Arboricity, h-Index, and Dynamic Algorithms[☆]

Min Chih Lin[a,2], Francisco J. Soulignac[b,3], Jayme L. Szwarcfiter[c,4]

[a] *CONICET and Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Instituto de Cálculo and Departamento de Computación, Buenos Aires, Argentina.*
[b] *Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación, Buenos Aires, Argentina.*
[c] *Universidade Federal do Rio de Janeiro, Instituto de Matemática, NCE and COPPE, Caixa Postal 2324, 20001-970 Rio de Janeiro, RJ, Brasil.*

Abstract

We propose a new data structure for manipulating graphs, called h-graph, which is particularly suited for designing dynamic algorithms. The structure itself is simple, consisting basically of a triple of elements, for each vertex of the graph. The overall size of all triples is $O(n + m)$, for a graph with n vertices and m edges. We describe algorithms for performing the basic operations related to dynamic applications, as insertions and deletions of vertices or edges, and adjacency queries. The data structure employs a technique first described by Chiba and Nishizeki [Chiba and Nishizeki: Arboricity and Subgraph Listing Algorithms, SIAM J. Comput. 14(1), pp. 210–223 (1985)], and relies on the arboricity of graphs. Using the proposed data structure, we describe several dynamic algorithms for solving problems as listing the cliques of a given size, recognizing diamond-free graphs, and finding simple, simplicial and dominated vertices. These algorithms are the first of their kind to be proposed in the literature. In fact, the dynamic algorithms for the

 DOI: 10.1016/j.tcs.2011.12.006

Email addresses: oscarlin@dc.uba.ar (Min Chih Lin), fsoulign@dc.uba.ar (Francisco J. Soulignac), jayme@nce.ufrj.br (Jayme L. Szwarcfiter)

[1]Partially supported by UBACyT Grants X456 and X143, and PICT ANPCyT Grant 1562.

[2]Partially supported by UBACyT Grant X456 and PICT ANPCyT Grant 1562.

[3]Partially supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico, CNPq, and Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro, FAPERJ, Brasil.

above problems lead directly to new static algorithms, and using the data structure we also design new static algorithms for the problems of counting subgraphs of size 4, recognizing cop-win graphs and recognizing strongly chordal graphs. The complexities of all of the proposed static algorithms improve over the complexities of the so far existing algorithms, for graphs of low arboricity. In addition, for the problems of counting subgraphs of size 4 and recognizing diamond-free graphs, the improvement is general.

Keywords: arboricity, cop-win graphs, data structures, diamond-free graphs, dynamic algorithms, h-index, strongly chordal graphs.
2010 MSC: 68P05, 05C85

---

1. Introduction

We describe a variation of a technique by Chiba and Nishizeki [4], leading to a data structure for graph algorithmic problems, called the *h-graph* data structure. It supports operations of insertion and removal of vertices, as well as insertion and removal of edges. Although the data structure can be used for general purpose, it is particularly suitable for applications in dynamic graph algorithms.

As an application of this data structure, we describe dynamic algorithms for several graph problems. We remark that no previous dynamic algorithms exist so far in the literature for the considered problems. On the other hand, in most cases, the proposed dynamic algorithms are also competitive as static algorithms, in the sense that there is also an improvement in time complexity (relative to the existing static graph algorithms) for graphs of low arboricity.

Basically, the proposed data structure consists of a triple for each vertex v of the graph. The first element of the triple is the degree of v, while the other two form a partition of the neighbors of v. The second element consists of a family of non-empty subsets of vertices, each one corresponding to the neighbors of v having degree exactly i, for each i less than the degree of v. The last element of the triple is the set of neighbors of v, having degrees at least the degree of v. The contents of each of the triples is maintained throughout the process, while vertices and edges are being inserted or removed. We show that the complexity of performing the dynamic operations of insertions and removals is strongly related to the arboricity and to the h-index of a graph.

A dynamic data structure designed for graphs with low h-index has been first defined by Eppstein and Spiro [8]. Such a data structure keeps, for

each graph G with h-index h, the set of vertices with degree at least h, and a dictionary that indicates the number of two-edge paths between any pair of vertices, for all those vertices at distance 2. The total size of the data structure is $O(mh)$ bits. Our h-graph data structure follows a different approach. First, we store no more than the adjacency lists of G in a special format, using $O(n + m)$ bits. Second, we do not compute the h-index of G.

In order to have an idea of the differences between these two data structures, refer to the problem of listing the triangles of a graph G. Using the data structure of [8], Eppstein and Spiro show that it is possible to maintain the family of triangles of G in $O(h)$ randomized amortized time while edges are inserted or removed. They also show how to keep other statistics of G with this data structure. By employing the data structure proposed in the present paper, we can maintain the family of triangles of G in $O(dh)$ deterministic worst case time per vertex insertion or removal, where d is the degree of the vertex. Furthermore, the time required by this algorithm when applied to all the vertices of the graph, so as to compute the family of triangles of G, is $O(\alpha m)$, where $\alpha \leq h$ is the arboricity of G. The disadvantage is that we can no longer maintain the triangles as efficiently as Eppstein and Spiro when edge operations are allowed. So, though both data structures have some similarities in their conceptions, they are better suited for different applications. In particular, our data structure allows efficient examination of the subgraph of G induced by the neighborhood of an inserted or removed vertex.

One of the similarities between the h-graph data structure and the data structure by Eppstein and Spiro, is that both differentiate between low and high degree vertices. The technique of handling differently vertices of high and low degree has been first employed by Alon et al. [1], and since then many other works made use of this technique (e.g. [8, 11]). In the present paper, we employ a variation of it, in the sense that the classification of each vertex into low or high is local. So, some vertices can be considered as both high and low depending on the local classification of each of its neighbors.

In the present paper, we apply the proposed data structure to formulate algorithms for the following seven problems.

1. Listing all the cliques of size k.
2. Counting the number of induced subgraphs, isomorphic to any desired graph having four vertices.
3. Counting the number of K_4's, diamonds, paws and claws, containing a

given vertex.

4. Recognizing diamond-free graphs and finding the maximal cliques of diamond-free graphs.
5. Finding simple, simplicial and dominated vertices of a graph.
6. Recognizing cop-win graphs, and finding a dismantling ordering of the vertices of a graph.
7. Recognizing strongly chordal graphs, and finding a simple elimination ordering of a graph.

For Problems 1 and 3–5, we propose dynamic algorithms, obtained by employing the h-graph data structure. The dynamic algorithms are the first in the literature for the considered problems. The degree of dynamics varies from application to application. For instance, for the problem of listing subgraphs of size k, we only consider insertion of vertices (in a sense, this is the natural case), whereas for the recognition of diamond-free graphs, we describe a fully dynamic algorithm, with insertions and deletions of both vertices and edges.

Table 1 describes the time complexities of the dynamic updates of the proposed dynamic algorithms, after the insertion or removal of a vertex v of degree d, of a graph G with n vertices, m edges, arboricity α and h-index h. For instance, if all the cliques of size k of G have already been listed, then $O(kdh\alpha^{k-3})$ time is required to include all cliques of size k containing v, of the graph $G \cup \{v\}$, into the list. Also, if G is diamond-free, then we can decide in $O(dh)$ time whether $G \cup \{v\}$ remains diamond-free.

We have also employed the h-graph data structure to design new static algorithms for all the applications above enumerated. Table 2 describes the time complexities of the proposed static algorithms to solve these problems, for connected graphs with n vertices, m edges, arboricity α and where n^ω is the time required for multiplying two $n \times n$ matrices. The table compares the complexities of the new algorithms, with those of the currently best static algorithms existing. It shows that the algorithms here proposed are competitive with the existing ones. That is, for graphs with low arboricity the new algorithms improve the complexity for all the considered problems, except for that of listing cliques, where the complexity matches. In some cases, the new algorithms actually decrease the complexities of the existing ones.

The paper is organized as follows. There are two distinct parts. In the first part, we describe the data structure, while the second one is devoted

Problem	**Inserting vertex v**	**Removing vertex v**
Listing all cliques of size 4	$O(kdh\alpha^{k-3})$	-
Recognizing diamond-free graphs	$O(dh)$	$O(dh)$
Finding dominated vertices	$O(dh)$	$O(dh)$
Finding simplicial vertices	$O(dh)$	$O(dh)$
Finding simple vertices	$O(m)$	$O(m)$

Table 1: Time complexities of the dynamic updates.

to various applications. In the next section we introduce the notation and terminology employed. In Section 3 we present the lemmas related to the complexity analysis of the basic operations of the h-graph data structure. The h-graph data structure itself is described in Section 4, together with the operations that it supports. Section 5 contains a more detailed description of the operations supported by the data structure, some implementation details and the complexity analysis of these operations. The remaining sections describe the applications. The problems above enumerated are then respectively solved in Sections 6 through 10. Some additional remarks, including the description of other applications form the last section.

2. Preliminaries

We work with undirected simple graphs. Let G be a graph with vertex set $V(G)$ and edge set $E(G)$, and call $n = |V(G)|$ and $m = |E(G)|$. Write vw to denote the edge of G formed by vertices $v, w \in V(G)$. For $v \in V(G)$, represent by $N_G(v)$ the subset of vertices adjacent to v, and let $N_G[v] = N_G(v) \cup \{v\}$. The set $N_G(v)$ is called the *neighborhood* of v, while $N_G[v]$ is the *closed neighborhood* of v. The *edge-neighborhood* of v, denoted by $N'_G(v)$, is the set of edges whose both endpoints are adjacent to v. Similarly, the *neighborhood* $N_G(vw)$ of an edge vw is the set of vertices that are simultaneously adjacent to both v and w. All the vertices in $N_G(vw)$ are said to be *edge-adjacent* to vw. The *degree* of v is $d_G(v) = |N_G(v)|$, the *degree*

Problem	Existing algorithms	Proposed algorithms
Listing all cliques of size k	$O(k\alpha^{k-2}m)$ [4]	$O(k\alpha^{k-2}m)$
Counting subgraphs of size 4	$O(n^\omega + min\{m^{1.61}, \alpha^2 m\})$ [11]	$O(min\{m^{1.61}, \alpha^2 m\})$
Recognizing diamond-free graphs	$O(m^{3/2})$ [7]	$O(\alpha m)$
Finding Simple simplicial and dominated vertices	$O(m^{1.41})$ [11]	$O(\alpha m)$
Recognizing cop-win graphs	$O(nm)$ [19] $O(n^3/\log n)$ [19]	$O(\alpha m)$
Recognizing strongly chordal graphs	$O(n^2)$ [17] $O(m \log n)$ [14]	$O(\alpha m)$

Table 2: Complexities of the static algorithms for connected graphs.

of vw is $d_G(vw) = |N_G(vw)|$, and the *edge-degree* of v is $d'_G(v) = |N'_G(v)|$. When there is no ambiguity, we may omit the subscripts from N and d.

