Discrete Mathematics 309 (2009) 5697–5702

Contents lists available at ScienceDirect

Discrete Mathematics

journal homepage: www.elsevier.com/locate/disc

A dynamic distributed approach to representing proper interval graphs

David Morgan

Camouflage Software Inc., St. John's NL A1A 1W7, Canada

ARTICLE INFO

Article history:
Received 12 July 2006
Accepted 22 April 2008
Available online 6 June 2008

Keywords:
Dynamic graph
Proper interval graph
Distributed computing
Informative labeling
Implicit representation

ABSTRACT

First studied by Brodal and Fagerberg [G.S. Brodal, R. Fagerberg, Dynamic representation of sparse graphs, in: Algorithms and Data Structures, Proceedings of the 6th International Workshop, Vancouver, Canada, in: Lecture Notes in Computer Science, vol. 1663, Springer-Verlag, 1999], a *dynamic adjacency labelling scheme* labels the vertices of a graph so that the adjacency of two vertices can be deduced from their labels. The scheme is dynamic in the sense that only a small adjustment must be made to the vertex labels when a small change is made to the graph.

Using a centralized dynamic representation of Hell, Shamir and Sharan [P. Hell, R. Shamir, R. Sharan, A fully dynamic algorithm for recognizing and representing proper interval graphs, SIAM Journal on Computing 31 (1) (2001) 289–305], we develop a $O(\log n)$ bit/label dynamic adjacency labelling scheme for proper interval graphs. Our fully dynamic scheme handles vertex deletion/addition and edge deletion/addition in $O(n)$ time. Furthermore, our dynamic scheme is *error-detecting*, as it recognizes when the new graph is not a proper interval graph.

1. Introduction

Consider a finite simple undirected graph $G = (V_G, E_G)$ on n vertices. To distinguish between the vertices, we often label them from 1 to n, yet such labels tell us nothing about the graph. If we allow more complicated labels, it is possible to determine the structure of the graph exclusively from the labels, thereby eliminating the need for a global representation, such as an adjacency matrix.

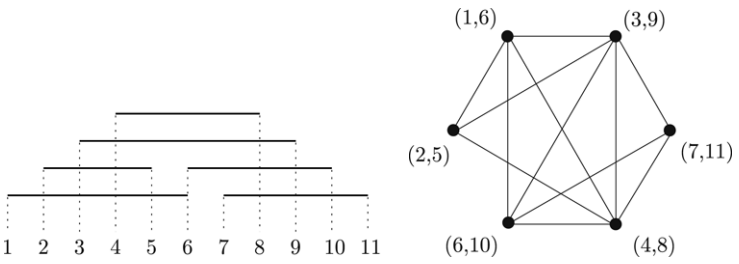
By introducing the distributed data structure known as an adjacency labelling scheme, Muller [15] and Kannan, Naor, and Rudich [8] offer a local representation of G that allows the adjacency of two vertices to be determined using only their labels. Specifically, an *adjacency labelling scheme* of a family $\mathcal{G}$ of finite graphs is a pair $(\mathcal{M}, \mathcal{D})$, defined as follows.

- $\mathcal{M}$, the marker, is a vertex labelling algorithm whose input is a member of $\mathcal{G}$.
- $\mathcal{D}$, the decoder, is a polynomial time evaluation algorithm that correctly determines the adjacency of two vertices using only their labels (we will say that $\mathcal{D}$ is adjacency-correct).

For example, consider the following adjacency labelling scheme for interval graphs [15]. Recall that a graph is said to be an *interval graph* if each vertex can be represented by an interval of reals such that two vertices are adjacent if and only if the corresponding intervals have non-empty intersection. Any such interval representation can be mapped to another interval representation using closed intervals with endpoints in $\{1, \ldots, 2n\}$. The marker labels each vertex with the two endpoints of its associated interval while the decoder determines adjacency in $O(1)$ time by comparing these integers just as it would two intervals. Each label requires $O(\log n)$ bits, therefore the entire labelling uses $O(n \log n)$ bits. An example of an adjacency labelling of an interval graph is given in Fig. 1.

E-mail address: dmorgan@datamasking.com.

