Hole and Antihole Detection in Graphs*

Stavros D. Nikolopoulos† Leonidas Palios‡

Abstract

In this paper, we study the problems of detecting holes and antiholes in general undirected graphs and present algorithms for them, which, for a graph on n vertices and m edges, run in $O(n + m^2)$ time and require $O(nm)$ space; we thus provide a solution to the open problem posed by Hayward, Spinrad, and Sritharan in [12] asking for an $O(n^4)$-time algorithm for finding holes in arbitrary graphs. The key element of the algorithms is a special type of depth-first search traversal which proceeds along P_4s (i.e., chordless paths on four vertices) of the input graph. We also describe a different approach which allows us to detect antiholes in graphs that do not contain chordless cycles on 5 vertices in $O(n + m^2)$ time requiring $O(n + m)$ space. Our algorithms are simple and can be easily used in practice. Additionally, we show how our detection algorithms can be augmented so that they return a hole or an antihole whenever such a structure is detected in the input graph; the augmentation takes $O(n + m)$ time and space.

Keywords: hole, antihole, weakly chordal graph, co-connectivity.

1 Introduction

We consider finite undirected graphs with no loops or multiple edges. Let G be such a graph and let $v_0, v_1, \ldots, v_{k-1}$ be a sequence of k distinct vertices such that there is an edge from v_i to $v_{(i+1) \bmod k}$ (for all $i = 0, \ldots, k-1$), and no other edge between any two of these vertices; we say that this is a *chordless cycle* on k vertices. A *hole* is a chordless cycle on five or more vertices; an *antihole* is the complement of a hole.

Holes and antiholes have been extensively studied in many different contexts in algorithmic graph theory. Most notable examples are the weakly chordal graphs (also known as weakly triangulated graphs) [9, 10], which contain neither holes nor antiholes, and the strong perfect graph conjecture (Berge [1]), which states that a graph is perfect if and only if it contains no holes and no antiholes on an odd number of vertices (the recent proof of the conjecture by Chudnovsky *et al.* [4] has renewed the interest in the problem). Thus, finding a hole or an antihole in a graph efficiently is an important graph-theoretic problem, both on its own and as a step in many recognition algorithms.

Several algorithms for detecting holes and antiholes in graphs have been proposed in the literature. The definition of holes and antiholes implies that such algorithms can be applied without error on the biconnected components of the input graph and of its complement, respectively, instead of the entire graph. Although this approach may lead to the fast detection of holes and antiholes in graphs with small biconnected components, it does not yield any gain in the asymptotic sense.

The problem of determining whether a graph contains a chordless cycle on k or more vertices, for some fixed value of $k \geq 4$, is solved in $O(n^k)$ time (Hayward [11]); Spinrad [16] provided an improved solution taking $O(n^{k-3}M)$ time, where $M \simeq n^{2.376}$ is the time required to multiply two $n \times n$ matrices. Note that the problem of determining whether a graph contains a chordless cycle on four or more vertices can be solved in $O(n+m)$ time [9, 15, 18] (it is the well-known chordal graph recognition problem).

The algorithms of [11] and [16] can be used for the recognition of weakly chordal graphs in $O(n^5)$ and $O(n^{4.376})$ time respectively. Further progress on the weakly chordal graph recognition problem includes the $O(n^4)$-time and $O(nm)$-space algorithm of Spinrad and Sritharan [17], and the $O(m^2)$-time and $O(n+m)$-space algorithm of Hayward, Spinrad, and Sritharan [12] and of Berry, Bordat, and Heggernes [2]. It is interesting to note that the algorithm of [12] produces a hole or an antihole certificate whenever the input graph is not weakly chordal. In the same paper, the authors posed as an open problem the designing of an $O(n^4)$-time algorithm to find a hole in an arbitrary graph.

In this paper, we study the above mentioned open problem and we present two algorithms, one for the detection of holes and another for the detection of antiholes in arbitrary graphs. Both algorithms run in $O(n + m^2)$ time and require $O(nm)$ space, and rely on the detection of a cycle satisfying certain conditions in the input graph or on its complement respectively. The existence of such a cycle is checked by means of a special depth-first search traversal, which we call P_4-DFS. We also describe another algorithm for the detection of antiholes in graphs that do not contain chordless cycles

*Research partially funded by the European Union and the Hellenic Ministry of Education through EPEAEK II.

†Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece; stavros@cs.uoi.gr

‡Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece; palios@cs.uoi.gr

on five vertices: the algorithm processes each edge of the input graph in order to determine whether the endpoints of the edge participate in an antihole, and relies on the computation of the co-connected components of subgraphs of the input graph; it runs in $O(n+m^2)$ time and takes $O(n+m)$ space. The same approach yields an $O(n^2m)$-time and $O(n^2)$-space algorithm for detecting holes in graphs that do not contain chordless cycles on five vertices.

Additionally, we describe how to augment our three detection algorithms so that they return a hole or an antihole in the case where such a structure exists in the input graph; the augmented hole (antihole, respectively) detection algorithm produces a hole (an antihole, respectively) in $O(n + m)$ additional time and $O(n + m)$ space. Finally, we note that, as a by-product, our hole and antihole detection algorithms can be used for recognizing weakly chordal graphs leading to a solution that matches the best currently known time complexity for this problem.

2 Preliminaries

Let G be a finite undirected graph with no loops or multiple edges. We denote by $V(G)$ and $E(G)$ the vertex set and edge set of G. The subgraph of a graph G induced by a subset S of vertices of G is denoted by $G[S]$.

A *path* in G is a sequence of vertices $v_0v_1 \dots v_k$ such that $v_iv_{i+1} \in E(G)$ for $i = 0, 1, \dots, k-1$; we say that this is a path from v_0 to v_k and that its *length* is k. A path is called *simple* if none of its vertices occurs more than once; it is called *trivial* if its length is equal to 0. A simple path $v_0v_1 \dots v_k$ is *chordless* if $v_iv_j \notin E(G)$ for any two non-consecutive vertices v_i, v_j in the path. Throughout the paper, the chordless path on k vertices is denoted by P_k. In particular, a chordless path on 3 vertices is denoted by P_3 and a chordless path on 4 vertices is denoted by P_4. A sequence of vertices $v_0v_1 \dots v_{k-1}$ forms a *cycle* (resp. *simple cycle*) iff $v_0v_{k-1} \in E(G)$ and $v_0v_1 \dots v_{k-1}$ is a path (resp. simple path) in G; its length is equal to k. A simple cycle $v_0v_1 \dots v_{k-1}$ is said to be *chordless* if no edge v_iv_j exists in $E(G)$ such that $|i-j| \neq 1 \bmod k$. The chordless cycle on k vertices is denoted by C_k; in particular, C_5 is the chordless cycle on 5 vertices.