For $W \subseteq V(G)$, denote by $G[W]$ the subgraph of G induced by W, and write $E_G(W)$ to represent $E(G[W])$. As before, we omit the subscript when there is no ambiguity about G. A *clique* is a set of pairwise adjacent vertices. We also use the term *clique* to refer to the corresponding induced subgraph. A clique of size k is represented by K_k, and the graph K_3 is called a *triangle*. We shall denote by $O(n^\omega)$ the time required for the multiplication of two $n \times n$ matrices. Up to this date, the best bounds on n^ω are $n^2 \leq n^\omega < n^{2.376}$ [6]. The *arboricity* $\alpha(G)$ of G is the minimum number of edge-disjoint spanning forests into which G can be decomposed. The *h-index* $h(G)$ of G is the maximum h such that G contains h vertices of degree at least h.

The following lemma relates these parameters.

Lemma 1. *For every non-trivial graph G, with minimum degree δ,*

$$\frac{\delta}{2} < \frac{m}{n-1} \leq \alpha(G) \leq h(G) \leq \sqrt{2m}$$

PROOF. The inequalities $\frac{\delta}{2} < \frac{m}{n-1} \leq \alpha(G)$ and $h(G) \leq \sqrt{2m}$ are straightforward. Let d be the degeneracy of G, i.e.,

$$d = \max \min_{w \in V(H)} \{d(w) \mid H \text{ is an induced subgraph of } G.\}$$

It is well known that $\alpha(G) \leq d$ (see e.g. [3]). We show that $d \leq h(G)$. Consider an induced subgraph H of G with minimum degree δ_H. Clearly, if H contains a vertex that has degree at most $h(G)$ in G, then $\delta_H \leq h(G)$. On the other hand, if all the vertices in H have degree at least $h(G)$ in G, then H has at most $h(G)$ vertices, by the definition of the h-index. Therefore, $\delta_H < |V(H)| \leq h(G)$, and so $d \leq h(G)$. □

For each vertex v of a graph G, define $N(v,i) = \{w \in N(v) \mid d(w) = i\}$, i.e., $N(v,i)$ is the set of neighbors of v with degree i. Denote by $L(v)$ the set of neighbors of v of degree at most $d(v)-1$, and $H(v)$ the set of neighbors of v of degree at least $d(v)$, i.e., $L(v) = N(v,1) \cup \ldots \cup N(v,d(v)-1)$, and $H(v) = N(v,d(v)), \ldots, N(v,n-1)$. We use $\ell(v)$ and $h(v)$ to respectively denote $|L(v)|$ and $|H(v)|$. Write

$$\mathcal{N}(v) = \{N(v,i) \mid N(v,i) \neq \emptyset \text{ and } 1 \leq i < d(v)\}.$$

Observe that v can have at most $h(G)$ neighbors of degree at least $d(v)+1$. Consequently, $h(v) \leq h(G)$, meaning that the number of nonempty sets that belong to $\mathcal{N}(v)$ is at most $2h(G)$.

3. Basic Propositions

In this section, we present the main propositions, which are basic for the complexity analysis of the algorithms, throughout the paper.

The first lemma was employed by Chiba and Nishizeki, in their algorithm for listing the triangles of a graph.

Lemma 2 ([4]). *For every graph G,*

$$\sum_{vw \in E(G)} \min\{d(v), d(w)\} \leq 2\alpha(G)m.$$

The next two lemmas and corollaries are employed in the algorithms we propose.

Lemma 3. *Let $e_1, \ldots, e_m$ be an ordering of $E(G)$ for a graph G, and set $e_i = v_i w_i$. Denote by $h_i(v)$ the value of $h(v)$ in the spanning subgraph of G that contains the edges $e_1, \ldots, e_i$, for every $1 \leq i \leq m$. Then,*

$$\sum_{i=1}^{m} (h_i(v_i) + h_i(w_i)) \leq 4\alpha(G)m.$$

Proof. Denote by G_i the spanning subgraph of G that contains only the edges $e_1, \ldots, e_i$, and let $d_i(v) = d_{G_i}(v)$ and $H_i(v) = H_{G_i}(v)$, for every $v \in V(G)$. For every $vw \in E(G)$, define the values $f_i(v, w)$, $f_i(w, v)$, $F(v, w)$, and $F(w, v)$ such that

$$f_i(x, y) = \begin{cases} 1 & \text{if (i) } e_i \text{ is incident to } x \text{ and (ii) } y \in H_i(x) \\ 0 & \text{otherwise,} \end{cases}$$

and $F(x, y) = \sum_{i=1}^{m} f_i(x, y)$. By (i), there are at most $d_G(v)$ edges e_i such that $f_i(v, w) = 1$, thus $F(v, w) \leq d_G(v)$. On the other hand, by (ii), $f_i(v, w) = 1$ only if $d_G(w) \geq d_i(w) \geq d_i(v) = |\{j \leq i \mid e_j \text{ is incident to } v\}|$. It is clear that if e_i and e_j are both incident to v and $j < i$, then $d_j(v) < d_i(v)$. Thus, at most $d_G(w)$ edges e_i hold condition (ii), which implies that $F(v, w) \leq \min\{d_G(v), d_G(w)\}$. Also, observe that $f_i(v, w) + f_i(w, v) \geq 1$ if only if one of the vertices v, w, say x, incides in e_i while the other belongs to $H_i(x)$. In other words, $H_i(v_i) = \{w \in N_G(v) \mid f(v_i, w) + f(w, v_i) \geq 1\}$, $H_i(w_i) = \{v \in N_G(w_i) \mid f(v, w_i) + f(w_i, v) \geq 1\}$, and $f_i(v, w) = 0$ when $\{v, w\} \cap \{v_i, w_i\} = \emptyset$. Finally, observe that $f_i(v_i, w_i) + f_i(w_i, v_i) = 2$ when $v_i \in H_i(w_i)$ and $w_i \in H_i(v_i)$. Therefore, by Lemma 2,

$$\begin{aligned} \sum_{i=1}^{m} (h_i(v_i) + h_i(w_i)) &\leq \sum_{i=1}^{m} \sum_{vw \in E(G)} (f_i(v, w) + f_i(w, v)) \\ &\leq \sum_{vw \in E(G)} (F_i(v, w) + F_i(w, v)) \\ &\leq \sum_{vw \in E(G)} 2 \min\{d_G(v), d_G(w)\} \leq 4\alpha(G)m \end{aligned}$$

□

Lemma 4. *For every graph G,* $\sum_{vw \in E(G)} (h(v) + h(w)) \leq 4\alpha(G)m$.

PROOF. The proof is similar to the one of Lemma 3. Just replace H_i with H in the definition of f_i and follow the same proof. □

The next corollaries are also relevant for the h-graph data structure to be presented.

Corollary 5. *Let $v_1, \ldots, v_n$ be an ordering of $V(G)$, for a graph G. Denote by $h_i(v)$ the value of $h(v)$ in the subgraph of G induced by $v_1, \ldots, v_i$, for every $1 \leq i \leq n$. Then,*

$$\sum_{i=1}^{n} \sum_{w \in N(v_i)} h_i(w) \leq 4\alpha(G)m.$$

Corollary 6. *For every graph G,* $\sum_{v \in V(G)} \sum_{w \in N(v)} h(w) \leq 4\alpha(G)m$.

4. The *h*-Graph Data Structure

In this section, we present a new data structure, called the h-graph data structure, which has been designed having in mind dynamic algorithms. It is well suited for graphs with low arboricity. We apply it in the next sections to solve some different graph algorithmic problems. The purpose of the data structure is to efficiently handle some typical operations of dynamic algorithms, as vertex and edge insertions, vertex and edge removals, adjacency queries, finding some desired subsets of vertices or edges, and constructing subgraphs induced by vertex neighborhoods.

For a given graph G, the h-graph data structure consists of one triple for each vertex v, defined as follows

$$(d(v), \mathcal{N}(v), H(v)).$$

Recall that $d(v)$ is the degree of v, $\mathcal{N}(v)$ is the family of subsets $N(v, i) \neq \emptyset$, $1 \leq i < d(v)$, and $H(v)$ is set of neighbors of v having degree at least $d(v)$.

4.1. Operations supported by the h-graph data structure

Table 3 lists the operations that the h-graph data structure supports, for a graph G having n vertices, m edges, arboricity α and h-index h. To simplify notation, we write d with the meaning of $d(v)$. The third column shows the complexity of performing the corresponding operation once, while the last column illustrates the complexity when the operation is applied for all the vertices or edges. For instance, the insertion of one vertex takes $O(dh)$ time, while inserting all the vertices of the graph requires $O(\alpha m)$ time.