0012-365X/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.disc.2008.04.039



(a) An interval representation of a graph. (b) The corresponding labels.

Fig. 1. An adjacency labelling of an interval graph.

In many applications, the underlying topology is subject to frequent small changes, however, adjacency labelling schemes lack a mechanism to adapt to these small changes. A *dynamic adjacency labelling scheme* of a family $\mathcal{G}$ of finite graphs is a tuple $(\mathcal{M}, \mathcal{D}, \Delta, \mathcal{R})$, defined as follows.

- $(\mathcal{M}, \mathcal{D})$ is an adjacency labelling scheme of $\mathcal{G}$.
- Δ is a set of dynamic graph operations.
- $\mathcal{R}$, the relabeller, is a polynomial time relabelling algorithm which, using only a vertex labelling, maintains an adjacency-correct labelling when a dynamic graph operation in Δ acts on a member of $\mathcal{G}$, providing the operation produces another graph in $\mathcal{G}$. Moreover, we say that the dynamic scheme is *error-detecting* if $\mathcal{R}$ can determine when a dynamic graph operation produces a graph that does not belong to $\mathcal{G}$. Note that $\mathcal{R}$ can be considered as the composition of several "smaller" relabelling algorithms, one for each operation in Δ.

As an example, consider the following dynamic adjacency labelling scheme for rooted trees which allows the addition and deletion of vertices. For simplicity, let us require that the root not be deleted. The marker first assigns each vertex a unique identifier chosen from $\{1, \ldots, n\}$; we will refer to each vertex by its identifier. The marker then labels each non-root vertex v as $(v, \text{parent}(v))$, and the root vertex, r, as $(r, 0)$. Given the labels of two vertices, u and v, the decoder deems u and v adjacent if and only if $u = \text{parent}(v)$ or $v = \text{parent}(u)$. In adding a new vertex, the relabeller chooses an identifier x, the smallest available natural number, then labels the new vertex $(x, \text{parent}(x))$. In deleting a vertex, the relabeller simply deletes its label from storage. Each relabelling can be performed in $O(1)$ time (throughout this work we assume a word-level RAM computation model for the marker, decoder, and relabeller, where word sizes are $\Omega(\log n)$). Unfortunately, this dynamic scheme is not error-detecting, because we cannot tell if the deletion of a vertex creates a disconnected forest; however, we can make the scheme error-detecting by adding, to each label, a counter that keeps track of how many children the vertex has. Note that we can tell if the root is being deleted as $\text{parent}(v) = 0 \Longrightarrow v = r$.

Although this dynamic adjacency labelling scheme for trees seems straightforward, there are two underlying problems.

1. It is possible to delete too many vertices, thereby causing the remaining labels to be too large (the point at which one decides that the labels are intolerably large depends on the application, as well as the family under consideration).
2. When a vertex is added and given an identifier, the relabeller must determine an acceptably short unused identifier to assign to it.

Given that we are approaching dynamic adjacency labelling schemes from a theoretical standpoint, we make certain assumptions to eliminate the problems discussed above. Specifically, we assume the following.

1. If n is the number of vertices presently in the graph, then there exists some constant k such that there has never been more than n^k vertices in the graph.
2. If an identifier is needed, a marker or relabeller can obtain the smallest available identifier in $O(1)$ time.

The validity of such assumptions is highly dependent on the application in which the dynamic scheme is being used. In our case, we do not want the restrictions of the application to hinder the development of the scheme. It is hoped that our dynamic scheme can be modified to work in different applications, with adjusted label sizes and running times as appropriate.

Observe that analogous schemes can be defined by replacing adjacency with any function defined on sets of vertices, for example, the distance between two vertices. By setting adjacency labelling schemes in the larger context of *informative labelling schemes*, Peleg [16] rejuvenated interest in the idea of space efficient distributed data structures as introduced by Muller [15] and Kannan et al. [8]. To date, informative labelling schemes have been developed for functions such as distance, routing, center of three vertices, ancestor, and nearest common ancestor. Detailed discussions on the application of informative labelling schemes to XML search engines and communication networks can be found in a survey paper of Gavoille and Peleg [5].