The *neighborhood* $N(x)$ of a vertex $x \in V(G)$ is the set of all the vertices of G which are adjacent to x. The *closed neighborhood* of x is defined as $N[x] := N(x) \cup \{x\}$. The neighborhood of a subset S of vertices is defined as $N(S) := \left(\bigcup_{x \in S} N(x)\right) - S$ and its closed neighborhood as $N[S] := N(S) \cup S$. The notion of the neighborhood can be extended to edges: for an edge $e = xy$, the *neighborhood* (*closed neighborhood*) of e is the vertex set $N(\{x, y\})$ (resp. $N[\{x, y\}]$) and is denoted by $N(e)$ (resp. $N[e]$). For an edge $e = xy$, we define the following three sets:

$$\begin{aligned} A(e; x) &= N(x) - N[y], \\ A(e; y) &= N(y) - N[x], \\ A(e) &= N(x) \cap N(y); \end{aligned}$$

clearly, these sets form a partition of the neighborhood $N(e)$ of the edge e.

We close this section by describing the co-connectivity problem which plays a crucial role in the antihole detection algorithm for graphs that do not contain a C_5, which we propose in this paper. The co-connectivity problem on a graph G is that of finding the connected components of the complement $\overline{G}$; the connected components of $\overline{G}$ are called *co-connected components* (or *co-components*) of G. The co-components of a graph G on n vertices and m edges can be computed in $O(n + m)$ time and space [8, 13, 3].

3 Detecting Holes

The hole detection algorithm relies on the result stated in the following lemma.

LEMMA 3.1. *An undirected graph G contains a hole if and only if G contains a cycle $u_0u_1 \dots u_k$, where $k \geq 4$, such that $u_iu_{i+1}u_{i+2}u_{i+3}$ for each $i = 0, 1, \dots, k-3$, and $u_{k-2}u_{k-1}u_ku_0$ are P_4s of G.*

Proof. ($\Longrightarrow$) Suppose that G contains a hole; then the vertices of the hole induce a cycle meeting the conditions of the lemma.

($\Longleftarrow$) Suppose now that G contains a cycle as described in the lemma; let $v_0v_1 \dots v_h$ be the shortest such cycle. Then, this cycle is a hole:

a) since the cycle meets the conditions of the lemma, then $h \geq 4$, which implies that the cycle is of length at least equal to 5;

b) the cycle is chordless. Suppose for contradiction that there existed chords. With each chord v_iv_j, we associate its "length," which is defined as $length(v_iv_j) = |j - i|$; let $v_\ell v_r$, where $\ell < r$, be the chord of minimum length. Note that $r \geq \ell + 4$; this follows from the fact that $r \geq 4$ (because $v_0v_1v_2v_3$ is a P_4) and the fact that $v_{r-3}v_{r-2}v_{r-1}v_r$ is a P_4. Then, $v_{r-2}v_{r-1}v_rv_\ell$ is a P_4 in G because it is a path in G, $v_{r-2}v_r \notin E(G)$ (recall that $v_{r-3}v_{r-2}v_{r-1}v_r$ is a P_4), and $v_\ell v_{r-2} \notin E(G)$ and $v_\ell v_{r-1} \notin E(G)$ for otherwise these would be chords

whose *length*-value would be smaller than that of the chord $v_\ell v_r$, in contradiction to the minimality of $length(v_\ell v_r)$. Additionally, $v_i v_{i+1} v_{i+2} v_{i+3}$ is a P_4 for all $i = \ell, \ell+1, \ldots, r-3$. Thus, the cycle $v_\ell v_{\ell+1} \ldots v_r$ would meet the conditions of the lemma; as it would be shorter than the cycle $v_0 v_1 \ldots v_h$, this would contradict the fact that the latter cycle is the shortest such cycle. Hence, the cycle $v_0 v_1 \ldots v_h$ is chordless.

Therefore, G contains a hole. ∎

Our algorithm for the detection of holes applies Lemma 3.1. In particular, it uses a special type of depth-first search traversal, which we will call *P_4-DFS*: the P_4-DFS traversal works similarly to the standard depth-first search [6], except that, in its general step, it tries to extend a P_3 *abc* into P_4s of the form *abcd*, then, for each such P_4, it proceeds extending the P_3 *bcd* into P_4s of the form *bcde*, and so on. Unlike the standard depth-first search, the P_4-DFS traversal may proceed to a vertex that has been encountered before; however, it does not need to proceed to a P_3 that has been encountered before. If the P_4-DFS has at a given moment proceeded to traverse a sequence of P_3s *abc*, *bcd*, ..., *wxy*, *xyz*, then we call the sequence of vertices $a, b, \ldots, y, z$ the *current P_4-DFS path*. Then, it is not difficult to see that the following holds:

LEMMA 3.2. *Suppose that the P_4-DFS traversal is applied on the input graph G. Then, if the current P_4-DFS path ρ is extended to a vertex which belongs to ρ, then G contains a cycle meeting the conditions of Lemma* 3.1.

Note that the cycle mentioned in Lemma 3.2 contains at least one P_4 and thus is of length at least equal to 5.

In order to exhaustively search the input graph G, the P_4-DFS traversal starts from each P_3 of G. On the other hand, in order to prevent processing a P_3 which has been encountered before, it uses an auxiliary array `not_in_hole[`$(u,v),w$`]`, where $u, v, w \in V(G)$ and u, v are adjacent in G; for each pair of adjacent vertices u, v, the array has entries `not_in_hole[`$(u,v),w$`]` as well as `not_in_hole[`$(v,u),w$`]` for every $w \in V(G)$, and hence its size is $2m \cdot n$. The entry `not_in_hole[`$(u,v),w$`]` is equal to 1 iff the vertices u, v, w induce a P_3 *uvw* of G which has been processed and found not to participate in a hole, otherwise it is equal to 0 (note that two entries of the array correspond to each P_3 *uvw*, namely, `not_in_hole[`$(u,v),w$`]` and `not_in_hole[`$(w,v),u$`]`). Additionally, in order to be able to test whether a vertex belongs to the current P_4-DFS path, the algorithm uses another auxiliary array `in_path[]` of size n; for a vertex v, `in_path[`v`]` is equal to 1 if v belongs to the current P_4-DFS path, and is 0 otherwise. Below, we give a detailed description of the algorithm when applied on a connected input graph G; the case of a disconnected input graph is discussed after the analysis of the algorithm. The algorithm assumes that G is given in adjacency list representation, from which it computes the adjacency matrix of G so that adjacency tests can be answered in constant time.

Hole-Detection Algorithm
Input: a connected undirected graph G.
Output: yes, if G contains a hole; otherwise, no.

