{C}$. The ℓ maximal cliques of G induce a partition on the vertex set $V(G)$ of G, called the *clique partition* of G, whose elements are the non-empty sets V_Y defined as:

$$V_Y = \bigcap_{C \in Y} C \ \setminus \bigcup_{C \in \mathcal{C} \setminus Y} C$$

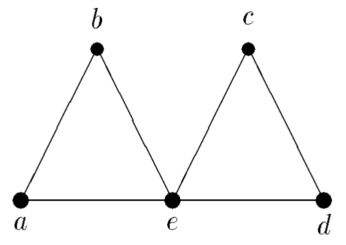


Figure 1: Two maximal cliques.

Thus, V_Y denotes the subset of $V(G)$ formed by the vertices belonging exactly to the maximal cliques of Y. Each $v \in V(G)$ belongs to exactly one subset V_Y. Figure 1 illustrates a graph with two maximal cliques $C_1 = \{a, b, e\}$ and $C_2 = \{c, d, e\}$. Then $V_{\{C_1\}} = \{a, b\}$, $V_{\{C_2\}} = \{c, d\}$ and $V_{\{C_1,C_2\}} = \{e\}$.

Two adjacent vertices v, w are *twins* when $N(v) = N(w)$. The following observation is immediate.

Observation 1 *Two vertices are twins if and only if they belong to the same element of the clique partition.*

A graph is *reduced* if no two distinct vertices are twins. The *reduced graph* G' of a graph G is the graph obtained from G by collapsing each set of twins into a single vertex and removing possible resulting parallel edges and loops.

A graph is *interval* if it is the intersection graph of a set of intervals of the real line. Moreover, if unitary intervals can be taken, then the graph is called *unitary interval*, *proper interval* or *indifference*. We shall adopt the latter.

Interval graphs are precisely those which admit a linear order on the set of maximal cliques such that the maximal cliques containing the same vertex are consecutive [2]. A clique that can appear as the first (or the last) clique in such an order will be called an *outer clique*.

Indifference graphs too can be characterized by a linear order: their vertices can be linearly ordered such that the vertices contained in the same clique are consecutive [11], [8]. We shall call such an order *indifference order*. We remark that this order is not unique in general. For instance, the reverse order is also an indifference order. If the graph is reduced, then there are at most two indifference orders [11].

Another characterization of indifference graphs shows they are interval graphs that do not admit $K_{1,3}$ as an induced subgraph [10] (see Figure 2).

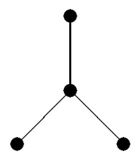


Figure 2: $K_{1,3}$ is forbidden in indifference graphs.

A vertex v is *simplicial* when $N(v)$ is a clique. In addition, for interval graphs, a vertex v is *external simplicial* when $N(v)$ is an outer clique. In an indifference graph, external simplicial vertices are exactly those which can start (or end) an indifference order.

A *lexicographic breadth-first search* (lexBFS) is a breadth-first search procedure with the additional rule that vertices with earlier visited neighbors are preferred. Following Korte and Möhring [7], we say that v, $w \in V(G)$ *disagree* on $u \in V(G)$ if one of them is adjacent to u, but not both. Then lexBFS produces an ordering $(v_1, \ldots, v_n)$ of $V(G)$ with the property:

> If there is some v_i on which v_j, v_k with $i < j < k$ disagree, then the leftmost vertex on which they disagree is adjacent to v_j.

A graph is *triangulated* or *chordal* when it does not contain induced cycles of length greater than three. Every interval graph is triangulated [3].

The classical triangulated graph recognition algorithm uses one lex-BFS to linearly order the vertices of the graph [12]. We call this order a *simplicial order* as it satisfies: v_i is a simplicial vertex in $G[v_1, \ldots, v_i]$.

3 Informal description of the algorithm

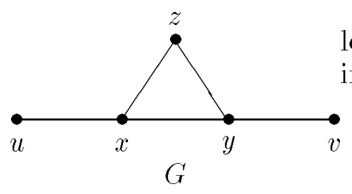
The algorithm works by trying to construct an indifference order for the input graph. It will succeed if and only if the graph is an indifference graph.