Operation	**Description**	**Complexity**	
		One	**All**
Vertex insertion	Inserts new vertex v with given $N(v)$	$O(dh)$	$O(\alpha m)$
Vertex removal	Removes vertex v	$O(dh)$	$O(\alpha m)$
Edge insertion	Inserts new edge vw	$O(h)$	$O(\alpha m)$
Edge removal	Removes edge vw	$O(h)$	$O(\alpha m)$
Adjacency query	Queries if two vertices v, w are adjacent	$O(h)$	-
Finding $H(v)$	Returns set $H(v)$	$O(1)$	-
Finding $N'(v)$	Returns set $N'(v)$	$O(dh)$	$O(\alpha m)$
Constructing $G[N(v)]$	Constructs the graph $G[N(v)]$	$O(dh)$	$O(\alpha m)$

Table 3: Operations supported by the h-graph data structure

4.2. Brief description of the operations supported by the h-graph structure

Next, we give a brief outline of how the operations of Table 3 are performed. Some implementation details, together with the complexity analysis, appears in Section 5.

Insertion of edges. The algorithm for inserting a new edge vw into G has two phases. In the first one, we update the families $\mathcal{N}(z)$, for every

$z \in N[v] \cup N[w]$. On the other hand, the second phase actually inserts the new edge into the sets $N(w, d_G(v)+1)$ and $N(v, d_G(w)+1)$, while updating the values of $d(v)$ and $d(w)$.

Insertion of vertices. For inserting a new vertex v, with given neighborhood $N(v)$, first, insert v as an isolated vertex, and then add the edges vw, for each $w \in N(v)$.

Removal of edges. For removing an edge vw, we undo the insertion process. But, this time, we start by the computations corresponding to the second phase. That is, first we physically remove the edge vw. Then we proceed to undoing the first phase. In this case, we need to update the families $\mathcal{N}_z$, for every $z \in N_G(v) \cup N_G(w)$.

Removal of vertices. Similarly as before, remove vertex v by removing all its incident edges first. At the end, remove v after becoming an isolated vertex.

Adjacency query. Querying whether two vertices v and w are adjacent is straightforward. Simply, traverse the set $H(z)$, where z is the vertex of least degree, between v and w.

Finding $H(v)$. Trivial, since in the data structure $H(v)$ belongs to the triple of v

Finding $N'(v)$. To construct $N'(v)$, we ought to traverse $N(v)$ and then $H(w)$, for each $w \in N(v)$.

Constructing $G[N(v)]$. Clearly, the subgraph of G induced by $N(v)$ is just the graph whose vertex set is $N(v)$, which is obtained from $\mathcal{N}(v)$ and $H(v)$, and whose edge set is $N'(v)$, which can be determined as above.

5. Complexity of the *h*-graph Basic Operations

In this section, we describe some implementation details employed in the algorithm for performing the basic operations illustrated in Table 3. The complexity analysis is done over the more detailed algorithms.

Recall that the h-graph data structure consists of a triple $(d(v), \mathcal{N}(v), H(v))$, for each vertex v of G. In the implementation, we represent $d(v)$ and $H(v)$ by the objects $\mathtt{d}(v)$ and $\mathtt{H}(v)$, respectively, while $N(v,i)$ is represented by the

object $\mathtt{N}(v,i)$. The family $\mathcal{N}(v)$ is stored as a double linked list, containing one object $\mathtt{N}(v,i)$, for each non-empty set $N(v,i)$, $1 \leq i < d(v)$. This list is ordered, so that $\mathtt{N}(v,i)$ appears in increasing values of i. On the other hand, each $\mathtt{N}(v,i)$, for any i, is also stored as a doubly linked list that contains one object for each $w \in N(v,i)$.

Finally, we mention that throughout the process, we maintain direct pointers to objects representing vertices, edges and lists. To make this statement more precise, suppose that v belongs to a list $\mathtt{L}$ of $\mathcal{N}(w) \cup \mathtt{H}(w)$, for some $w \in N(v)$. Then, the appearance of w in $N(v)$ is associated with a pointer $\mathtt{p}$ referencing the appearance of v inside L, a pointer $\mathtt{l}$ referencing $\mathtt{L}$, and a pointer $\mathtt{n}(v,w)$ referencing v (see Figure 2).

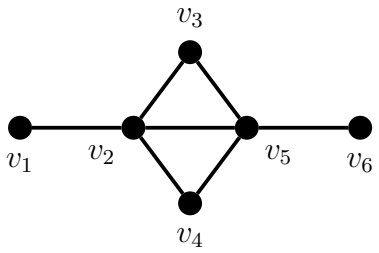

Figure 1: Example graph used to describe the h-graph data structure.

The implementation of the data structure is now complete. Next, we proceed to the implementation and complexity of the basic operations.

5.1. Insertion of vertices and edges

First, consider the insertion of an edge vw. Recall that we divide it into two phases. In the first phase, create the set $\mathtt{N}(v, d_G(v))$, move the vertices with degree $d_G(v)$ from $\mathtt{H}(v)$ to $\mathtt{N}(v, d_G(v))$, and move v from $\mathtt{N}(z, d_G(v))$ to $\mathtt{N}(z, d_G(v)+1)$, for every $z \in H(v)$. Next, apply the analogous operations for w. In the second phase, insert v at the end of $\mathtt{N}(w, d_G(v)+1)$ and w at the end of $\mathtt{N}(v, d_G(w)+1)$, update the values of $\mathtt{d}(v)$ and $\mathtt{d}(w)$.

Discuss the time complexity of the above algorithm. For the first phase, apply Algorithm 1 twice, once for v and once for w. Recall that this algorithm is applied before incrementing $\mathtt{d}$ for v and w, thus $\mathtt{d}(z)$ is the degree of z before the insertion of vw. Note that each iteration of the main loop can be implemented so as to run in $O(1)$ time, by using the associated pointers.

$$v_1\boxed{1} = \begin{cases} \mathtt{d} = 1 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{7} = [v_2\boxed{8} : \mathtt{p} = 10, \mathtt{l} = 9, \mathtt{n} = 2] \end{cases}$$

$$v_2\boxed{2} = \begin{cases} \mathtt{d} = 4 \\ \mathcal{N} = \begin{cases} \mathtt{N}(1)\boxed{9} = [v_1\boxed{10} : \mathtt{p} = 8, \mathtt{l} = 7, \mathtt{n} = 1], \\ \mathtt{N}(2)\boxed{11} = [v_3\boxed{12} : \mathtt{p} = 17, \mathtt{l} = 16, \mathtt{n} = 3, v_4\boxed{13} : \mathtt{p} = 20, \mathtt{l} = 19, \mathtt{n} = 4] \end{cases} \\ \mathtt{H}\boxed{14} = [v_5\boxed{15} : \mathtt{p} = 28, \mathtt{l} = 27, \mathtt{n} = 5] \end{cases}$$

$$v_3\boxed{3} = \begin{cases} \mathtt{d} = 2 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{16} = [v_2\boxed{17} : \mathtt{p} = 12, \mathtt{l} = 11, \mathtt{n} = 2, v_5\boxed{18} : \mathtt{p} = 25, \mathtt{l} = 24, \mathtt{n} = 5] \end{cases}$$

$$v_4\boxed{4} = \begin{cases} \mathtt{d} = 2 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{19} = [v_2\boxed{20} : \mathtt{p} = 13, \mathtt{l} = 11, \mathtt{n} = 2, v_5\boxed{21} : \mathtt{p} = 26, \mathtt{l} = 24, \mathtt{n} = 5] \end{cases}$$

$$v_5\boxed{5} = \begin{cases} \mathtt{d} = 4 \\ \mathcal{N} = \begin{cases} \mathtt{N}(1)\boxed{22} = [v_6\boxed{23} : \mathtt{p} = 30, \mathtt{l} = 29, \mathtt{n} = 6] \\ \mathtt{N}(2)\boxed{24} = [v_3\boxed{25} : \mathtt{p} = 18, \mathtt{l} = 16, \mathtt{n} = 3, v_4\boxed{26} : \mathtt{p} = 21, \mathtt{l} = 19, \mathtt{n} = 4] \end{cases} \\ \mathtt{H}\boxed{27} = [v_2\boxed{28} : \mathtt{p} = 15, \mathtt{l} = 14, \mathtt{n} = 2] \end{cases}$$

$$v_6\boxed{6} = \begin{cases} \mathtt{d} = 1 \\ \mathcal{N} = \emptyset \\ \mathtt{H}\boxed{29} = [v_5\boxed{30} : \mathtt{p} = 23, \mathtt{l} = 22, \mathtt{n} = 5] \end{cases}$$

Figure 2: Snapshot of the variables used by the h-graph data structure for the graph of Figure 1. An object representing each vertex v_i is shown to the left of an equal sign, for $i = 1, \ldots, 6$. The positions of these objects in the main memory are shown inside a box, e.g., the object representing v_1 occupies the cell number 1 of the memory. The data maintained for each vertex is shown to the right of the equal sign. Again, the position of each object in the main memory is shown in a box, but only for those objects that can be referenced by a pointer. For instance, variable H for v_1 occupies the cell number 7 of the main memory. Symbols p, l, and n represent the various pointers that are kept through the process. Observe that the total space required is $O(n + m)$ bits.

Thus, the update of $\mathcal{N}(z)$, for every $z \in N[v] \cup N[w]$, takes $O(h(v) + h(w))$ time. For the second phase, traverse the family $\mathcal{N}(w)$ until the first set $\mathtt{N}(w, d)$ with $d > d_G(v)$ is reached. Then, create the set $\mathtt{N} = \mathtt{N}(w, d_G(v) + 1)$, if $d > d(v) + 1$, and insert v into N. Next, traverse $\mathcal{N}(v)$ so as to find the set that ought to contain w, and insert w. Recall that there are at most $2h(G)$ sets inside each of the families $\mathcal{N}(v)$ and $\mathcal{N}(w)$. So, the time required by these steps is $O(\min\{d(v), d(w), h(G)\})$, while the creation of the pointers

Algorithm 1 Update of $\mathcal{N}(z)$ for every $z \in N[v]$.