1. Initialize the entries of the arrays `not_in_hole[]` and `in_path[]` to 0;
   compute the adjacency matrix `A[]` of G;
2. For each vertex u of G do
   2.1 `in_path[`u`]` $\leftarrow 1$;
   2.2 for each edge vw of G do
   if u is adjacent to v and non-adjacent to w
   and `not_in_hole[`$(u,v),w$`]` $= 0$
   then `in_path[`v`]` $\leftarrow 1$;
   `process`(u,v,w);
   `in_path[`v`]` $\leftarrow 0$;
   2.3 `in_path[`u`]` $\leftarrow 0$;
3. Print that G does not contain a hole.

where the procedure `process()` is as follows:

`process`(a,b,c)
1. `in_path[`c`]` $\leftarrow 1$;
2. for each vertex d adjacent to c in G do
   2.1 if d is adjacent to neither a nor b in G
   then {*abcd is a P_4 of G*}
   2.2 if `in_path[`d`]` $= 1$
   then print that G has a hole; Stop.
   else if `not_in_hole[`$(b,c),d$`]` $= 0$
   then `process`(b,c,d);
3. `in_path[`c`]` $\leftarrow 0$;
4. `not_in_hole[`$(a,b),c$`]` $\leftarrow 1$;
   `not_in_hole[`$(c,b),a$`]` $\leftarrow 1$;

It is important to observe that the description of the procedure `process()` guarantees that from a P_3 *abc* we proceed to a P_3 *bcd* only if *abcd* is a P_4 of the input graph G. Before returning, the procedure sets the corresponding entries of the array `not_in_hole[]`, thus preventing a second call to the procedure on the same P_3. Additionally, a call `process`(a,b,c) does not cause, for any depth of recursion, another call `process`(a,b,c) or `process`(c,b,a), because, for this to happen, the vertex a (respectively, c) would be encountered again; then, the condition of the if statement in Step 2.2 of `process()` would be found true and the algorithm would instantly terminate. Thus, the

procedure `process()` is called exactly once for each P_3 of G.

The correctness of the algorithm follows from Lemmas 3.1 and 3.2 and the following result.

LEMMA 3.3. *If for a P_3 abc of the input graph G, the entry* `not_in_hole[`$(a,b),c$`]` *is set to 1, then the P_3 abc does not participate in a hole of G.*

Proof. For the entry `not_in_hole[`$(a,b),c$`]` to be set to 1, a call `process(`a,b,c`)` or `process(`c,b,a`)` needs to have been made; suppose without loss of generality that this is `process(`a,b,c`)`. The proof applies induction on the number of calls to the procedure `process()` that have returned before the assignment "`not_in_hole[`$(a,b),c$`]` $\leftarrow$ 1" in Step 4 of the call `process(`a,b,c`)`.

For the basis step, let us suppose that no calls to `process()` have returned. Then, no entry of the array `not_in_hole[]` has been set to 1. Hence, no P_4 of the form $abcd$ exists in G; if it existed, either the algorithm would have terminated (if d belonged to the P_4-DFS path) or a call `process(`b,c,d`)` would have been made, which should have returned for the control to proceed to Step 4. Therefore, since no P_4 $abcd$ exists in G, the P_3 abc does not participate in a hole of G.

For the inductive hypothesis, we assume that the statement of the lemma is true for P_3s for which the corresponding entries of the array `not_in_hole[]` have been set equal to 1 after fewer than $i_0 > 0$ calls to the procedure `process()` have returned. For the inductive step, we assume that the entry `not_in_hole[`$(a,b),c$`]` corresponding to the P_3 abc has been set equal to 1 after i_0 calls to the procedure `process()` have returned, and we show that the lemma holds for the P_3 abc. Suppose, for contradiction, that this is not the case; then, abc participates in a hole of G, which implies that there exists a vertex x such that $abcx$ is a P_4 of the hole. Clearly, this vertex x has been considered in Step 2.2 of the execution of `process(`a,b,c`)`. It must be the case that `in_path[`x`]` is not equal to 1, for otherwise the algorithm would have terminated. Thus, if `not_in_hole[`$(b,c),x$`]` is equal to 0, a call `process(`b,c,x`)` is made; if `not_in_hole[`$(b,c),x$`]` is not equal to 0, then it must have been set to 1 by a preceding call `process(`b,c,x`)` or `process(`x,c,b`)`. In either case, this call to `process()` was completed before the execution of Step 4 of `process(`a,b,c`)`. Thus, the assignment "`not_in_hole[`$(b,c),x$`]` $\leftarrow$ 1" has been made after fewer than i_0 calls to `process()` have terminated; by the inductive hypothesis, we conclude that the P_3 bcx does not participate in any hole of G. This comes into contradiction with the fact that the P_4 $abcx$ is a P_4 of a hole of G. Therefore, the P_3 abc does not participate in a hole of G. Our inductive proof is complete; the lemma follows. ∎

Time and Space Complexity. Let n and m be the number of vertices and edges of the input graph G respectively. Since G is connected, then $n = O(m)$. Before analyzing the time complexity of each step of the algorithm, we turn to the procedure `process()`. We note that the procedure is called exactly once for each P_3 of G, i.e., $O(nm)$ times, and that, if we ignore the time taken by the recursive calls, a call `process(`a,b,c`)` takes $O(|N(c)| + 1)$ time by using the adjacency list of the vertex c to retrieve c's neighbors, and by using the adjacency matrix `A[]` to answer adjacency tests in constant time. Therefore, the time taken by all the calls to the procedure `process()` is $O(m^2)$, since each quadruple of vertices a, b, c, d where abc is a P_3 and d is adjacent to c is uniquely characterized by the ordered pair $((a,b),(c,d))$ where (a,b) and (c,d) are ordered pairs of adjacent vertices in G.

Step 1 of the main body of the algorithm clearly takes $O(nm)$ time. If the time taken by the calls to the procedure `process()` is ignored, Step 2 takes $O(nm)$ time; again, the adjacencies are checked in constant time by means of the adjacency matrix `A[]` of G. Step 3 takes constant time. Thus, the time complexity of the algorithm for a connected graph on n vertices and m edges is $O(m^2)$. The space needed is $O(nm)$: $O(n)$ and $O(nm)$ for the arrays `in_path[]` and `not_in_hole[]` respectively, and $O(n^2)$ for the matrix `A[]` and the adjacency list representation of the input graph.

Summarizing, we have the following result.