The starting point for this algorithm is the observation that a lexBFS in an indifference graph always ends in an external simplicial vertex (see Section 5 for a proof). Thus, the idea is to remove this last vertex (call it v), perform a recursive call with the remaining graph, and then include v in the indifference order returned. Since v is external simplicial, there are only two possibilities: it goes either in the beginning or in the end of the returned order. If v is not adjacent to either extreme, or if the recursive call rejects, the algorithm rejects.

Furthermore, since a lexBFS ordering without its last vertex is still a lexBFS on the graph without this vertex, a single lexBFS run can be used throughout the whole process.

Unfortunately, this simple scheme does not work because there are, in general, many indifference orders for a given graph. If the wrong one is returned, the algorithm will reject a good graph. As an example, consider the graph G depicted in Figure 3. There are two indifference orders for $G \setminus \{v\}$. Only one of them permits the placement of v besides its only neighbor y. In the other order, y is "hidden" by the other nodes, so the algorithm will incorrectly report that G is not an indifference graph.

To overcome this problem, we work on the *reduced* graph, where the indifference order is unique (except for its reverse) [11]. As a by-product , the algorithm constructs the reduced graph of G, by giving the clique partition of G, and an indifference order for it. This reduced indifference order can be seen as a compact representation of *all* indifference orders for G. The details of this procedure are described in the next section.



lexBFS ordering: $u\ x\ y\ z\ v$.
indifference orders for $G - v$:
$u\ x\ z\ y$.
$u\ x\ y\ z$.

Figure 3: The naïve algorithm fails to recognize this indifference graph.

4 The algorithm

The input for our algorithm is a graph $G(V, E)$, and the output is a variable *indif* which is *true* if the graph is an indifference graph and *false* otherwise. If the algorithm terminates with *indif* = *true*, then a variable P stores an indifference order for the reduced graph of G.

The algorithm considers the vertex set of G ordered according to a lexBFS, $(v_1, \ldots, v_n)$. It builds a sequence of induced graphs of G, defined by $G_i = G[v_1, \ldots, v_i]$. For each value of i, either the indifference order for the reduced graph G_i' of G_i is built or the value of the variable *indif* is changed to *false* and the algorithm stops.

In our description, we repeatedly use the notation $Adj(i, v)$ to mean $Adj(v) \cap \{v_1, \ldots, v_i\}$ and also $N(i, v) = N(v) \cap \{v_1, \ldots, v_i\}$, with an extended notation for sets.

Notice that the recursion mentioned in Section 3 was replaced by an equivalent iterative statement (**while** loop). The algorithm appears in Figure 4.

5 Proof of correctness

In this section we will be consistent with the notation used in the algorithm as much as possible.

```
**Indif**($G$)

Let $(v_1, \dots, v_n)$ be a lexBFS order on $G$;
Initialize: $P \longleftarrow (\{v_1\})$; $indif \longleftarrow true$; $i \longleftarrow 1$;
**while** $i < n$ **and** $indif$ **do**
    $v \longleftarrow v_{i+1}$;
    Let $P = (A_1, \dots, A_l)$; $/\star$ $P$ is the indifference order for $G_i'$ $\star/$
    **if** $Adj(i,v) \cap A_1 \neq \emptyset$ **then**
      Take $j$ as the minimum index such that $Adj(i,v) \subseteq A_1 \cup \dots \cup A_j$;
      **if** $(A_j \cap N(i, A_1) = \emptyset)$ **or** $(A_1 \cup \dots \cup A_{j-1} \not\subseteq Adj(i,v))$ **then**
        $indif \longleftarrow false$;
      **if** $A_j \subseteq Adj(i,v)$ **then**
        **if** $\exists\ w \in A_1$ with $N(i,v) = N(i,w)$ **then**
          $P \longleftarrow (A_1 \cup \{v\}, A_2, \dots, A_l)$
        **else** $P \longleftarrow (\{v\}, A_1, \dots, A_l)$
      **else** $P \longleftarrow (\{v\}, A_1, \dots, A_{j-1}, B, C, \dots, A_l)$
            with $B = A_j \cap Adj(i,v)$ and $C = A_j \setminus Adj(i,v)$
    **else** $/\star$ $Adj(i,v) \cap A_1 = \emptyset$ $\star/$
        Take $j$ as the maximum index such that $Adj(i,v) \subseteq A_j \cup \dots \cup A_l$;
        **if** $(A_j \cap N(i, A_l) = \emptyset)$ **or** $(A_{j+1} \cup \dots \cup A_l \not\subseteq Adj(i,v))$ **then**
          $indif \longleftarrow false$;
        **if** $A_j \subseteq Adj(i,v)$ **then**
          **if** $\exists\ w \in A_l$ with $N(i,v) = N(i,w)$ **then**
            $P \longleftarrow (A_1, A_2, \dots, A_l \cup \{v\})$
          **else** $P \longleftarrow (A_1, \dots, A_l, \{v\})$
        **else** $P \longleftarrow (A_1, \dots, A_{j-1}, C, B, \dots, A_l, \{v\})$
              with $B = A_j \cap Adj(i,v)$ and $C = A_j \setminus Adj(i,v)$
    $i \longleftarrow i + 1$;
**return** $indif$
```