Input: the h-graph data structure representing G, and $v \in V(G)$
Output: none. The h-graph data structure is modified.

1. Insert a new empty set $\mathtt{N}(v, d(v))$ at the end of $\mathcal{N}(v)$.
2. For each $z \in \mathtt{H}(v)$:
3. If $\mathtt{d}(z) = \mathtt{d}(v)$, then move z from $\mathtt{H}(v)$ to $\mathtt{N}(v, d(v))$.
4. Move v from $\mathtt{N}(z, d(v))$ to $\mathtt{N}(z, d(v) + 1)$. If $\mathtt{N}(z, d(v)) = \emptyset$, then delete $\mathtt{N}(z, d(v))$.
5. If $\mathtt{N}(v, d(v)) = \emptyset$, then delete $\mathtt{N}(v, d(v))$.

and the increase of $\mathtt{d}$ take $O(1)$ time. Therefore, the insertion of vw requires time

$$O\left(\min\{d(v), d(w), h(G)\} + h(v) + h(w)\right) = O(h(G))$$

As for the insertion of a new vertex v, it follows directly that we require $O(1 + d(v)h(G))$ time. If we use the above algorithm for building G from scratch, then the total time is $O(n + \alpha(G)m)$, by Lemma 3.

Corollary 7. *The time required for inserting the vertices and edges of a graph G, one at a time in no particular order, into an initially empty h-graph data structure is $O(n + \alpha(G)m)$.*

Proof. The insertion of the n vertices takes $O(n)$ time. Let $e_1, \ldots, e_m$ be an ordering of $E(G)$, denote by $d_i(v)$ and $h_i(v)$ the values of $d(v)$ and $h(v)$ in the graph prior the insertion of e_i. By the analysis above, the time required for the insertion of $e_i = vw$ is

$$O(\min\{d_i(v), d_i(w), h(G)\} + h_i(v) + h_i(w)).$$

Thus, by Lemmas 2 and 3, the total time required for the insertion of all the edges is

$$\sum_{vw \in E(G)} O(\min\{d_i(v), d_i(w), h(G)\}) + \sum_{vw \in E(G)} O(h_i(v) + h_i(w)) = O(\alpha(G)m)$$

□

5.2. Removal of vertices and edges

For removing an edge vw, assume $d_G(v) \leq d_G(w)$, i.e. $w \in H(v)$. Recall that we undo the process of insertion, starting by the second phase of the insertion. That is, we ought to physically remove edge vw. With this purpose, traverse $\mathtt{H}(v)$ so as to locate and remove the object that represents w in $\mathtt{H}(v)$. Using the associated pointers, we can easily remove v from $\mathtt{N}(w, d_G(v))$ in $O(1)$ time. Since v has $h(v)$ neighbors in $H(v)$, this phase takes $O(h(v)) = O(h(G))$ time. To undo the first phase, we need to update the families $\mathcal{N}(z)$ for every $z \in N_G(v) \cup N_G(w)$. For this, we move v from $N(z, d_G(v))$ to $N(z, d_G(v) - 1)$ for every $z \in H_G(v)$, and then we remove $\mathtt{N}(v, d_G(v) - 1)$ from $\mathcal{N}(v)$ so as to append it to $\mathtt{H}(v)$. Next, we should apply the same operations for w. This phase is rather similar to the one for edge insertion and it also takes time

$$O(h_{G\setminus\{vw\}}(v) + h_{G\setminus\{vw\}}(w)) = O(h(G)).$$

It then follows that the removal of a vertex requires $O(d(v)h(G))$. Finally, if we use the above algorithm for decomposing G, then the time required for removing all the edges is $O(n + \alpha(G)m)$, by Lemma 3.

Corollary 8. *The time required for removing the vertices and edges of a graph G, one at a time and in no particular order, from an h-graph data structure is $O(n + \alpha(G)m)$.*

5.3. Adjacency query

To check whether vertices v, w are adjacent, we traverse set $H(z)$, where z is the vertex of least degree between v and w. Clearly, this requires $O(h(v) + h(w)) = O(h(G))$ time.

5.4. Finding $N'(v)$

To compute the set $N'(v)$ for a vertex v, first mark each $z \in N(v)$ with 1. Following, traverse each $z \in H(w)$ for every $w \in N(v)$ and, if z is marked with 1, then insert it into $N'(v)$ and mark it with 2. Since each w is traversed in $O(h(w)) = O(h(G))$ time, the algorithm takes $O(d(v)h(G))$ time. Furthermore, the time required to find $\{N'(v)\}_{v \in V(G)}$, by applying this algorithm to all the vertices in the graph, is $O(\alpha(G)m)$, by Corollary 6.

5.5. Constructing $G[N(v)]$

Recall that the subgraph of G induced by $N(v)$ is just the graph whose vertex set is $N(v)$ and whose edge set is $N'(v)$. Thus, the graph $G[N(v)]$, implemented with adjacency lists, can be computed in $O(d(v)h(G))$ time, while the family $\{G[N(v)]\}_{v\in V(G)}$ can be computed in $O(\alpha(G)m)$ time.

In the next few sections we list several problems that can be improved by using the technique implicit in Lemma 3 and Corollary 5. These algorithms build upon the h-graph data structure, and serve as examples of its applicability. One of the most appealing aspects of some of these algorithms is that they are simple to obtain from the definitions of the problems.

6. Listing the Cliques of Size k of a Graph

In this section, we show a simple modification of Chiba and Nishizeki's algorithms for listing the cliques of size k of a graph, that transforms it into a dynamic algorithm, while preserving its complexity.

First, consider the problem of finding all the cliques of size $k > 2$ that contain a given vertex v. As in [4], this problem is solved by computing all the K_{k-1}'s in $G' = G[N(v)]$ and listing v plus these cliques. By [4] all cliques of size $k-1$ of G' are obtained in time

$$O(|V(G')| + (k-1)\alpha(G')^{k-3}|E(G')|).$$

Note that

$$\begin{aligned} |E(G')| &\leq \sum_{w\in N(v)} \min\{d(v), d(w)\} \leq \\ &\leq \sum_{\substack{w\in N(v)\\ d(w)\leq h(G)}} d(w) + \sum_{\substack{w\in N(v)\\ d(w)>h(G)}} d(v) = O(d(v)h(G)), \end{aligned}$$

thus the time required to find all the K_k's that contain v is

$$O\left(d(v)h(G) + |V(G')| + k\alpha(G')^{k-3}|E(G')|\right) = O\left(kd(v)h(G)\alpha(G)^{k-3}\right).$$

The total time required for listing all the cliques, by iteratively executing the above algorithm for each vertex v (and then removing v), is

$$O\left(n + \alpha(G)m + k\alpha(G)^{k-3}\textstyle\sum_{v\in V(G)}\sum_{w\in N(v)}\min\{d(v), d(w)\}\right) = O\left(n + k\alpha(G)^{k-2}m\right),$$

matching the time complexity of the algorithm by Chiba and Nishizeki.

7. The 4-Subgraph Counting Problem

The 4-subgraph counting problem is the problem of counting how many copies of some graph H on four vertices appear as induced subgraphs of a given graph G. The connected graphs on four vertices are six: the complete graph K_4, the diamond $K_4 \setminus \{e\}$ for $e \in E(K_4)$, the square C_4, the path P_4, the paw $\overline{P_3 \cup K_1}$, and the claw $\overline{K_3 \cup K_1}$. The disconnected graphs on four vertices are five: $\overline{K_4}$, $\overline{K_4 \setminus \{e\}}$, $\overline{C_4}$, $P_3 \cup K_1$, and $K_3 \cup K_1$. In [11], Kloks et al. showed a system of linear equations that solves the 4-subgraph counting problem for connected graphs. Specifically, Kloks et al. proved the following theorem.

Theorem 9 ([11]). *Let $\tilde{H}$ be a connected graph on four vertices such that there is an $O(t(G))$ time algorithm counting the number of induced $\tilde{H}$'s in a graph G. Then, there is an $O(n^\omega + t(G))$ time algorithm counting the number of induced H's of G for all connected graphs H on four vertices.*

Since the number of K_4's can be computed in either $O(n + m^{(\omega+1)/2}) = O(n + m^{1.61})$ or $O(n + \alpha(G)^2 m)$ time [4, 11], solving the 4-subgraph counting problem for connected graphs takes $O(n^\omega + \min\{m^{1.61}, \alpha(G)^2 m\})$ time.

In this section, we improve the above results in two ways. First, we formulate a system of equations, which also includes all disconnected graphs with 4 vertices. Furthermore, by employing the h-graph data structure, we show that the time complexity can be decreased to $O(n + \min\{m^{1.61}, \alpha(G)^2 m\})$.

The new system of linear equations appears in the proof of the next theorem.