LEMMA 3.4. *Let G be a connected undirected graph on n vertices and m edges. Then, it can be determined whether G contains a hole in $O(m^2)$ time and $O(nm)$ space.*

The case of a disconnected input graph. If the input graph G is disconnected, we work on each of its connected components; let n_i and m_i denote the number of vertices and edges of the i-th connected component respectively. The computation of the connected components takes $O(n + m)$ time [6], while processing each of them takes $O(m_i^2)$ time. Since $\sum_i m_i = m$, we have that $O(n + m^2)$ time suffices for detecting holes in any graph on n vertices and m edges. In this case, the space needed is $O\big(\sum_i (n_i m_i)\big) = O(nm)$. Therefore, we obtain the following theorem.

THEOREM 3.1. *Let G be an undirected graph on n vertices and m edges. Then, it can be determined whether G contains a hole in $O(n+m^2)$ time and $O(nm)$ space.*

3.1 Providing a Certificate

The hole detection algorithm can be easily augmented so that it provides a certificate whenever it decides that the input graph G contains a hole. In particular, we need the following:

(i) The updating of the entries of the array `in_path[]` is done so that they reflect the position of the corresponding vertex in the current P_4-DFS path. More specifically, `in_path[`v`]` is set equal to i, if v is the i-th vertex in the P_4-DFS path; this may necessitate replacing the call `process(`a, b, c`)` by `process(`a, b, c, i`)`, if c is the i-th vertex in the path.

(ii) The vertices in the current P_4-DFS path are stored in an array `pathvertex[]` in the order they appear along the path.

(iii) If the algorithm concludes that G contains a hole, then the condition in Step 2.2 of the procedure `process()` during the execution of a call, say, `process(`a, b, c, k`)`, is found true for some vertex d; suppose that d is located in the j-th position of the current P_4-DFS path. Then, the vertices located in positions j, $j+1$, ..., k of the path form a cycle satisfying the conditions in the statement of Lemma 3.1. To isolate a hole, we call the following procedure `get_hole(`j, k`)` before terminating in Step 2.2 of `process(`a, b, c, k`)`. The procedure computes the range $[i_{min}, i_{max}]$ of indices of the entries in the array `pathvertex[]` which store the vertices inducing a hole in G.

```
get_hole(j, k)
    i_min ← j;        i_max ← k;
    i ← i_min;
    repeat
        u ← pathvertex[i];
        h ← i_max + 1;
        for each vertex x adjacent to u in G do
            if  in_path[x] ≥ i + 4
               and  in_path[x] < h
            then  h ← in_path[x];
        if  h ≤ i_max
        then  i_min ← i;        i_max ← h;
        i ← i + 1;
    until  i > i_max − 4;
    print the vertices in the entries i_min, i_min + 1,
    ..., i_max of the array pathvertex[];
```

The vertices printed induce a hole in G.

Note that, in each iteration of the repeat-until loop, at the end of the execution of the for loop, the variable h is equal to the minimum index of any entry in the subarray `pathvertex[`$i + 4 ... i_{max}$`]` storing a neighbor of `pathvertex[`i`]` in G, whenever such an entry exists, and is equal to $i_{max} + 1$ otherwise. The correctness of the computation follows from Lemma 3.5.

LEMMA 3.5. *The vertices printed by procedure* `get_hole()` *induce a hole in the input graph G.*

Proof. Let $i_{min}(t)$ and $i_{max}(t)$ denote the values of i_{min} and i_{max} at the beginning of the iteration of the repeat-until loop for $i = t$, and let $\hat{\imath}_{min}$ and $\hat{\imath}_{max}$ be their final values. Clearly, the vertices printed by `get_hole()` induce a cycle. Moreover, its length is at least equal to 5, since $i_{max}(i) \geq i_{min}(i) + 4$ for every i; note that $k \geq j + 4$ and that $h \geq i + 4$. Finally, we show that the cycle is chordless. Suppose for contradiction that there existed a chord and suppose that it were incident on the vertices `pathvertex[`ℓ`]` and `pathvertex[`r`]`, where $\hat{\imath}_{min} \leq \ell < r \leq \hat{\imath}_{max}$ and $\ell \neq \hat{\imath}_{min}$ or $r \neq \hat{\imath}_{max}$ or both; then, $r - \ell < \hat{\imath}_{max} - \hat{\imath}_{min}$. The definition of the P_4-DFS traversal implies that every four consecutive vertices in the array `pathvertex[]` form a P_4, and thus $r - \ell \geq 4$. Hence, $\ell \leq r - 4 \leq \hat{\imath}_{max} - 4$, which, along with the fact that the value of i_{max} never increases, implies that the repeat-until loop of the procedure `get_hole()` has been executed for $i = \ell$. At the end of the execution of the for loop when $i = \ell$, the variable h would not exceed r, because the vertex `pathvertex[`r`]` is adjacent to `pathvertex[`ℓ`]` in G, and $\ell + 4 \leq r \leq \hat{\imath}_{max} \leq i_{max}(\ell)$; then, the condition "$h \leq i_{max}$" would have been found true, and the variables i_{min} and i_{max} would have been set to ℓ and to an integer not exceeding r respectively. This, however, comes into contradiction with the fact that $r - \ell < \hat{\imath}_{max} - \hat{\imath}_{min}$, since the value of i_{min} never decreases and the value of i_{max} never increases. ∎

The certificate computation takes $O(n + m)$ time: note that the vertices in the array `pathvertex[]` are distinct, and that their neighbors can be accessed in constant time per neighbor using the adjacency list representation of the input graph. The space required is linear in the size of the input graph. Therefore, we have:

THEOREM 3.2. *Let G be an undirected graph on n vertices and m edges. The hole detection algorithm presented in this section can be augmented so that it provides a certificate that G contains a hole, whenever it decides so of G. The certificate computation takes $O(n + m)$ time and $O(n + m)$ space.*

4 Detecting Antiholes

Since an antihole is the complement of a hole, one can use the algorithm of the previous section on the

complement of a graph in order to determine whether it contains an antihole. Such an approach may however require $\Theta(n^4)$ time, where n is the number of vertices of the graph, since the complement may have as many as $\Theta(n^2)$ edges. Below, we present an algorithm for the detection of antiholes which applies the P_4-DFS traversal on the complement of the input graph G without however computing the complement explicitly and which takes $O(n + m^2)$ time when G has n vertices and m edges. The algorithm uses an array `not_in_antihole[`$(a, b), c$`]`, where $a, b, c \in V(G)$ and $ab \in E(G)$, and thus is of size $2m \cdot n$; `not_in_antihole[`$(a, b), c$`]` is equal to 1 iff acb is a P_3 of $\overline{G}$ which has been found not to participate in an antihole of G, and is 0 otherwise. The input graph G is assumed to be connected; if G is disconnected, then we apply the algorithm on each of G's connected components.