Figure 4: Our lexBFS Algorithm for Indifference Graph Recognition.

The following results imply that v_{i+1} is an external simplicial vertex of G_{i+1}.

Lemma 1 *Let G be an indifference graph, $(v_1, \ldots, v_n)$ a lexBFS ordering of its vertices, $G_i = G[v_1, \ldots, v_i]$, $v = v_{i+1}$, $Adj(i, v)$ and $A_1, \ldots, A_l$ as in the algorithm. Then either $Adj(i, v) \cap A_1 \neq \emptyset$ or $Adj(i, v) \cap A_l \neq \emptyset$.*

Proof: By Theorem 3.3 of [7], $Adj(i, v)$ is contained in an outer clique of G_i. This is true for any interval graph. Since G_i is an indifference graph, there are just two outer cliques, $N(i, A_1)$ and $N(i, A_l)$. Moreover, when $Adj(i, v) \subseteq N(i, A_1)$ then $Adj(i, v) \cap A_1 \neq \emptyset$ and when $Adj(i, v) \subseteq N(i, A_l)$ then $Adj(i, v) \cap A_l \neq \emptyset$. We shall prove the former implication in the sequel (the other one is analogous).

If $Adj(i, v) \cap A_1 = \emptyset$, let a_1 be any vertex of A_1 and w be any element of $Adj(i, v)$. Since $Adj(i, v) \subseteq N(i, A_1)$, w and a_1 are adjacent, but only w is adjacent to v. Furthermore, $Adj(i, w) \not\subseteq N(i, A_1)$ because w and a_1 are not twins. Now taking any element $z \in Adj(i, w) \setminus N(i, A_1)$ we have that $G_i[w, v, a_1, z]$ is isomorphic to $K_{1,3}$, contradicting the fact that G_i is an indifference graph. Hence, $Adj(i, v)$ must meet A_1. $\square$

Lemma 2 *Let $(v_1, \ldots, v_n)$ be a lexBFS order on an indifference graph G. For all subgraphs $G_i = G[v_1, \ldots, v_i]$, an indifference order $(A_1, \ldots, A_n)$ on the reduced graph G_i' satisfies, for $v = v_{i+1}$:*

i. If $Adj(i, v) \cap A_1 \neq \emptyset$, then, for j the minimum index such that $Adj(i, v) \subseteq A_1 \cup \cdots \cup A_j$, we have $A_j \cap N(i, A_1) \neq \emptyset$ and $A_1 \cup \cdots \cup A_{j-1} \subseteq Adj(i, v)$.

ii. If $Adj(i, v) \cap A_l \neq \emptyset$, then, for j the maximum index such that $Adj(i, v) \subseteq A_j \cup \cdots \cup A_l$, we have $A_j \cap N(i, A_l) \neq \emptyset$ and $A_{j+1} \cup \cdots \cup A_l \subseteq Adj(i, v)$.

Proof: We shall only prove Statement i here. Statement ii can be proved analogously, or by reasoning with the reverse indifference order.