Theorem 10. *Let $\tilde{H}$ be a graph on four vertices such that there is an $O(t(G))$ time algorithm counting the number of induced $\tilde{H}$'s in a graph G. Then, there is an $O(n + \alpha(G)m + t(G))$ time algorithm counting the number of induced H's of G for every graph H on four vertices.*

Proof. Let k, d, s, p, q, and y denote the number of induced K_4's, diamonds, squares, P_4's, paws, and claws in G, respectively. Similarly, let $\bar{k}$, $\bar{d}$, $\bar{s}$, $\bar{q}$ and $\bar{y}$ be the number of induced complements of K_4's, diamonds, squares, paws, and claws in G, respectively. Define $\overline{m} = \binom{n}{2} - m$, $\overline{d(v)} = n - d(v) - 1$ for $v \in V(G)$, and $\delta(v, w) = d(v) - d(vw)$ for $vw \in E(G)$. That is, $\overline{m}$ is the number of edges of $\overline{G}$, $\overline{d(v)}$ is the degree of v in $\overline{G}$, and $\delta(v, w)$ is the number of vertices that are adjacent to v and not w. Finally, let $\mathcal{S}$ be the

set obtained after executing the algorithm $C4$ in [4]. In $\mathcal{S}$, each element is a triple (v, w, L) where $v, w \in V(G)$ and L is a set of vertices. Then, G fulfills the system of linear equations shown in Figure 3.

$$\sum_{(v,w,L)\in\mathcal{S}} \binom{|L|}{2} = 3k + d + s \tag{1}$$

$$\sum_{vw\in E(G)} \binom{d(vw)}{2} = 6k + d \tag{2}$$

$$\sum_{vw\in E(G)} \delta(v, w)\delta(w, v) = 4s + p \tag{3}$$

$$\sum_{vw\in E(G)} \left(\binom{\delta(v, w)}{2} + \binom{\delta(w, v)}{2} \right) = q + 3y \tag{4}$$

$$\sum_{vw\in E(G)} \binom{d(v) + d(w) - d(vw) - 2}{2} = 6k + 5d + 4s + p + 3q + 3y \tag{5}$$

$$\sum_{v\in V(G)} d'(v)(n - 3) = 12k + 6d + 3q + 3\bar{y} \tag{6}$$

$$\sum_{v\in V(G)} \binom{d(v)}{2}(n - 3) = 12k + 8d + 4s + 2p + 5q + 3y + \bar{q} + 3\bar{y} \tag{7}$$

$$\binom{m}{2} - \sum_{v\in V(G)} \binom{d(v)}{2} = 3k + 2d + 2s + p + q + \bar{s} \tag{8}$$

$$\binom{\overline{m}}{2} - \sum_{v\in V(G)} \binom{\overline{d(v)}}{2} = s + p + 3\bar{k} + 2\bar{d} + 2\bar{s} + \bar{q} \tag{9}$$

$$\binom{n}{4} = k + d + s + p + q + y + \bar{k} + \bar{d} + \bar{s} + \bar{q} + \bar{y} \tag{10}$$

Figure 3: Equations for Theorem 10

Equations (2)–(4) correspond to the first, third and fifth equations of [11], respectively. For the translation between them, observe that if A is the adjacency matrix of G and C is the adjacency matrix of the complement of G, then $A^2_{v,w} = d(vw)$ and $AC_{v,w} = \delta(v, w)$. Thus, G satisfies these

equations [11].

Each triple of $\mathcal{S}$ represents a set of non induced cycles of G in the following way. Let $v_1, \ldots, v_n$ be an ordering of $V(G)$ in non increasing order of degree. Then (v_i, v_j, L) is a triple of $\mathcal{S}$ if and only if $i < \min\{j, k, l\}$, $L \subset N(v_i) \cap N(v_j)$ and $|L| \geq 2$. Therefore, $\binom{|L|}{2}$ counts the number of non induced 4-length cycles that contain v_i and v_j and such that $v_i v_j$ is not an edge of the cycle. Such cycles count three times each K_4, and once each diamond and each square. Thus, (1) is fulfilled by G.

The remaining equations follow analogously, by observing that: (5) counts, for each edge vw, the number of pairs of vertices that are adjacent to at least one of v and w; (6) counts, for each vertex v, the number of triangles of v plus one vertex; (7) counts those graphs on four vertices where v has degree at least 2; (8) and (9) count the number of pair of disjoint edges on G and $\overline{G}$, respectively; and (10) counts the number of induced graphs on 4 vertices of G.

There are 10 equations and 11 variables to be determined

$$k, d, s, p, q, y, \overline{k}, \overline{d}, \overline{s}, \overline{q}, \overline{y}.$$

If we add an 11th equation, fixing a given value for k, then the system of equation becomes triangular. In this case the values of the variables can be easily determined, as each one becomes uniquely determined by the values of the variables that precede it in the above sequence of variables. This implies that these 11 equations are linearly independent, and so are equations (1)-(10). Consequently, by fixing the number of subgraphs $\tilde{H}$, we can solve the system.

As for the time complexity of computing the constant terms of the system (i.e. appearing at the left sides of the equations), those corresponding to equations (1)-(7) can all be found in $O(n + \alpha(G)m)$ time, while those of equations (7)–(10) are computed in $O(n + m)$ time. □

Corollary 11. *The number of diamonds, paws and claws of a graph can be computed in* $O(n + min\{m^{1.61}, \alpha(G)^2 m\})$.

We now consider a slight modification of the 4-subgraph counting problem. Given a vertex v, the goal is to count the number of graphs on 4 vertices that contain v. We focus our attention on four types of connected graphs: K_4's, diamonds, paws, and claws. For $i \in \{1, 2, 3\}$, define $k_i(v)$, $d_i(v)$, $q_i(v)$, and $y_i(v)$ as the number of K_4's, diamonds, paws, and claws that contain a

given vertex v, where the degree of v in such an induced subgraph is i. In the previous section we saw that $k_3(v)$ can be computed in $O(d(v)h(G)\alpha(G))$ time. The following theorem shows that we can compute $d_i(v)$, $q_i(v)$, and $y_i(v)$, once $k_3(v)$ is given.

Theorem 12. *There is a dynamic graph data structure, requiring $O(n+m)$ space, with the following properties:*

- *Both the insertion and the removal of a vertex v take $O(d(v)h(G))$ time.*
- *The time required for inserting the vertices of G, one at a time, into an initially empty instance of the data structure is $O(n + \alpha(G)m)$.*
- *Given $k_3(v)$, the values of $d_i(v)$, $q_i(v)$, and $y_i(v)$ can be determined in time $O(d(v)h(G))$, for every $v \in V(G)$ and $i \in \{1, 2, 3\}$.*
- *If $k_3(v)$ is given for every $v \in V(G)$, then the number of diamonds, paws, and claws can be found in $O(n + \alpha(G)m)$ time.*

Proof. Fix a vertex v and let $k_i = k_i(v)$, $d_i = d_i(v)$, $q_i = q_i(v)$, and $y_i = y_i(v)$, for $i \in \{1, 2, 3\}$. Define δ as in Theorem 10. Then, v fulfills the system of linear equations shown in Figure 4.

These equations are similar to those in Theorem 10, e.g., $\binom{d(vw)}{2}$ counts the number of pairs of vertices x, z that are both adjacent to v and w, thus $\sum_{w \in N(v)} \binom{d(vw)}{2}$ counts three times each K_4 that contains v, and one time each diamond that contains v with degree 3. The other equations follow analogously.

The dynamic data structure is just the h-graph data structure with the addition that $d(wz)$ is stored in a pointer, for every edge wz. This value can be easily updated in $O(d(v)h(G))$ time when a vertex v is either inserted or removed. Indeed, according to whether v is inserted or removed, we can increase or decrease in 1 the value of $d(wz)$ for every $wz \in N'(v)$ with a single traversal of $N'(v)$. On the other hand, the value of $d(vw)$ is simply the degree of w in $G[N(v)]$, for every $w \in N(v)$. Then, the values of $d_i(v)$, $q_i(v)$, and $y_i(v)$ can be obtained in $O(d(v)h(G))$ time by solving the system of equations above, once $k_3(v)$ is given. □

8. Dynamic Recognition of Diamond-Free Graphs

Recall that the graph obtained by removing one edge from a complete graph of four vertices is called a *diamond*. A *diamond-free* graph is a graph

$$
\begin{aligned}
d_3 &= \sum_{w \in N(v)} \binom{d(vw)}{2} - 3k_3 \\
d_2 &= \sum_{wz \in N'(v)} (d(wz) - 1) - 3k_3 \\
2q_3 &= \sum_{w \in N(v)} d(vw)\delta(v,w) - 2d_3 \\
q_2 &= \sum_{w \in N(v)} d(vw)\delta(w,v) - 2d_2 \\
q_1 &= \sum_{w \in N(v)} (d'(w) - d(vw)) - 3k_3 - 2d_2 \\
3y_3 &= \sum_{w \in N(v)} \binom{\delta(v,w)}{2} - q_3 \\
y_1 &= \sum_{w \in N(v)} \binom{\delta(w,v)}{2} - q_1
\end{aligned}
$$

Figure 4: Equations for Theorem 12

that contains no induced diamond. Diamond-free graphs appear in many contexts; for example in the study of perfect graphs [5, 10, 22].

In [11], Kloks et al. showed how to find an induced diamond in $O(m^{3/2} + n^{\omega})$ time, if one exists. The fast matrix multiplication algorithm is used in one of the steps of this algorithm, which explains why n^{ω} is a term of the complexity order. However, the fast matrix multiplication can be avoided improving the time complexity to $O(m^{3/2})$ time, as shown by Eisenbrand and Grandoni [7]. Talmaciu and Nechita [21] devised a recognition algorithm based on decompositions, but they claim that in the worst case the time required by their algorithm is not better than the one by Kloks et al. Note that Theorem 10 implies that there is an $O(\alpha(G)^2 m)$ time algorithm for recognizing whether a graph is a diamond-free graph, improving over previous algorithms for some sparse graphs. Finally, Vassilevska [23] used the algorithm by Eisenbrand and Grandoni to find an induced $K_k \setminus e$ in a graph. A $K_k \setminus e$ is a complete graph on k vertices, minus one edge. For every even k, the algorithm by Vassilevska takes $O\left(d(n,m)m^{(k-4)/2}\right)$ time, where $d(n,m)$

is the time required to find a diamond in a graph with n vertices and m edges. Thus, this algorithm is implicitly improved with each improvement on $d(n, m)$.

The algorithm by Kloks et al. is based on the fact that a graph G is diamond-free if and only if $G[N(v)]$ is a disjoint union of maximal cliques, for every $v \in V(G)$. Testing whether a graph is a disjoint union of cliques takes linear time, and we saw in Section 5 how to compute the family $\{G[N(v)]\}_{v \in V(G)}$ in $O(\alpha(G)m)$ time. Therefore, by using the h-graph data structure, the algorithm by Kloks et al. can be implemented so as to run in $O(\alpha(G)m)$ time, improving over the algorithm by Eisenbrand and Grandoni and the algorithm implied by Theorem 10. (Further discussion about this approach is given in Section 11.)