Antihole-Detection Algorithm
Input: a connected undirected graph G.
Output: yes, if G contains an antihole; otherwise, no.

1. Initialize the entries of arrays `not_in_antihole[]` and `in_path[]` to 0;
   compute the adjacency matrix of G;
2. For each vertex u of G do
   2.1 `in_path[`u`]` $\leftarrow 1$;
   2.2 for each edge vw of G do
   if u is adjacent to neither v nor w
   and `not_in_antihole[`$(v, w), u$`]` $= 0$
   then `in_path[`v`]` $\leftarrow 1$;
   `process(`v, u, w`)`;
   `in_path[`v`]` $\leftarrow 0$;
   2.3 `in_path[`u`]` $\leftarrow 0$;
3. Print that G does not contain an antihole.

where the procedure `process()` is as follows:

`process(`a, b, c`)`
1. `in_path[`c`]` $\leftarrow 1$;
2. for each vertex d adjacent to b in G do
   2.1 if d is adjacent to a and non-adjacent to c
   then {*$abcd$ is a P_4 of $\overline{G}$*}
   2.2 if `in_path[`d`]` $= 1$
   then print that G has an antihole; Stop.
   else if `not_in_antihole[`$(b, d), c$`]` $= 0$
   then `process(`b, c, d`)`;
3. `in_path[`c`]` $\leftarrow 0$;
4. `not_in_antihole[`$(a, c), b$`]` $\leftarrow 1$;
   `not_in_antihole[`$(c, a), b$`]` $\leftarrow 1$;

Note that for a call `process(`a, b, c`)`, a and c are adjacent in G, while b is adjacent to neither a nor c. So, if there exists a vertex d such that d is adjacent to a and b and not adjacent to c, then the vertices a, b, c, d induce the P_4 $abcd$ in $\overline{G}$.

The correctness of the algorithm is established as in the case of the hole detection algorithm of the previous section.

Time and Space Complexity. Similarly to the case of the hole detection algorithm, we obtain the result stated in Theorem 4.1; observe that if we ignore the time taken by the recursive calls, the execution of the call `process(`a, b, c`)` takes $O(|N(b)| + 1)$ time, and that each quadruple of vertices a, b, c, d where abc is a P_3 of $\overline{G}$ and d is adjacent to b in G is uniquely characterized by the ordered pair $((a, c), (b, d))$, where (a, c) and (b, d) are ordered pairs of adjacent vertices in G.

THEOREM 4.1. *Let G be an undirected graph on n vertices and m edges. Then, it can be determined whether G contains an antihole in $O(n + m^2)$ time and $O(nm)$ space.*

4.1 Providing a Certificate

Similarly to the hole detection algorithm, the above algorithm can be easily augmented so that it provides a certificate whenever it decides that the input graph G contains an antihole. In particular,

(i) we use arrays `in_path[]` and `pathvertex[]` as described in Section 3.1;

(ii) if the algorithm concludes that G contains an antihole, then the condition in Step 2.2 of the procedure `process()` during the execution of a call, say, `process(`a, b, c, k`)`, is found true for some vertex d; suppose that d is located in the j-th position of the current P_4-DFS path. Then, the vertices located in positions j, $j + 1$, ..., k of the path form a cycle in $\overline{G}$ satisfying the conditions in the statement of Lemma 3.1. To isolate an antihole, we call the following procedure `get_antihole(`j, k`)` before terminating in Step 2.2 of `process(`a, b, c, k`)`; the procedure computes the range $[i_{min}, i_{max}]$ of indices of the entries in the array `pathvertex[]` which store the vertices inducing an antihole in G.

```
get_antihole(j, k)
    i_min ← j;      i_max ← k;
    i ← i_min;
    repeat
        u ← pathvertex[i];
        h ← i + 4;
        while h ≤ i_max
          and pathvertex[h] is adjacent to u do
            h ← h + 1;
        if h ≤ i_max    {a non-neighbor was found}
        then i_min ← i;      i_max ← h;
```

$i \leftarrow i + 1$;
until $i > i_{max} - 4$;
print the vertices in the entries $i_{min}, i_{min} + 1, \ldots, i_{max}$ of the array `pathvertex[]`;

The vertices printed induce an antihole in G.

It is not difficult to see that, for each value of i, when the condition of the while loop is found false, the variable h is equal to the minimum index of any entry in the subarray `pathvertex[`$i + 4 ... i_{max}$`]` storing a non-neighbor of `pathvertex[`i`]` in G, whenever such an entry exists, and is equal to $i_{max} + 1$ otherwise. The correctness of the procedure `get_antihole()` follows from an argument similar to that used to prove Lemma 3.5; as in the case of holes, the value of i_{min} never decreases, whereas the value of i_{max} never increases.

The procedure requires $O(n + m)$ time: the initialization assignments take $O(1)$ time; during the execution of the repeat-until loop for the vertex u stored in `pathvertex[`i`]`, the condition of the while loop is checked at most $|N(u)|$ times (it is found false at the first-encountered non-neighbor of u) and thus the while loop takes $O(N(u))$ time (thanks to the adjacency matrix of G), while $O(1)$ time suffices for the remaining assignments. Since the vertices in the array `pathvertex[]` are distinct, it follows that the procedure `get_antihole()` takes $O(n + m)$ time. The space required is linear in the size of the input graph. Therefore, the following theorem holds:

THEOREM 4.2. *Let G be an undirected graph on n vertices and m edges. The antihole detection algorithm presented in this section can be augmented so that it provides a certificate that G contains an antihole, whenever it decides so of G. The certificate computation takes $O(n + m)$ time and $O(n + m)$ space.*

5 Detecting Antiholes in Graphs that do not Contain a C_5

Antiholes in general graphs can be detected by taking advantage of the following property:

LEMMA 5.1. *Let G be an undirected graph. Then, G contains an antihole if and only if there exists an edge $e = xy$ of G and a vertex $u \in V(G) - N[e]$ such that in the complement of the subgraph of G induced by $N(e) \cap N(u)$ there exists a path from a vertex in $A(e; x)$ to a vertex in $A(e; y)$.*