Since $Adj(i,v) \cap A_1 \neq \emptyset$, take $a_1 \in Adj(i,v) \cap A_1$. By the minimality of j, there is also $a_j \in Adj(i,v) \cap A_j$. If $A_j \cap N(i, A_1) = \emptyset$, v would not be a simplicial vertex in G_{i+1}, because a_1 and a_j would be two nonadjacent vertices in $Adj(i+1, v)$. Therefore, $A_j \cap N(i, A_1) \neq \emptyset$.

We shall now show that $A_1 \cup \cdots A_{j-1} \subseteq Adj(i,v)$. If $j = 1$, this is immediate, since the left-hand side is empty. For $j > 1$, assume for a moment that the inclusion does not hold, and take $w \in (A_1 \cup \cdots A_{j-1}) \setminus Adj(i,v)$. Notice that a_j is adjacent to w, because both belong to the clique $N(i, A_1)$. However, being in distinct classes, they are not twins in G_i. It follows that there is either an element adjacent to w but not to a_j, or an element adjacent to a_j but not to w. The first case must be excluded since such an element would necessarily be in a class to the left of w in the indifference order, and a_j meets all classes down to A_1. Hence, there is $z \in Adj(i, a_j) \setminus Adj(i, w)$. But then $G[a_j, v, w, z]$ is isomorphic to $K_{1,3}$, a contradiction. $\square$

The following theorem proves the correctness of our algorithm.

Theorem 1 *The algorithm returns* indif = true *if and only if G is an indifference graph.*

Proof: Suppose G is an indifference graph. Then, by Lemmas 1 and 2, all computation paths which assign *false* to *indif* are avoided. Since *indif* starts out with the value *true*, it keeps this value until the end.

Conversely, suppose that the algorithm returns *indif* = *true*. We shall prove the following invariant of the **while** loop.

Claim: Each time the **while** test is done, if the variable *indif* has value *true*, then $P = (A_1, \ldots, A_l)$ is an indifference order for reduced graph G_i', i.e., G_i is an indifference graph.

For $i = 1$ this is obvious, since G_1 is a trivial graph and $P = (\{v_1\})$.

Consider now the i-th iteration of the loop.

Let $P = (A_1, \ldots, A_l)$ be the indifference order for reduced graph of G_i.

Let $v = v_{i+1}$. Suppose $Adj(i,v) \cap A_1 \neq \emptyset$ and let j be the minimum index such that $Adj(i,v) \subseteq A_1 \cup \cdots \cup A_j$.

Since that the algorithm does not end with *indif* $=$ *false*, $A_j \cap N(i, A_1) \neq \emptyset$ and $A_1 \cup \cdots \cup A_{j-1} \subseteq Adj(i, v)$.

We will examine the cases that occur according to the tests performed in the algorithm.

Case 1: $A_j \subseteq Adj(i, v)$.

Then $A_1 \cup \cdots \cup A_j = Adj(i, v)$. This implies that two vertices x, y distinct from v are twins in G_i if and only if they are twins in G_{i+1}. Hence, there will be no changes in the current classes, except for the possible inclusion of v in one of them.

If there exists $w \in A_1$ such that $N(i, w) = N(i, v)$, then v and w are twins in G_{i+1}, and they must be in the same element of the clique partition of G_{i+1}. Hence, $(A_1 \cup \{v\}, A_2, \ldots, A_l)$ is an indifference order for G'_{i+1}.

If v is not twin with vertices of A_1, then it is not twin with any other vertex. Indeed, while $Adj(i, v) \subseteq N(i, A_1)$, all other vertices have neighbors outside $N(i, A_1)$. For this reason, v must start a new class by itself, which should be placed near A_1 to satisfy the consecutive requirement for the maximal clique $\{v\} \cup A_1 \cup \cdots \cup A_j$ of G_{i+1}. Then, $(\{v\}, A_1, \ldots, A_l)$ is an indifference order for G'_{i+1}.

Case 2: $A_j \not\subseteq Adj(i, v)$.