In this section we employ the h-graph data structure for dynamically maintaining diamond-free graphs. The data structure can also be used to find an induced diamond of a static graph G in $O(\alpha(G)m)$ time, if one exists. As a by-product, the data structure can be used to query the maximal cliques of the dynamic diamond-free graph in constant time. The data structure is based on this well known theorem about diamond-free graphs (see for instance [12]).

Theorem 13. *A graph is a diamond-free graph if and only if every edge belongs to exactly one maximal clique.*

We assume a dynamic context, in which vertices and edges can be inserted or removed from the graph.

8.1. Insertion of vertices

Start by examining the insertion of vertices. Suppose that G is a diamond-free graph and $v \notin V(G)$ is to be inserted into G. We want to know whether $G \cup \{v\}$ is still diamond-free. The following definitions are useful.

Say that v is *edge-adjacent* to a clique C of G when $E(C) \cap N'(v) \neq \emptyset$, while v is *fully edge-adjacent* to C when $E(C) \cap N'(v) = E(C)$. Applying these concepts, the effect of inserting v into G is shown by the next theorem.

Theorem 14. *The graph $G \cup \{v\}$ is diamond-free if and only if the following two statements hold for every maximal clique C to which v is edge-adjacent.*

1. *v is fully edge-adjacent to C, and*

2. *if v is edge-adjacent to a maximal clique $C' \neq C$, then $V(C') \cap V(C) = \emptyset$.*

PROOF. If v is not fully edge-adjacent to C, then there is some vertex $u \in C$ which is not adjacent to v. Since v is edge-adjacent to C, then there is an edge $wz \in N'(v) \cap E(C)$. But then, u, v, w, z induce a diamond in G. Suppose now that v is edge-adjacent to a maximal clique $C' \neq C$ that contains some vertex $u \in C$. Since C and C' are maximal cliques, it follows that there is some vertex $w \in C$ which is not adjacent to a vertex $z \in C'$. Thus, u, v, w, z induce a diamond.

For the converse, suppose that $G \cup \{v\}$ is not a diamond-free graph. Since G is diamond-free, then there are three vertices u, w, z such that together with v induce a diamond in G. If u and v are not adjacent, then u, w, z belong to some maximal clique C of G. Since v is adjacent to w, z but not to u, we obtain that v is edge-adjacent but not fully edge-adjacent to C. So, we may assume that v is adjacent to u, w, z, and that w and z are not adjacent. But then, uw and uz belong to different maximal cliques C and C'. Thus, v is edge-adjacent to C and C' and $u \in V(C) \cap V(C')$. □

We remark that part 2 of the above theorem is somehow similar to a proposition contained in [11].

Algorithm 2, which is obtained from Theorem 14, can be used to decide whether $G \cup \{v\}$ is a diamond-free graph.

Algorithm 2 Insertion of a vertex v into a diamond-free graph G

Input: a diamond-free graph G, a vertex $v \notin V(G)$, and a set $N(v) \subseteq V(G)$.
Output: a message indicating whether $G \cup \{v\}$ is diamond-free or not.

1. Unmark all previously marked vertices $w \in V(G)$.
2. For each maximal clique C of G to which v is edge-adjacent:
3. If v is not fully edge-adjacent to C, then output "$G \cup \{v\}$ is not diamond-free" and stop.
4. For each maximal clique C of G to which v is edge-adjacent:
5. If there is some marked $w \in C$, then output "$G \cup \{v\}$ is not diamond-free" and stop. Otherwise, mark every $w \in C$.
6. Output "$G \cup \{v\}$ is a diamond-free graph".

Observe that after $G \cup \{v\}$ is claimed to be a diamond-free graph in Step 6, all the vertices of every edge-adjacent maximal clique C have a mark, and every vertex $w \in C$ was traversed and marked only once. Then, v is fully-adjacent to C and no vertex of C belongs to other clique to which v is edge-adjacent. Therefore, by Theorem 14, the algorithm is correct.

Discuss the implementation of Algorithm 2. The input of the algorithm is formed by the graph G, the vertex v, and the set $N(v)$ of neighbors of v in G. An h-graph data structure is used to represent graph G. Also, the family of non singleton maximal cliques of G is stored in the dynamic data structure, each one having its own pointer. For a diamond-free graph G, denote by C_{wz} the maximal clique of G containing edge wz. For each wz, we keep a direct pointer to C_{wz}. Finally, each clique C is also associated to a variable $c(C)$, which counts the number of neighbors of v inside C, when a vertex v is inserted into G.

Before traversing each edge-adjacent maximal clique in Steps 2 and 4, we compute $N'(v)$, in $O(d(v)h(G))$ time, as in Section 5. The family $\mathcal{C} = \{C_{wz}\}_{wz \in N'(v)}$ is computed by iteratively inserting C_{wz} into $\mathcal{C}$, for every $wz \in N'(v)$. While $\mathcal{C}$ is generated, the value $|N'(v) \cap E(C_{wz})|$ can be computed by increasing $c(C_{wz})$ by 1 when wz is first traversed. This operation takes $O(1)$ time per $wz \in N'(v)$, thus it takes $O(d(v)h(G))$ total time. Once $\mathcal{C}$ is computed, each maximal clique $C \in \mathcal{C}$ is traversed.

If $c(C) = |N'(v) \cap E(C)| \neq |E(C)|$, then v is not fully edge-adjacent to C and the algorithm stops with a failure message in Step 3.

Summing up, the computations involved in Algorithm 2 can be implemented so as to run in $O(d'(v) = O(d(v)h(G))$ time. On the other hand, the data structure that represents the graph has to be also updated, and such operations also require $O(d(v)h(G))$ time. The implementation details are omitted.

Observe that if $G \cup \{v\}$ is a diamond-free graph, then its family of maximal cliques is obtained in $O(1)$ time and the maximal clique to which an edge vw belongs can be queried in $O(1)$ time.

Algorithm 2 can also be modified so as to output an induced diamond in $O(d(v)h(G))$ time, when $G \cup \{v\}$ is not diamond-free. Consider the two alternatives for the algorithm to stop in failure. First, if v is not fully edge-adjacent to $C \in \mathcal{C}$, then there is some vertex $w \in C$ that is not adjacent to v. In this case, v, w and the endpoints of an edge in $N'(v) \cap C$ induce a diamond. To find w, traverse the edges of C and query if each endpoint is adjacent to v. The first vertex that is not adjacent to v is taken as w. To find

an edge in $N'(v) \cap C$, traverse $N'(v)$ until some edge of C is reached. Thus, the certificate in this case can be found in $O(d'(v))$ time. The second reason for the algorithm to stop in failure is that v is fully edge-adjacent to C and C', and vw is marked with C' while trying to mark it with $C \neq C'$. In this case, we ought to find two non adjacent vertices of $C \cup C'$. As in the proof of Theorem 14, these two vertices together with v and w induce a diamond. To find the two vertices observe that w is the unique vertex in $C \cap C'$ and that all the vertices of $C \setminus \{w\}$ are not adjacent to the vertices of $C' \setminus \{w\}$. Thus, the two non adjacent vertices of $C \cup C'$ can be found in $O(1)$ time by traversing at most one edge of both C and C'.

8.2. Removal of vertices:

For the removal of a vertex v, note that $G \setminus \{v\}$ is always a diamond-free graph. Thus, all we need to do is to remove v from the h-graph data structure, and to remove vw from C_{vw}, for every $w \in N(v)$. As explained in Section 5, the former operation takes $O(d(v)h(G))$ time, while the latter takes $O(d(v))$ time.

8.3. Insertion of edges:

With respect to the insertion of an edge vw, graph $G \cup \{vw\}$ is diamond-free if and only if $|N(v) \cap N(w)| \leq 1$ and, if there is some $z \in N(v) \cap N(w)$, then $d_G(vz) = d_G(wz) = 0$.

We can compute $|N(v) \cap N(w)|$ in $O(d(v) + d(w))$ time, while, for $z \in N(v) \cap N(w)$, we decide in $O(1)$ time whether the maximal cliques C_{vz} and C_{wz} have exactly one edge. If so, these cliques have to merged, again in $O(1)$ time. Thus the insertion of vw takes $O(d(v) + d(w))$ time.

8.4. Removal of edges:

It remains only to examine the removal of an edge vw from the diamond-free graph G.

Clearly, the graph $G \setminus \{vw\}$ is diamond-free if and only if $d(vw) \leq 1$, i.e., if C_{vw} has at most three edges. If $d(vw) = 1$, then C_{vw} has to be split into two maximal cliques of $G \setminus \{vw\}$ in $O(1)$ time. Therefore, the removal of an edge takes $O(\min\{d_i(v), d_i(w)\} + h_i(v) + h_i(w))$ because we need to remove vw from the h-graph data structure.

Finally, we mention that the diamond-free data structure can be used in a static algorithm to test whether a graph G is diamond-free, just by iteratively inserting the vertices of G into the data structure. At each step,

the operations of greater complexity are the insertion of a new vertex into the h-graph data structure, and the computation of $N'(v)$. As we have discussed in Section 5, the total time cost for these operations is $O(\alpha(G)m)$. Thus, this algorithm runs as fast as the improvement of the algorithm by Kloks et al. discussed before.

Corollary 15. *Diamond-free graphs can be recognized in time* $O(n+\alpha(G)m)$.

9. Finding Simple, Simplicial, and Dominated Vertices

In this section, we describe dynamic algorithms for finding all the simple, simplicial and dominated vertices of a graph.

A vertex v is *dominated* by a vertex w if $N[v] \subseteq N[w]$. Equivalently, v is dominated by $w \in N(v)$ if $d(v) - d(vw) = 1$. We say that v and w are *comparable* if either v is dominated by w or w is dominated by v. If v is dominated by all its neighbors, then v is a *simplicial* vertex, while if v is a simplicial vertex and every pair of neighbors are comparable, then v is a *simple* vertex.