Proof. ($\Longrightarrow$) Suppose that G contains an antihole and let this be the complement of the hole $v_0 v_1 \ldots v_k$, where $k \geq 4$. Then, the vertices v_1 and v_k are adjacent in G; let e be the edge of G connecting them. Then, $v_0 \in V(G) - N[e]$, $v_2 \in A(e; v_k)$, $v_{k-1} \in A(e; v_1)$, and $\{v_2, \ldots, v_{k-1}\} \subseteq N(e) \cap N(v_0)$; these and the fact that the sequence $v_{k-1}, \ldots, v_2$ induces a path in $\overline{G}$ imply that the conditions of the lemma hold for the edge $v_1 v_k$ of G and the vertex v_0.
($\Longleftarrow$) Suppose now that there exists an edge $e = xy$ of G and a vertex $u \in V(G) - N[e]$ such that in the complement of the subgraph $G[N(e) \cap N(u)]$ there exists a path from a vertex in $A(e; x)$ to a vertex in $A(e; y)$. Let $p_0 p_1 \ldots p_k$ be a shortest such path; that is, $p_0 \in A(e; x) \cap N(u)$, $p_k \in A(e; y) \cap N(u)$, $p_i \in A(e) \cap N(u)$ for all $i = 1, \ldots, k-1$, and $p_i p_j \in E(G)$ for $0 \leq i < j - 1 \leq k - 1$. Then, the subgraph $G[\{x, u, y, p_0, p_1, \ldots, p_k\}]$ of G is the complement of a chordless cycle of length at least 5; in other words, G contains an antihole. ∎

Lemma 5.1 readily implies an antihole detection algorithm which for a graph G on n vertices and m edges runs in $\Theta(nm^2)$ time in the worst case: for the appropriate pairs of an edge $e = xy$ and a vertex u, we compute the co-components of the subgraph $G[N(e) \cap N(u)]$ and check whether any of them contains vertices from both $A(e; x)$ and $A(e; y)$; as there may be as many as $\Theta(nm)$ such pairs, and for each one of them $\Theta(n+m)$ time may be needed and suffices for the above mentioned computations, the overall time complexity is $\Theta(nm^2)$. An improved algorithm can be obtained for graphs not containing C_5s, for which the following fact holds.

FACT 5.1. *Let G be an undirected graph which does not contain a C_5, and let $e = xy$ be an edge of G. Then, for every pair of vertices a and b such that $a \in A(e; x)$, $b \in A(e; y)$, and $(N(a) \cap N(b)) - N[e] \neq \emptyset$, it holds that a and b are adjacent in G.*

Proof. Let w be any vertex in $(N(a) \cap N(b)) - N[e]$; clearly, $wa, wb \in E(G)$, and $wx, wy \notin E(G)$. If the vertices a and b were not adjacent in G, then the subgraph of G induced by a, x, y, b, and w would be a C_5, a contradiction. ∎

In light of Fact 5.1, for any edge $e = xy$ and any vertex $u \in V(G) - N[e]$ of a graph G that does not contain a C_5, any path from a vertex in $A(e; x)$ to a vertex in $A(e; y)$ in the complement of the subgraph of G induced by $N(e) \cap N(u)$ is of length at least 2 and contains a vertex in $A(e)$. Then, instead of computing such a path we need only determine if there exist vertices $a \in A(e; x) \cap N(u)$, $b \in A(e; y) \cap N(u)$, and $v \in A(e) \cap N(u)$ such that v, a belong to the same co-component of the subgraph $G[N(x) \cap N(u)]$, and v, b belong to the same co-component of the subgraph $G[N(y) \cap N(u)]$; see Figure 1: C and D denote the

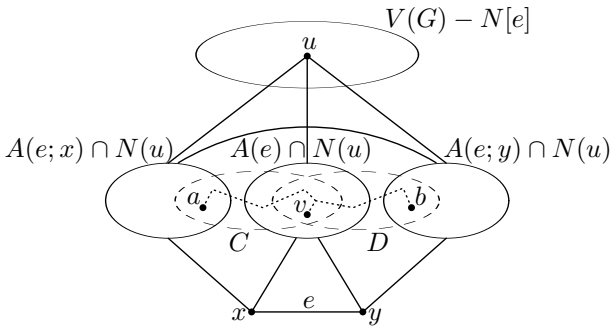

Figure 1

co-components of the subgraphs $G[N(x) \cap N(u)]$ and $G[N(y) \cap N(u)]$ containing v and a, and v and b respectively, and the dotted paths indicate chordless paths in the complements of these subgraphs. We show next that the existence of such vertices v, a, and b is equivalent to the existence of an antihole in G.

LEMMA 5.2. *Let G be an undirected graph which does not contain a C_5. Then, G contains an antihole if and only if there exists an edge $e = xy$ of G and vertices $u \in V(G) - N[e]$, $a \in A(e;x) \cap N(u)$, $b \in A(e;y) \cap N(u)$, and $v \in A(e) \cap N(u)$ such that a, v belong to the same co-component of the subgraph $G[N(x) \cap N(u)]$, and b, v belong to the same co-component of the subgraph $G[N(y) \cap N(u)]$.*

Proof. ($\Longrightarrow$) Suppose that G contains an antihole. Since G does not contain a C_5, the antihole is of length at least 6; let it be the complement of the hole $v_0 v_1 \dots v_k$ ($k \geq 5$). The vertices v_1 and v_k are adjacent in G; if e is the edge of G connecting them, then, $v_0 \in V(G) - N[e]$, $v_2 \in A(e; v_k) \cap N(v_0)$, $v_{k-1} \in A(e; v_1) \cap N(v_0)$, and $\{v_3, \dots, v_{k-2}\} \subseteq A(e) \cap N(v_0)$. Since $k \geq 5$, it is easy to see that the conditions of the lemma hold for $v_1, v_k, v_0, v_{k-1}, v_2, v_3$ in place of x, y, u, a, b, v respectively.
($\Longleftarrow$) Suppose now that there exists an edge $e = xy$ of G and vertices u, a, b, v as described in the statement of the lemma. Then, in the complement of the subgraph of G induced by $N(e) \cap N(u)$ there exists a path from vertex $a \in A(e;x)$ to vertex $b \in A(e;y)$. Then, by Lemma 5.1, G contains an antihole. ∎

We give below a detailed description of the antihole detection algorithm for a connected input graph G; for disconnected input graphs, we apply the algorithm on each of their connected components.

Antihole-Detection Algorithm for Graphs that do not contain a C_5
Input: a connected undirected graph G that does not contain a C_5.
Output: yes, if G contains an antihole; otherwise, no.