The dynamic version of adjacency labelling schemes was mentioned in the seminal paper of Kannan et al. [8], however, the authors did not consider the problem in detail. The first paper to address this dynamic problem was that of Brodal and Fagerberg [1], who developed a dynamic adjacency labelling scheme for graphs of bounded arboricity. More recently, Korman and Peleg [9] and Korman, Peleg, and Rodeh [10] have considered dynamic distance labelling schemes for trees, in

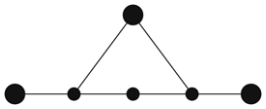
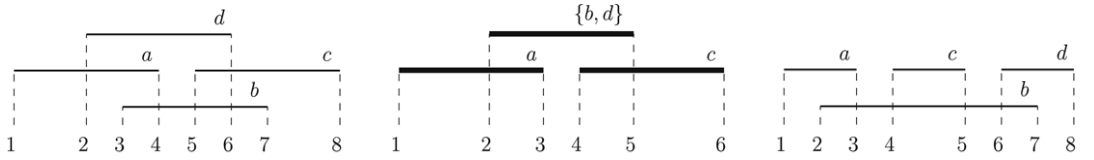


Fig. 2. An astral triple. The bold vertices indicate the astral triple.



(a) A proper interval representation. (b) Blocks as "mega-intervals". (c) An interval representation, which is not, and cannot be, proper.

Fig. 3. Interval representations and blocks.

the context of distributed computing. Cohen, Kaplan, and Milo [2] consider dynamic ancestor labellings of XML trees with persistent labels, that is, the label of a vertex cannot be changed once it has been assigned. As well, Morgan [13] develops a dynamic adjacency labelling scheme for line graphs, using a technique that employs graph substructures and circular doubly linked lists to distribute information about the neighbourhood of a vertex across the labels of the neighbours.

By extending the technique of Morgan [13] to blocks, we are able to use the work of Hell, Shamir, and Sharan [6] to develop dynamic schemes for proper interval graphs. In Section 3, we develop an error-detecting dynamic adjacency labelling scheme for proper interval graphs that allows the addition and deletion of vertices and edges. The labels used in this scheme require $O(\log n)$ bits, and updates can be performed in $O(n)$ time. In comparison, the best known (static) adjacency labelling scheme for proper interval graphs is the scheme previously presented for interval graphs [15], which uses $O(\log n)$ bit labels and requires as much as $\Theta(n + m)$ time to generate a labelling (here we presume that the marker is input with only the proper interval graph, perhaps as an adjacency matrix, and must use a $\Theta(n + m)$ time algorithm like that of Corneil, Kim, Natarjan, Olariu, and Sprague [3] to determine the proper interval representation from the graph itself.) Proper interval graphs have been shown useful in the study of problems in genetics and psychology; a good starting point for information on the application of proper interval graphs is the text of McKee and McMorris [12].

2. Proper interval graphs, blocks, and straight enumerations

A graph is a proper interval graph if it has an interval representation in which no interval contains another interval. Proper interval graphs can also be characterized using structures known as astral triples. An astral triple is a set of three vertices for which each pair are connected by a path in which no two consecutive vertices belong to the neighbourhood of the third vertex. In the graph shown in Fig. 2, the bold vertices form an astral triple.

Perhaps the simplest example of an astral triple is $K_{1,3}$, often referred to as a claw. In the case of $K_{1,3}$, the three pendant vertices form the astral triple. The relationship between proper interval representations and astral triples is explicitly addressed in the following theorems.

Theorem 2.1 ([7]). *A graph is a proper interval graph if and only if it contains no astral triple.*

Theorem 2.2 ([17]). *An interval graph is proper if and only if it contains no induced $K_{1,3}$.*

Another characterization of interval graphs is based on the notion of blocks [4]. Consider the equivalence relation R determined by uRv if and only if $N[u] = N[v]$. This equivalence relation partitions the vertices into equivalence classes known as *blocks*. For example, the blocks of the proper interval graph represented in Fig. 3 are $\{a\}$, $\{b, d\}$, and $\{c\}$. Ultimately, we can consider each block as a "mega-interval", as depicted in Fig. 3(b).