In [11], Kloks et al. showed how to compute the set of simplicial vertices in $O(n+m^{2\omega/(\omega+1)}) = O(n+m^{1.41})$ time, using the fast matrix multiplication algorithm.

First, we mention that with the h-graph data structure, we can find all the simple, simplicial, and dominated vertices in $O(\alpha(G)m)$ time, as follows. First, find the degree $d(vw)$ for every $vw \in E(G)$ in $O(n + \alpha(G)m)$ time, as discussed in Section 4. Next, for each vertex v, find the set of vertices $D(v)$ that dominate v, by testing whether $d(v) - d(vw) = 1$, for every $w \in N(v)$. Clearly, if $D(v) \neq \emptyset$, then v is a dominated vertex, while if $|D(v)| = d(v)$, then v is a simplicial vertex. To determine if a simplicial vertex v is simple, it is enough to check whether $z \in D(w)$, for each edge $wz \in N'(v)$ with $d(w) \leq d(z)$. As discussed in Section 5, we can traverse all the edge-neighborhoods of G in $O(\alpha(G)m)$ time, thus simple, simplicial, and dominated vertices can be found in $O(\alpha(G)m)$ time.

Next, we consider dynamic algorithms. Our aim is to show how can the h-graph data structure can be used to maintain the simple, simplicial, and dominated vertices, while vertices are inserted to or removed from a dynamic graph. Let G be a graph implemented with the h-graph data structure.

9.1. Insertion of vertices

Start by analyzing the operation of inserting vertices.

First, consider finding dominated vertices. Let D be the family of dominated vertices of graph G. Suppose that a new vertex v with neighborhood $N(v) \subseteq V(G)$ is to be inserted into G, and that we want to update D so as to store the dominated vertices of $G \cup \{v\}$. The next lemma shows how to find the new dominated vertices.

Lemma 16. *A vertex $w \neq v$ is dominated in $H = G \cup \{v\}$ if and only if at least one of the following statements is true:*

- *$w \notin N(v)$ and w is dominated in G,*
- *$d_H(w) - d_H(vw) = 1$, or*
- *$d_H(w) - d_H(wz) = 1$, for some $wz \in N'_H(v)$.*

The first step is to insert v into G and to compute $d(vw)$ for every $w \in N(v)$. Both steps take $O(d(v)h(G))$ time, as discussed in Section 5 and Theorem 12. Next, we update the set D. By the lemma above, we need not consider the vertices outside $N(v)$. To find those $w \in N(v)$ that are dominated by v, we traverse $N(v)$ while checking whether $d(w) - d(vw) = 1$. Next, we find all the other dominated vertices, by checking the values of $d(w) - d(wz)$ and $d(z) - d(wz)$, for every $wz \in N'(v)$. Finally, we remove from D those neighbors of v that are no longer dominated, and we insert v if there is some w such that $d(v) - d(vw) = 1$. Since $N'(v)$ is computed in $O(d(v)h(G))$ time, the whole procedure takes $O(d(v)h(G))$ time.

Next, consider finding the simplicial vertices. A similar procedure, as above, can be used to update the family S of simplicial vertices of G when v is inserted into G. In this case, the simplicial vertices are found as in the next lemma.

Lemma 17. *A vertex $w \neq v$ is simplicial in $H = G \cup \{v\}$ if and only if one of the following statements is true:*

- *$w \notin N(v)$ and w is simplicial in G, or*
- *w is simplicial in G, and $d(w) - d(vw) = 1$.*

Again, begin with the insertion of v into G and the computation of $d(vw)$ for every $w \in N(v)$, in $O(d(v)h(G))$ time. In this case, we update S by first traversing each $w \in N(v)$ and checking whether $w \in S$ and $d(w)-d(vw) = 1$. On the other hand, we insert v into S if and only if $d(v) - d(vw) = 1$ for every $w \in N(v)$. The time required by these operations is $O(d(v))$, once v was inserted into G. Thus, the update of S takes $O(d(v)h(G))$ time.

Next, consider finding simple vertices.

We remark that updating the family Q of simple vertices is not as simple as updating the families D and S. The reason is that we can no longer skip the vertices outside $N(v)$. To provide an efficient update of Q, we store for each vertex x, the number $\mu(x)$ of edges $wz \in N'(x)$ such that w and z are not comparable. So, x is simple if and only if x is simplicial and $\mu(x) = 0$.

We can find out the value of $\mu(v)$ in $O(d(v)h(G))$ time, by traversing every $wz \in N'(v)$ and checking whether $\min\{d(w), d(z)\} - d(wz) = 1$. The update of the values of μ is based on the following lemma.

Lemma 18. *Two vertices w and z of G are comparable in $G \cup \{v\}$ if and only if they are comparable in G and either $\{w, z\} \subseteq N(v)$ or $\{w, z\} \cap N(v) = \emptyset$.*

The update of $\mu(x)$ for every $x \neq v$ is done as in Algorithm 3.

Algorithm 3 Update of μ after the insertion of v.

Input: the h-graph data structure of $G \cup v$ and the function μ of G.
Output: none; μ is updated to be the function μ of $G \cup \{v\}$.

1. For each $w \in N(v)$:

2. For each $z \in H(w)$:

3. If $z \notin N(v)$, $d_G(w) - d_G(wz) = 1$ and $d_G(z) - d_G(wz) > 1$, then set $\mu(x) = \mu(x) + 1$, for every $x \in N(wz)$.

In Algorithm 3, the actual update of μ occurs when w, z were comparable before inserting v, but ceased to be so after the insertion. It requires $O(m)$ time in the worst case.

Next, suppose we want to iteratively apply Algorithm 3 for computing the simple vertices. Algorithm 4 describes such a procedure.

In the above algorithm, the condition of the inner loop of Algorithm 3 is executed at most once for each edge wz. Indeed, if w and z are not

Algorithm 4 Iterative update of μ for a graph G.

Input: a graph G represented with adjacency lists.
Output: the function μ of G.

1. Let $v_1, \ldots, v_n$ be an ordering of $V(G)$, and G' be an empty graph.

2. For $i = 1, \ldots, n$: insert v_i into G' while executing Algorithm 3.

comparable prior the insertion of v, then they are not comparable after the insertion of v. On the other hand, when wz meets the condition of the inner loop, we know that $d_G(w) \leq d_G(z)$. Hence, as discussed in Section 5, the time required by Algorithm 4 is $O(n+\alpha(G)m)$ for the update of the h-graph data structure and $d(vw)$. On the other hand, for the update of μ, we require time

$$O\left(n + \sum_{wz \in E(G)} \min\{d(v), d(z)\}\right) = O(n + \alpha(G)m)$$

Corollary 19. *Simple, simplicial, and dominated vertices of a graph G can be found in $O(n + \alpha(G)m)$ time.*

9.2. Removal of vertices

Next, we consider the removal of vertices.

The process for updating the sets of simple, simplicial, and dominated vertices when a vertex is removed is similar. First, consider dominated and simplicial vertices. When v is removed, again we should not consider those vertices outside $N(v)$ for the update of the sets D and S, of dominated and simplicial vertices, respectively. A vertex in $N(v)$ is dominated in $G \setminus \{v\}$ if, in $G \setminus \{v\}$, $d(w) - d(wz) = 1$ for some $z \in H(w)$, while it is simplicial in $G \setminus \{v\}$, if $L(w) = \emptyset$ and $d(w) - d(wz) = 1$ for every $z \in H(w)$. Thus, we can update D and S in $O(d(v)h(G))$ time.

Finally, consider simple vertices. For updating the set Q of simple vertices, apply Algorithm 5, below.

The above algorithm in fact performs the inverse operations of Algorithm 3. In this case, the actual update of μ occurs precisely when w and z were not comparable before removing v, but turned comparable after after the removal. It requires $O(m)$ time for completion.

Algorithm 5 Update of μ after the removal of v.

1. For each $w \in N(v)$:

2. For each $z \in H(w)$:

3. If $z \notin N(v)$, $d_G(w) - d_G(wz) = 2$ and $d_G(z) - d_G(wz) > 1$, then set $\mu(x) = \mu(x) - 1$, for every $x \in N(wz)$.

As for the iterative complexity, using arguments similar to those of Algorithm 4, it is straightforward to conclude that the overall time complexity of Algorithm 5 is $O(n + m\alpha(G))$ when it is applied while vertices are removed.

Corollary 20. *The sets of simple, simplicial and dominated vertices of a graph can be found in time* $O(n + \alpha(G)m)$

10. Recognizing Cop-Win and Strongly Chordal Graphs

In this section, we describe static algorithms for recognizing cop-win graphs and strongly chordal graphs.

The vertex removal operations described in Section 9 can be used to improve the best known algorithms for the recognition these classes.

10.1. Cop-win graphs

A *cop-win order* of a graph G is an ordering $v_1, \ldots, v_n$ of $V(G)$ such that v_i is dominated in the subgraph induced by $v_i, \ldots, v_n$, for $1 \leq i \leq n$. A graph that admits a cop-win order is a *cop-win graph*. The cop-win name comes from the fact that cop-win graphs are precisely the graphs in which a cop can always catch the robber in a pursuit game [13]. This class has been introduced in [15], cf. [2]. Cop-win graphs are also known in the literature under the name of dismantlable graphs [16], and they form an important tool in the study of clique graphs [20]. The currently best algorithms for recognizing cop-win graphs run in $O(nm)$ time or in $O(n^3/\log n)$ time [19].

A *dismantling* of a graph G is a graph H obtained by iteratively removing one dominated vertex of G, until no more dominated vertices remain. It is not hard to see that all the dismantlings of G are isomorphic. Using the h-graph data structure, we can compute the dismantling of a graph, and the cop-win order of a cop-win graph, in $O(\alpha(G)m)$ time easily. First, find the set D of dominated vertices in $O(n + \alpha(G)m)$ time (cf. above). Then, while $D \neq \emptyset$, choose a vertex D, and remove it from G while updating D as

explained above. The graph obtained after this procedure is the dismantling H of G. If H has a unique vertex, then G has a cop-win order, given by the order in which the vertices were removed by the algorithm. This algorithm takes $O(n + \alpha(G)m)$ time, as discussed above.