In this case, class A_j must be split. Its elements will no longer be all twins in G_{i+1}, since some of them meet v while others don't. Let $B = A_j \cap Adj(i, v)$ and $C = A_j \setminus Adj(i, v)$. Then $Adj(i, v) = A_1 \cup \cdots \cup A_{j-1} \cup B$ and $\{v\} \cup A_1 \cup \cdots \cup B$ is a maximal clique of G_{i+1}. The new classes B and C will replace A_j in the order, with B closer to A_1 because of v's maximal clique. Hence, $(\{v\}, A_1, \ldots, B, C, A_{j+1}, \ldots, A_l)$ is an indifference order for the reduced graph of G_i.

If we suppose $Adj(i, v) \cap A_1 = \emptyset$, the proof is analogous. $\square$

6 Linear-time implementation

Let n and m be the number of vertices and the number of edges of the input graph G, respectively. The algorithm proposed in Section 4 basically consists of a lexBFS followed by a loop whose body is executed at most $n-1$ times.

It is well-known that a lexBFS can be performed in $O(n+m)$ time [12]. Thus, it remains to show how to implement the loop so that the same time bound applies. This will be done by showing that each iteration consumes $O(1+|Adj(v)|)$, where v is the vertex tentatively added in this iteration. (The "$1+\ldots$" accounts for control operations.) Adding up for all iterations, this gives $O(n+m)$.

We begin with the description of the structure used to represent the equivalence classes $A_1,\ldots,A_l$. Each set A_r is represented by a doubly linked list of the vertices it contains, in arbitrary order. Pointers to the list heads are then stored on another doubly linked list, with the sets appearing in indifference order. We need doubly linked lists to be able to do deletions in constant time. We also keep track of the cardinality of $Adj(i,v)$ inside each class A_r. In addition, the number of classes contained in each one of the two outer cliques is computed. To accomplish that, we just need a variable b (for *beginning*) indicating that $A_1 \cup A_2 \cup \ldots \cup A_b = N(A_1)$, the maximal clique containing A_1, and a variable e (for *end*) indicating that $A_e \cup \ldots \cup A_{l-1} \cup A_l = N(A_l)$, the maximal clique containing A_l.

We are now ready to show how to implement the i-th iteration, which processes vertex v. Initially, all neighbors of v already in the structure are marked. This takes $O(1+|Adj(v)|)$. When a vertex is marked, a counter for marked vertices in its class is incremented. These counters must start with zero for each iteration. We will denote by m_r the value of this counter for class A_r.

Call a class *clean* if none of its vertices is marked, and *saturated* if all its elements are marked. Classes with both marked and unmarked vertices are called *mixed*. These concepts refer to the current iteration only. It is easy to express these states by means of the counters: "A_r is clean" is equivalent to $m_r = 0$, "A_r is saturated" to $m_r = |A_r|$, and so on.

The test $Adj(i,v) \cap A_1 \neq \emptyset$ is equivalent to $m_1 \neq 0$, and can be done in constant time. For the remaining conditions, we will assume that $m_1 \neq 0$. Otherwise we check the opposite end of the structure in an analogous manner.

We examine the counters for classes $A_1, A_2, \ldots$ in this order, stopping when at least one of the following conditions is met:

1. we reach an unsaturated class (that is, either clean or mixed.)

2. we reach class A_b.

Let k be the last class examined. We claim that v can be successfully added to the structure if and only if $k = j$, where j is defined as in the algorithm. To see this, observe that the condition $A_j \cap N(i, A_1) = \emptyset$ means $b < j$, which implies $k < j$, since $k \leq b$. The other condition tested in the algorithm, namely, $A_1 \cup A_2 \cup \cdots \cup A_{j-1} \not\subseteq Adj(i, v)$, implies $k \leq j - 1$. In either case, we have $k < j$. On the other hand, k is clearly not greater than j. If $k \neq j$, this is because $k < j$, that is, an unsaturated class or A_b is reached before all marked classes are examined, which implies one of the conditions tested in the algorithm.

Since we don't compute j, it is impossible to test equality to k directly. However, it is easy to see that $k = j$ is equivalent to the testable condition

$$\sum_{r=1}^{k} m_r = |Adj(i, v)|$$

Note that at most $|Adj(v)|$ classes are examined, staying within the allowed time bound. Also, during the examining phase, we can reset counters $m_1, m_2, \ldots, m_k$ to zero, preparing them for the next iteration. If v is added successfully, $k = j$ and those are the only counters modified in this iteration. If $k \neq j$, the loop will stop and the graph will be rejected anyway, so we don't have to worry about future iterations.