Two blocks B and B' are said to be adjacent if there exists an edge bb' for which b in B and b' in B' (furthermore, if B is adjacent to B', then, for all b in B and all b' in B', b is adjacent to b'). In an extension of conventional graph terminology, we say that the (open) neighbourhood of a block is the set of blocks that are adjacent to it, and that its closed neighbourhood is its open neighbourhood unioned with itself. Similarly, we say that the degree of a block is the cardinality of its open neighbourhood, where $\deg(B)$ denotes the degree of B.

As preliminary observations, consider the following properties of blocks.

Lemma 2.3. *The induced subgraph formed on the vertices of a block is a clique.*

Proof. Consider any two vertices u and v belonging to the same block of a graph. By definition, $N[u] = N[v]$, therefore, u and v are adjacent. The result follows. □

Lemma 2.4. *No component can be comprised of only two blocks.*

Proof. Consider a component C consisting of two blocks B_1 and B_2. If B_1 is not adjacent to B_2, then B_1 is a component itself, thereby, C is not a component. If B_1 is adjacent to B_2, then $B_1 \cup B_2$ forms a clique, therefore, $B_1 = B_2$, which is also a contradiction. □

Lemma 2.5. *No two blocks can be adjacent to the same set of blocks.*

Proof. Consider two blocks B_1 and B_2, which are adjacent to the same blocks. For any vertices b_1 in B_1 and b_2 in B_2, $N[b_1] = N[b_2]$. Therefore, $B_1 = B_2$, which is a contradiction. □

A *straight enumeration* of a graph is a linear ordering of its blocks such that, for every block, the blocks in its closed neighbourhood are consecutive. In the case of the proper interval graph represented in Fig. 3(a), the straight enumerations are $\Phi = \{a\} \prec \{b, d\} \prec \{c\}$ and $\Phi^R = \{c\} \prec \{b, d\} \prec \{a\}$, where Φ^R denotes the reversal of the straight enumeration Φ. The following theorem characterizes proper interval graphs in terms of straight enumerations.

Theorem 2.6 ([4]). *A graph is a proper interval graph if and only if it has a straight enumeration. Moreover, a connected proper interval graph has a unique straight enumeration (up to reversal).*

Hell, Shamir, and Sharan [6], on whose work we will heavily rely, refer to a straight enumeration of a connected proper interval graph as a *contig*.

Finally, fundamental to any work involving block representations of proper interval graphs is the following lemma, referred to as the "umbrella property".

Lemma 2.7 ([11]). *Consider a straight enumeration Φ of a connected proper interval graph G. If B_1, B_2, and B_3 are blocks of G, such that $B_1 \prec B_2 \prec B_3$ in Φ and B_1 is adjacent to B_3, then B_2 is adjacent to B_1 and to B_3.*

3. A dynamic scheme for proper interval graphs

3.1. Vertex labels, marker, and decoder

Our scheme closely resembles a dynamic representation of proper interval graphs due to Hell, Shamir, and Sharan [6]; however, their representation is not designed to allow implicit adjacency testing from vertex labels. For each component of the proper interval graph, they maintain a data structure to represent a contig. In each contig, the first and last blocks are called *end blocks* and their members are *end vertices*; all other blocks are referred to as *inner blocks* and their members are *inner vertices*. Specifically, the data structure used in the fully dynamic scheme of Hell, Shamir, and Sharan consists of the following.

- For each vertex, they maintain the name of its block.
- For each block, they maintain the following information.
  - The size of the block.
  - Left and right *near pointers* which point to the adjacent blocks immediately to the left and right, respectively, in the straight enumeration.
  - Left and right *far pointers* which point to the furthest adjacent blocks to the left and right, respectively, in the straight enumeration.
  - Left and right *self pointers* which point to the block itself.
- For each connected component, they maintain the constituent vertices.

Unfortunately, we do not have the liberty of using pointers at the block level, rather, we must do so at the vertex level. To this effect, we select a *pointer vertex* $P(B)$ from each block B. If we wish to include a pointer Q from block B to block B', then we include that pointer in the label of $P(B)$, such that $Q(P(B)) = b'$, where $b' \in B'$. In essence, we create a "distributed" pointer.