Corollary 21. *Cop-win graphs can be recognized in time $O(n + \alpha(G)m)$.*

10.2. Strongly chordal graphs

A *simple elimination ordering* of a graph G is an ordering $v_1, \ldots, v_n$ of $V(G)$ such that v_i is a simple vertex in the subgraph of G induced by $v_i, \ldots, v_n$. The family of graphs that admit a simple elimination ordering is precisely the family of strongly chordal graphs [9]. Strongly chordal graphs were introduced as a subclass of chordal graph for which the domination problem is solvable in polynomial time [9]. The best known algorithms for computing a simple elimination ordering run in either $O(n^2)$ or $O((n+m)\log n)$ time [14, 17, 18]. These algorithms work by finding a doubly lexical ordering of the adjacency matrix of the graph, and then testing if this sorted matrix contains some forbidden structure. Our approach, instead, is based on iteratively finding a simple elimination ordering by iteratively removing the simple vertices.

Every strongly chordal graph G has at least one simple vertex, and $G\setminus\{v\}$ is strongly chordal for every $v \in V(G)$ [9]. Thus, we can compute a simple elimination ordering of G by removing the simple vertices in any order. That is, first find the set Q of simple vertices in $O(n + \alpha(G)m)$ time (cf. above). Then, while $Q \neq \emptyset$, choose a vertex in Q, and remove it from G while updating Q with Algorithm 5. If all the vertices are removed from Q, then the removal order given by the algorithm is a simple elimination ordering of G. As discussed for Algorithm 4, the inner loop of Algorithm 5 is executed at most once for each edge (cf. above). Therefore, the simple elimination ordering is computed in $O(n + \alpha(G)m)$ time, improving the previous best algorithms for graphs with low arboricity.

Corollary 22. *Strongly chordal graphs can be recognized in time $O(n + \alpha(G)m)$.*

11. Conclusions

We have described a new data structure that is especially suitable for handling dynamic algorithms on graphs when the edge-neighborhood of each

inserted or removed vertex has to be examined. In a sense, it represents a compromise between the time required for the insertion and removal of vertices and edges, and the time required for the examination of the edge-neighborhoods. The complexity of the operations is based on concepts as the arboricity and the h-index of graphs. The data structure can also be used in static algorithms whose outputs depend on an examination of the subgraphs induced by the neighborhoods of each vertex. The paper describes several applications of the new data structure, leading to dynamic algorithms for the corresponding problems. The dynamic algorithms proposed in the paper are the first of their kind in the literature. On the other hand, the data structure has been employed also to formulate new static algorithms for the considered problems. Most of these new static algorithms improve the complexity of the existing algorithms, for graphs having low arboricity.

Besides the applications that have been considered in the paper, there are several others in which the h-graph data structure can be employed in order to formulate new dynamic and static algorithms. In particular, we mention that the data structure can be employed to obtain new algorithms some different recognition problems. These algorithms would be suited for graphs with low arboricity. A class of these recognition problem is as follows.

Let $\mathcal{C}$ be any class of graphs. Say that some graph G is $l$ocally $\mathcal{C}$, whenever all vertex neighborhoods of G induce graphs belonging to $\mathcal{C}$.

The theorem below follows from Section 3.

Theorem 23. *Let $\mathcal{C}$ be a class of graphs which can be recognized in $O(m)$ time. Then, locally $\mathcal{C}$ graphs can be recognized in $O(\alpha(G)m)$ time.*

We already used this theorem in Section 8 to conclude that diamond-free graphs can be recognized in $O(\alpha(G)m)$ time because the class of diamond-free graphs is precisely the class of locally "disjoint union of cliques" graphs. The above theorem also implies, for instance, that the classes of gem-free graphs, i.e. locally cographs, wheel-free graphs, i.e. locally forests, and $\{W_k\}_{k\geq 4}$-free graphs, i.e. locally chordal graphs, can all be recognized in $O(\alpha(G)m)$ time (see Figure 5).

Finally, we mention that the h-graph data structure can be extended so as to handle directed graphs as well. Using this extension, and considering that a transitive orientation of a directed graph is a property that can be formulated only terms of the neighborhoods of its vertices, it follows that we can recognize transitive orientations in $O(\alpha m)$ time. Consequently we can recognize comparability graphs in $O(\alpha m)$ time.

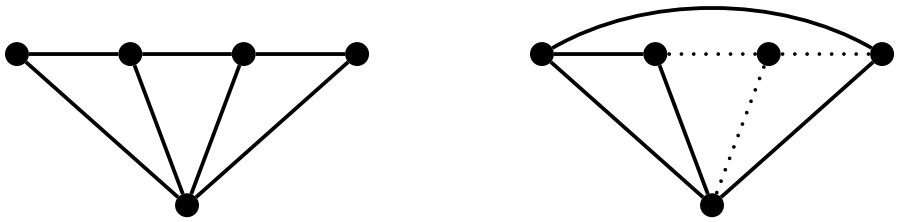

Figure 5: (a) The gem graph and (b) the wheel graphs. Wheel graphs have at least 4 vertices. The wheel graph with $k+1$ vertices is denoted by W_k, for $k \geq 3$.

NOTE ADDED TO THE REVISED VERSION

A referee has pointed out that similar equations to those presented in Section 7, for counting the number of 4-subgraphs of a graph, have been also recently and independently proposed in two papers due to appear in conferences, one by D. Eppstein, M. T. Goodrich, D. Strash, and L. Trott (COCOA' 2010), and the other by M. Kowaluk, A. Lingas, E. Lundell (SODA' 2011).

[1] N. Alon, R. Yuster, U. Zwick, Finding and counting given length cycles, Algorithmica 17 (3) (1997) 209–223.

[2] H.-J. Bandelt, E. Prisner, Clique graphs and Helly graphs, J. Combin. Theory Ser. B 51 (1) (1991) 34–45.

[3] B. Bollobás, Extremal graph theory, Dover Publications Inc., Mineola, NY, 2004, reprint of the 1978 original.

[4] N. Chiba, T. Nishizeki, Arboricity and subgraph listing algorithms, SIAM J. Comput. 14 (1) (1985) 210–223.

[5] M. Conforti, (K_4-e)-free perfect graphs and star cutsets, in: B. Simeone (Ed.), Combinatorial optimization (Como, 1986), Vol. 1403 of Lecture Notes in Math., Springer, Berlin, 1989, pp. 236–253.

[6] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, J. Symbolic Comput. 9 (3) (1990) 251–280.

[7] F. Eisenbrand, F. Grandoni, On the complexity of fixed parameter clique and dominating set, Theoret. Comput. Sci. 326 (1-3) (2004) 57–67.

[8] D. Eppstein, E. S. Spiro, The h-index of a graph and its application to dynamic subgraph statistics, in: F. K. H. A. Dehne, M. L. Gavrilova, J.-R. Sack, C. D. Tóth (Eds.), Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings, Vol. 5664 of Lecture Notes in Computer Science, Springer, 2009, pp. 278–289.

[9] M. Farber, Characterizations of strongly chordal graphs, Discrete Math. 43 (2-3) (1983) 173–189. doi:10.1016/0012-365X(83)90154-1.
URL http://dx.doi.org/10.1016/0012-365X(83)90154-1

[10] J. Fonlupt, A. Zemirline, A polynomial recognition algorithm for perfect $K_4 \setminus \{e\}$-free graphs, Rev. Maghrébine Math. 2 (1) (1993) 1–26.

[11] T. Kloks, D. Kratsch, H. Müller, Finding and counting small induced subgraphs efficiently, Inform. Process. Lett. 74 (3-4) (2000) 115–121.

[12] N-V. R. Mahadev and T. M.-Wang, On uniquely intersectable graphs, Discrete Math. 207 (1999), 149-159.

[13] R. Nowakowski, P. Winkler, Vertex-to-vertex pursuit in a graph, Discrete Math. 43 (2-3) (1983) 235–239.

[14] R. Paige and R. E. Tarjan, Three partition refinement algorithms, SIAM J. on Computing 16 (6) (1987), 973-989.

[15] T. Poston, Fuzzy Geometry, Ph.D. thesis, University of Warwick (1971).

[16] A. Quilliot, Homomorphismes, points fixes, rétractions et jeux de poursuite dans les graphes, les ensembles ordonnés et les espaces métriques, Ph.D. thesis, Université de Paris VI, France (1983).

[17] J. P. Spinrad, Doubly lexical ordering of dense 0-1 matrices, Information Proc. Letters 45 (1993), 229-235.

[18] J. P. Spinrad, Efficient graph representations, Vol. 19 of Fields Institute Monographs, American Mathematical Society, Providence, RI, 2003.

[19] J. P. Spinrad, Recognizing quasi-triangulated graphs, Discrete Appl. Math. 138 (1-2) (2004) 203–213.

[20] J. L. Szwarcfiter, A survey on clique graphs, in: C. Linhares Sales, B. Reed (Eds.), Recent Advances in Algorithms and Combinatorics, Vol. 11 of CMS Books Math./Ouvrages Math. SMC, Springer, New York, 2003, pp. 109–136.

[21] M. Talmaciu, E. Nechita, Recognition algorithm for diamond-free graphs, Informatica (Vilnius) 18 (3) (2007) 457–462.

[22] A. Tucker, Coloring perfect $(K_4 - e)$-free graphs, J. Combin. Theory Ser. B 42 (3) (1987) 313–318.

[23] V. Vassilevska, Efficient algorithms for path problems in weighted graphs, Ph.D. thesis, School of Computer Science, Carnegie Mellon University (August 2008).