Finally, we need to specify how the structure update is done when v is added. The test $A_j \subseteq Adj(i, v)$ is equivalent to $m_k = |A_k|$ (recall that $k = j$.) If this is true, the test "$\exists w \in A_1$ with $N(i, v) = N(i, w)$" is equivalent to $k = b$. Adding a vertex to a class or creating a new singleton class are easily accomplished in constant time, as well as updating the cardinality counters. The potential problem here is splitting a mixed class in two, one containing the marked and the other the unmarked vertices. However, this can be done with the following trick. Each time a vertex is marked, it is transferred to the beginning of the list for its class. This way, when we need to split a class A_r, we know that the first m_r elements are the ones adjacent to v. This is a convenient way of marking vertices without using an extra bit field. It also obviates the need to clear the marks: zeroing the counter automatically does this.

A last remark concerns updating the outer clique indicators b and e. Except in the case when v is added to an existing class, b receives the value $k+1$, reflecting the new outer clique. Variable e is updated analogously when v is adjacent to A_l instead of A_1.

7 Conclusions

To the best of our knowledge, no algorithm for proper interval graph recognition has been published yet. The algorithm presented here is easy to understand, runs in linear time, and is capable of producing all indifference orders for the graph.

Interval graphs have been studied extensively, and various optimization problems can be solved in polynomial time for this class [3]. When restricted to proper interval graphs, these problems may admit simpler and/or faster algorithmic solutions. One such example is to find the minimum bandwidth [9]. It would be interesting to study these problems.

We are not convinced that the known algorithms for interval graph recognition are the best possible. The MPQ-tree algorithm is still complicated, and it is not clear how to get all consecutive clique arrangements with Simon's approach. We are currently working on extending two other recognition algorithms for proper interval graphs to interval graphs. These algorithms probably will yield simpler algorithms than the ones known to date.

References

[1] K. S. Booth and G. S. Lueker, Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms, *J. Comput. System Sci.* **13** (1976) 335–379. A preliminary version appeared in *Proc. 7th ACM Symposium on Theory of Computing* (1975) 255-265.

[2] P. C. Gilmore and A. J. Hoffman, A characterization of comparability graphs and interval graphs, *Canad. J. Math.* **16** (1964) 539–548.

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

[4] G. Hajós, Über eine Art von Graphen, *Internationale mathematische nachrichten* **11** (1957) problem 65.

[5] R. C. Hamelink, A partial characterization of clique graphs, *J. Comb. Theory* **5** (1968) 192–197.

[6] B. Hedman. Clique graphs of time graphs, *J. Comb. Theory B* **37** (1984) 270–278.

[7] N. Korte and H. Möhring, An incremental linear-time algorithm for recognizing interval graphs, *SIAM J. Comput.* **18** (1989) 68–81.

[8] H. Maehara, On time graphs, *Discrete Math.* **32** (1980) 281–289.

[9] R. Mahesh, C. Pandu Rangan, and A. Srinivasan, On finding the minimum bandwidth of interval graphs, *Information and Computation* **95** (1991) 218–224.

[10] F. S. Roberts, Indifference graphs, A characterization of comparability graphs and of interval graphs, in: F. Harary (ed.), *Proof Techniques in Graph Theory*, Academic Press, New York (1969) 139–146.

[11] F. S. Roberts, On the compatibility between a graph and a simple order, *J. Comb. Theory B* **11** (1971) 28–38.

[12] D. J. Rose, R. E. Tarjan and G. S. Leuker, Algorithmic aspects of vertex elimination on graphs, *SIAM J. Comput.* **5** (1976) 266–283.

[13] K. Simon, A new simple linear algorithm to recognize interval graphs, in: H. Bieri and H. Noltemeier (eds.), *Computational Geometry – Methods, Algorithms and Applications*, Springer-Verlag, LNCS **553** (1991) 289–308.

Relatórios Técnicos – 1992

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`