Specifically, our labelling scheme is as follows.

- For each vertex v, we maintain the following.
  - A unique identifier for each vertex. Where $\mathcal{L}$ is the size of the largest identifier in the current graph, and n^* is the maximum number of vertices that have existed in the dynamic graph, the uniqueness of the identifiers ensures that $\mathcal{L} \in \Omega(\log n^*)$ bits. Given the assumption that the smallest available identifier can be obtained in $O(1)$ time, as stated in Section 1, we guarantee that $\mathcal{L} \in O(\log n^*)$, thereby $\mathcal{L} \in \Theta(\log n^*)$. Our other assumption on the size of identifiers, also stated in Section 1, gives $\mathcal{L} \in \Theta(\log n)$.
  - The identifier of the block to which it belongs. Although we do not differentiate between a vertex and its identifier, we will differentiate between a block and its identifier, as the identifier of a block may change over time while we maintain a straight enumeration. To this effect, we denote the block containing v by $B(v)$, and denote the identifier of $B(v)$ by $b(v)$.

    Just as we required a unique identifier for each vertex, we require a unique identifier for each block. Where $\mathcal{B}$ is the size of the largest identifier, we ensure that $\mathcal{B} \in O(\log n)$.

  - The identifiers of the furthest adjacent blocks to the left and right of $B(v)$, denoted $f_L(v)$ and $f_R(v)$, respectively. This information requires $O(\mathcal{B})$ bits.
- For each block B, we encode the following information via the vertex labels.
  - The vertices in each block. This information is represented using a circular doubly linked list of the vertices in each block. This circular doubly linked list adds $O(\mathcal{L})$ bits to the label of each vertex. Again, we will say that we traverse B, although we really mean that we traverse the circular doubly linked list of vertices in B.
  - A pointer vertex, denoted $P(B)$. The label of the pointer vertex must contain a bit to denote that it is a pointer vertex. All other vertices in the block, contain the identifier of $P(B)$, as well as a bit to denote that they are not the pointer vertex of B. This information adds $O(1)$ bits to the label of the pointer vertex, and $O(\mathcal{L})$ bits to the label of all other vertices.

    To clarify how these distributed pointers are used at the vertex level, let us consider a pointer Q and a vertex v. The label of v will contain the identifier of $P(B(v))$, the pointer vertex of the block containing v (assuming $v \neq P(v)$); for simplicity, we will shorten $P(B(v))$ to $P(v)$. The label of $P(v)$ will contain the identifier of $Q(P(v))$, which we will similarly shorten to $Q(v)$. For any vertex v and pointer Q, $Q(v)$ can be "followed" in $O(1)$ time using the labels of v and $P(v)$.
  - A pointer to the blocks immediately to the left and right of B, denoted by $I_L(B)$ and $I_R(B)$, respectively. Both $I_L(B)$ and $I_R(B)$ are artificial constructs, as they are achieved by including I_L and I_R pointers in the label of $P(B)$, as per the pointer technique described above. These pointers add $O(\mathcal{L})$ bits to the label of the pointer vertex only.
  - A pointer to the furthest adjacent blocks to the left and right of B, denoted by $F_L(B)$ and $F_R(B)$, respectively. This is achieved using the pointer technique described above. These pointers add $O(\mathcal{L})$ bits to the label of the pointer vertex only.
  - The size of B, denoted $s(B)$. This value is kept in the label of the pointer vertex, adding $O(\log n)$ bits to its label.

We observe that the total size for each label is $O(\mathcal{L} + \mathcal{B}) \in O(\log n)$. Furthermore, we can determine the adjacency of two vertices u and v in $O(1)$ time, using only their labels, by evaluating $f_L(v) \leq b(u) \leq f_R(v)$.