1. For each vertex u of G do
   - 1.1 for each vertex w not adjacent to u in G do
     - 1.1.1 compute the set $N_{u,w} = N(u) \cap N(w)$;
     - 1.1.2 compute the co-components of $G[N_{u,w}]$;
     - 1.1.3 store the set $N_{u,w}$ as a list of vertex records, ordered by vertex index, where each vertex $z \in N_{u,w}$ is associated with the representative `cc(`$N_{u,w}; z$`)` of the co-component to which it belongs;
   - 1.2 for each edge $e = xy$ of G s.t. $x, y \notin N[u]$ do {$x, y \notin N[u]$ *is equivalent to* $u \in V(G) - N[e]$}
     - 1.2.1 {*mark the co-components of* $G[N_{u,x}]$} {*containing a vertex in* $A(e;x)$}
       for each vertex $w \in N_{u,x}$ do `mark1[`w`]` $\leftarrow 0$;
       for each vertex $w \in N_{u,x} - N_{u,y}$ do {*mark the representative*} `mark1[cc(`$N_{u,x}; w$`)]` $\leftarrow 1$;
     - 1.2.2 {*mark the co-components of* $G[N_{u,y}]$} {*containing a vertex in* $A(e;y)$}
       for each vertex $w \in N_{u,y}$ do `mark2[`w`]` $\leftarrow 0$;
       for each vertex $w \in N_{u,y} - N_{u,x}$ do {*mark the representative*} `mark2[cc(`$N_{u,y}; w$`)]` $\leftarrow 1$;
     - 1.2.3 for each vertex $v \in N_{u,x} \cap N_{u,y}$ do
       if `mark1[cc(`$N_{u,x}; v$`)]` $= 1$ and `mark2[cc(`$N_{u,y}; v$`)]` $= 1$ then print that G contains an antihole; Stop;
2. Print that G does not contain an antihole.

The correctness of the algorithm follows from Lemma 5.2: for a vertex u of G and an edge $e = xy$ such that $x, y \notin N[u]$, then $A(e;x) \cap N(u) = N_{u,x} - N_{u,y}$, $A(e;y) \cap N(u) = N_{u,y} - N_{u,x}$, and $A(e) \cap N(u) = N_{u,x} \cap N_{u,y}$; moreover, the condition "if `mark1[cc(`$N_{u,x}; v$`)]` $= 1$ and `mark2[cc(`$N_{u,y}; v$`)]` $= 1$" and the fact that the vertex v belongs to $N_{u,x} \cap N_{u,y}$ imply that in the complement of $G[N_{u,x}]$ there exists a path from v to a vertex in $A(e;x)$ and that in the complement of $G[N_{u,y}]$ there exists a path from v to a vertex in $A(e;y)$.

Time and Space Complexity. Let n and m be the number of vertices and edges of the input graph G; since G is connected, $n = O(m)$. Step 1.1.1 can be completed in $O(n)$ time, while the construction of $G[N_{u,w}]$ and the computation of its co-components can be done in $O(|N_{u,w}|^2)$ time [8, 13, 3]. Since $|N_{u,w}| \leq \min\{|N(u)|, |N(w)|\}$, we have that $|N_{u,w}|^2 \leq$

$|N(u)| \cdot |N(w)|$; thus, for a vertex u of G, Step 1.1.2 takes $O(n) + \sum_w O\big(|N(u)| \cdot |N(w)|\big) = O(m\,|N(u)|)$ time[1]. The construction of the list storing $N_{u,w}$ in Step 1.1.3 can be done in $O(|N(u)|)$ time by traversing the adjacency list of u, by collecting those vertices that belong to $N_{u,w}$, and by updating the co-component representative information; the ordering by vertex index comes for free, had we sorted the adjacency lists of the vertices in G by vertex index, something which can be achieved in $O(n+m)$ time using radix sorting during a preprocessing phase.

The sorting of the lists representing the sets $N_{u,w}$ implies that determining which vertices belong to $N_{u,x} - N_{u,y}$, $N_{u,y} - N_{u,x}$, and $N_{u,x} \cap N_{u,y}$ can be achieved by simply traversing the lists for $N_{u,x}$ and $N_{u,y}$ in lockstep fashion. Then, each execution of Step 1.2 takes $O(|N(u)|)$ time, since $N_{u,x}, N_{u,y} \subseteq N(u)$. Thus, for a vertex u of G, Step 1.2 takes $\sum_e O(|N(u)|) = O(m\,|N(u)|)$ time. Step 2 takes constant time. In total, the entire execution of the algorithm on G takes $O\big(\sum_u O(n+m+m\,|N(u)|)\big) = O(m^2)$ time.

Let us now turn to the space complexity of the algorithm. The adjacency list representation of the input graph requires $O(n+m)$ space. For an iteration of the for loop in Step 1, we need: $O(n)$ space to store $N_{u,w}$ and the re-indexing arrays, and $O(n+m)$ space to store $G[N_{u,w}]$, both reusable in each iteration of the for loop in Step 1.1; $\sum_w O(1+|N_{u,w}|) = O(n+m)$ space to store the list representations of the sets $N_{u,w}$ for all $w \in V(G) - N[u]$; $O(n)$ space for the arrays `mark1[]` and `mark2[]`, reusable in each iteration of the for loop in Step 1.2. This space can be reused from iteration to iteration of the for loop in Step 1, so that the space complexity of the algorithm is $O(n+m)$.

In summary, our antihole detection algorithm runs in $O(m^2)$ time using $O(n+m)$ space when applied on a connected undirected graph on n vertices and m edges. If the input graph is disconnected, then we apply the algorithm on each of its connected components. Since the connected components of a graph can be computed in time and space linear in the size of the graph [6] and since these components are pairwise vertex- and edge-disjoint, we obtain the following result.

THEOREM 5.1. *Let G be an undirected graph on n vertices and m edges which does not contain a C_5. Then, it can be determined whether G contains an antihole in $O(n+m^2)$ time and $O(n+m)$ space.*

[1] Note that working on the subgraph $G[N_{u,w}]$ requires re-indexing of vertices, i.e., mapping the indices $1,\ldots,n$ of vertices in G to the indices $1,\ldots,|N_{u,w}|$ of vertices in $G[N_{u,w}]$ and vice versa; this can be done by using two arrays of $O(n)$ total space which take $O(n)$ time to initialize and constant time to answer each re-indexing request.

5.1 Providing a Certificate

Like the previous algorithms, this algorithm too can be augmented so that it provides a certificate whenever it decides that the input graph G contains an antihole. In particular, if G contains an antihole, then whenever the algorithm finds that, for a vertex u and an edge $e = xy$ of G such that $x, y \notin N[u]$, there exists a vertex $v \in N_{u,x} \cap N_{u,y}$ for which `mark1[cc(`$N_{u,x}$`; v)]` $= 1$ and `mark2[cc(`$N_{u,y}$`; v)]` $= 1$, it executes the following in Step 1.2.3 before terminating:

(i) computes the subgraph $G[N(e) \cap N(u)]$;

(ii) uses a dummy vertex s and makes it adjacent to all the vertices of the subgraph except for those in $N_{u,x} - N_{u,y}$;

(iii) runs BFS on the *complement* of the resulting graph starting at s until a vertex, say, b, in $N_{u,y} - N_{u,x}$ is encountered;

It is not difficult to see that if the path on tree edges from s to b in the BFS-tree of Step (iii) is $sv_1v_2\ldots v_kb$, then the vertices $x, u, y, v_1, \ldots, v_k, b$ induce an antihole in G of length at least 6 (since G does not contain a C_5, then $k \geq 2$ in accordance with Fact 5.1).