Given our use of distributed pointers and circular doubly linked lists, our labelling scheme is functionally identical to that of Hell, Shamir, and Sharan; however, we should address two additional points which might not be entirely obvious. Firstly, our vertex labels do not include self pointers because they are obsolete in our vertex centered setting; in essence, each pointer vertex can be considered to point to itself, so each block implicity has a distributed self pointer. Secondly, although we do not maintain the set of connected components, we can still determine if two vertices belong to the same component, which is the reason Hell et al. maintain this information. By maintaining connected components, they can determine if two vertices are in the same component in $O(\log n)$ time; in contrast, we must traverse far left and far right pointers ($F_L(B)$ and $F_R(B)$), possibly requiring $\Theta(n)$ time. The use of connected components by Hell et al. is critical in establishing the $O(d + \log n)$ bound on update times for their fully dynamic scheme, where d is the number of edges added to, or deleted from, the dynamic graph. In our distributed scheme, we cannot globally represent the connected components; more importantly, we choose not to represent them at the vertex level because we cannot improve our $O(n)$ time bound by doing so. In Section 3.2, observerations regarding updates of b, f_L, and f_R values will show that the lack of information about connected components is not the only limiting aspect of our scheme.

As a final difference, consider that for every block B, Hell et al. point to the adjacent blocks immediately to the left and right of B, whereas we include a similar pointer that omits the adjacency condition. By dropping the adjacency condition we are able to maintain additional information about distinct contigs in the straight enumeration without sacrificing our $O(\log n)$ bound on label size, nor our $O(n)$ bound on update time.

Although it is much easier to discuss pointers and values at a block level, we must always ensure that these items can be observed at the vertex level. For instance, a vertex v is an end vertex if and only if $F_L(B(v)) = B(v)$ or $F_R(B(v)) = B(v)$. However, to determine this condition, we must check to see if $f_L(v) = b(v)$ or $f_R(v) = b(v)$. As previously discussed, $F_L(B(v))$ is an artificial construct.

Although we have given significant consideration to the labels of the dynamic scheme, we have not yet discussed the marker. Deng, Hell, and Huang [4] provide an $O(n + m)$ time algorithm for generating a straight enumeration of an interval graph from an adjacency matrix representation (actually, their algorithm presents a vertex ordering, however, minor bookkeeping will give a straight enumeration of blocks). Where $\mathbf{B}$ is the number of blocks in the straight enumeration, we can use the straight enumeration to establish the $\mathbf{B}$ circular doubly linked lists in $\Theta(n)$ time. Next, establishing pointer vertices and block identifiers, as well as the b, f_L, and f_R values requires a further $O(n)$ time. Finally, establishing the various pointers requires an additional $\Theta(\mathbf{B}) \in O(n)$ time. Therefore, if provided with the straight enumeration, the marker requires $\Theta(n)$ time; otherwise, the marker requires $O(n + m)$ time.

3.2. Relabeller

Given that our distributed scheme is functionally equivalent to the fully dynamic scheme of Hell, Shamir, and Sharan, our relabeller follows directly from the algorithms they derive for dynamic updates. That being said, our upper bound of $O(n)$ on update time is not as tight as their $O(d + \log n)$ bound; this gap requires some further explanation.

Due to the linear nature of the straight enumeration, the limiting factor inherent in our labelling scheme is the maintenance of the b, f_L and f_R values that are used to implicitly determine adjacency. Specifically, these values must be

constantly updated to guarantee that they can be represented using $O(\log n)$ bits. When the graph is modified, our first task is to modify blocks, pointer vertices, and pointers, as necessary, in order to maintain a straight enumeration. Once this is complete, we can traverse I_L and I_R pointers to determine the entire straight enumeration. Knowing the entire straight enumeration, one pass through the ordering (from least to greatest) is sufficient to re-assign optimal block identifiers by traversing the circular linked list of vertices in each block. Having assigned these optimal block identifiers, a second pass is sufficient to assign the f_L and f_R values, which depend on the block identifiers, to the vertices in each block.

Regardless of the graph operation under consideration, the maintenance of the b, f_L, and f_R values takes as much as $\Theta(n)$ time. Because this approach can be used to maintain optimal b values, we do not employ the assumption on the size of the identifiers, as stated in Section 1, to the size of the largest block identifier.

Readers interested in seeing relabelling details are encouraged to consult the doctoral thesis of Morgan [14]; however, it should be noted that the scheme detailed in the doctoral thesis is considered in comparison to the incremental algorithms of Hell, Shamir, and Sharan, which employ the use of "end" pointers.