The computation of the adjacency list representation of the subgraph $G[N(e) \cap N(u)]$ can be done in $O(n+m)$ time and space by using a copy of the adjacency list representation of G and by removing from it all unnecessary lists and vertex records. The addition of the dummy vertex s can be done in $O(n)$ time and space. Executing BFS on the complement of the resulting graph can be done in time and space linear in the size of the graph, i.e., in $O(n+m)$ time and space (see [8, 13, 14]). Finally, the path on tree edges needed to complete the antihole can be easily obtained in time linear in its length if the BFS-tree is represented by means of parent pointers. Therefore, we have:

THEOREM 5.2. *Let G be an undirected graph on n vertices and m edges which does not contain a C_5. The antihole detection algorithm presented in this section can be augmented so that it provides a certificate that G contains an antihole, whenever it decides so of G. The certificate computation takes $O(n+m)$ time and space.*

Remark. Since an antihole is the complement of a hole and the complement of a C_5 is also a C_5, one can detect whether a graph G, which does not contain a C_5, contains a hole by applying the above algorithm on its complement $\overline{G}$; this results into an $O(n^4)$-time and $O(n^2)$-space algorithm. If however the operation of the algorithm on $\overline{G}$ is interpreted in terms of G so that

$\overline{G}$ is not constructed explicitly, then it can be shown that the algorithm runs in $O(n^2m)$ time and requires $O(n^2)$ space. This result indicates that the same approach results in a hole detection algorithm which in the worst case proves asymptotically more time- and space-consuming than the corresponding antihole detection algorithm. This seems to be due to the fact that checking whether a graph contains an antihole of length k requires that certain $\Theta(k^2)$ edges exist and that certain k edges are missing, whereas in the case of a hole of length k, one needs to verify that k edges exist and $\Theta(k^2)$ edges are missing; in the former case, the cost of checking the non-existence of the k edges can be paid for by the $\Theta(k^2)$ existing edges, something which does not hold in the latter case.

6 Concluding Remarks

We have presented algorithms for detecting holes and antiholes in general undirected graphs. For an input graph on n vertices and m edges, both algorithms run in $O(n + m^2)$ time and require $O(nm)$ space. The algorithms can be augmented so that they return a hole or an antihole (whenever such a structure exists in the graph) in $O(n + m)$ additional time and space. We have also described an antihole detection algorithm for graphs not containing a C_5 which runs in $O(n + m^2)$ time and requires only $O(n + m)$ space.

The obvious open problem is to design algorithms for finding a hole and/or an antihole in general graphs with improved time and/or space complexity; note that all the P_3s participating in P_4s of a graph on n vertices and m edges can be computed in $O(nm)$ time [14]. It is worth mentioning that $o(n + m^2)$-time algorithms for both problems would imply an improvement on the currently best algorithms for recognizing weakly chordal graphs [12, 2].

We also pose as an open problem the construction of $O(n + m^2)$-time algorithms for detecting whether a graph contains a C_5. None of our algorithms seems to be modifiable to handle this special case while maintaining the $O(n + m^2)$ time complexity. We note that, due to our antihole detection algorithm for graphs that do not contain a C_5, an $O(n + m^2)$-time and $O(n + m)$-space algorithm for detecting a C_5 would imply an antihole detection algorithm for general graphs of the same time and space complexity.

Finally, in light of the "strong perfect graph theorem" [4], it would be very interesting to come up with efficient algorithms for the detection of odd-length holes and/or odd-length antiholes in general graphs. For a graph on n vertices, the currently fastest algorithms for these problems run in $O(n^9)$ time [5, 7].

References

[1] C. Berge, Färbung von Graphen deren sämtliche bzw. deren ungerade Kreise starr sind, *Wiss. Zeitschrift*, Mathematisch-Naturwissenschaftliche Reihe, Martin-Luther-Univ. Halle-Wittenberg, 114–115, 1961.

[2] A. Berry, J.-P. Bordat, and P. Heggernes, Recognizing weakly triangulated graphs by edge separability, *Nordic J. Computing* **7**, 164–177, 2000.

[3] K.W. Chong, S.D. Nikolopoulos, and L. Palios, An optimal parallel co-connectivity algorithm, *Theory of Computing Systems* (to appear); *Technical Report 27-02*, Dept of Computer Science, Univ. of Ioannina, 2002.

[4] M. Chudnovsky, N. Robertson, P.D. Seymour, and R. Thomas, The strong perfect graph theorem, *preprint*, 2002.

[5] M. Chudnovsky and P.D. Seymour, Recognition algorithm for Berge graphs, *preprint*, 2003.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd edition), MIT Press, Inc., 2001.

[7] G. Cornuéjols, X. Liu, and K. Vušković, Perfect graph recognition, *Proc. 44th IEEE Symp. on Foundations of Computer Science (FOCS 2003)*, 2003.

[8] E. Dahlhaus, J. Gustedt, and R.M. McConnell, Efficient and practical algorithms for sequential modular decomposition, *J. Algorithms* **41**, 360–387, 2001.

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

[10] R.B. Hayward, Weakly triangulated graphs, *J. Comb. Theory Ser. B* **39**, 200–208, 1985.

[11] R.B. Hayward, Two classes of perfect graphs, *PhD Thesis*, School of Computer Science, McGill Univ., 1987.

[12] R.B. Hayward, J. Spinrad, and R. Sritharan, Weakly chordal graph algorithms via handles, *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA 2000)*, 42–49, 2000.

[13] H. Ito and M. Yokoyama, Linear time algorithms for graph search and connectivity determination on complement graphs, *Inform. Process. Letters* **66**, 209–213, 1998.

[14] S.D. Nikolopoulos and L. Palios, Recognizing P_4-comparability graphs, *Proc. 28th Int. Workshop on Graph Theoretic Aspects of Computer Science (WG 2002)*, 355–366, 2002.

[15] D.J. Rose, R.E. Tarjan, and G.S. Lueker, Algorithmic aspects of vertex elimination on graphs, *SIAM J. Computing* **5**, 266–283, 1976.

[16] J.P. Spinrad, Finding large holes, *Inform. Process. Letters* **39**, 227–229, 1991.

[17] J.P. Spinrad and R. Sritharan, Algorithms for weakly triangulated graphs, *Discrete Applied Math.* **59**, 181–191, 1995.

[18] R.E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, *SIAM J. Computing* **13**, 566–579, 1984.