4. Conclusion

We have applied a distributed pointer technique, along with a distributed circular doubly linked list technique of Morgan [13], to develop an error-detecting dynamic adjacency labelling scheme for proper interval graphs. Our fully dynamic scheme, which is largely based on a centralized scheme of Hell, Shamir, and Sharan [6], uses $O(\log n)$ bit labels and handles all updates in $O(n)$ time. It is hoped that the development of an $O(\log n)$ bit/label dynamic adjacency labelling scheme for interval graphs might benefit from the techniques used in this paper. As an interim goal, one might establish an $O(\log n)$ bit/label dynamic adjacency labelling scheme for proper interval graphs that handles all operations in $o(n)$ time; such a result would close the time gap between this work, and that of Hell, Shamir, and Sharan [6].

Acknowledgement

The author was partially supported by the Alberta Informatics Circle of Research Excellence and the Department of Computing Science, University of Alberta, Edmonton AB T6G 2E8, Canada.

References

[1] G.S. Brodal, R. Fagerberg, Dynamic representation of sparse graphs, in: Algorithms and Data Structures, Proceedings of the 6th International Workshop, Vancouver, Canada, in: Lecture Notes in Computer Science, vol. 1663, Springer-Verlag, 1999.
[2] E. Cohen, H. Kaplan, T. Milo, Labeling dynamic XML trees, in: Proceedings of the 21st ACM Symposium on Principles of Database Systems, Madison, USA, ACM, 2002.
[3] D.G. Corneil, H. Kim, S. Natarajan, S. Olariu, A.P. Sprague, Simple linear time recognition of unit interval graphs, Information Processing Letters 55 (1995) 99–104.
[4] X. Deng, P. Hell, J. Huang, Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs, SIAM Journal on Computing 25 (1996) 390–403.
[5] C. Gavoille, D. Peleg, Compact and localized distributed data structures, Journal of Distributed Computing 16 (2003) 111–120.
[6] P. Hell, R. Shamir, R. Sharan, A fully dynamic algorithm for recognizing and representing proper interval graphs, SIAM Journal on Computing 31 (1) (2001) 289–305.
[7] Z. Jackowski, A new characterization of proper interval graphs, Discrete Mathematics 105 (1992) 103–109.
[8] S. Kannan, M. Naor, S. Rudich, Implicit representation of graphs, SIAM Journal on Discrete Mathematics 5 (4) (1992) 596–603.
[9] A. Korman, D. Peleg, Labeling schemes for weighted dynamic trees, in: Automata, Languages and Programming, Proceedings of the 30th International Colloquium, Eindhoven, The Netherlands, in: Lecture Notes in Computer Science, vol. 2719, Springer-Verlag, 2003.
[10] A. Korman, D. Peleg, Y. Rodeh, Labeling schemes for dynamic tree networks, Theory of Computing Systems 37 (2004) 49–75.
[11] P.J. Looges, S. Olariu, Optimal greedy algorithms for indifference graphs, Computers and Mathematics with Applications 25 (1993) 15–25.
[12] T.A. McKee, F.R. McMorris, Topics in Intersection Graph Theory, in: SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia, 1999.
[13] D. Morgan, A dynamic implicit adjacency labelling scheme for line graphs, in: Algorithms and Data Structures, Proceedings of the 9th International Workshop, Waterloo, Canada, in: Lecture Notes in Computer Science, vol. 3608, Springer-Verlag, 2005.
[14] D. Morgan, Dynamic adjacency labelling schemes, Ph.D. Thesis, University of Alberta, October 2006.
[15] J.H. Muller, Local structure in graph classes, Ph.D. Thesis, Georgia Institute of Technology, March 1988.
[16] D. Peleg, Informative labeling schemes for graphs, in: Mathematical Foundations of Computer Science, Proceedings of the 25th International Symposium, Bratislava, Slovakia, in: Lecture Notes in Computer Science, vol. 1893, Springer-Verlag, 2000.
[17] F.S. Roberts, Indifference graphs, in: F. Harary (Ed.), Proof Techniques in Graph Theory, Academic Press, New York, 1969, pp. 139–146.