HOLE AND ANTIHOLE DETECTION IN GRAPHS

Stavros D. Nikolopoulos and Leonidas Palios

15– 2002

Preprint, no 15 – 02 / 2002

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece

Hole and Antihole Detection in Graphs

Stavros D. Nikolopoulos and Leonidas Palios
Department of Computer Science, University of Ioannina
P.O. Box 1186, GR-45110 Ioannina, Greece
e-mail: {stavros, palios}@cs.uoi.gr

Abstract: In this paper, we study the problems of detecting holes and antiholes in general undirected graphs, and we present algorithms for these problems. For an input graph G on n vertices and m edges, both algorithms 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 [7] 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 G; 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 an induced 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) [1, 4, 5], which contain neither holes nor antiholes, and the perfect graph conjecture [4], which states that a graph is perfect if and only if it contains no holes and no antiholes on an odd number of vertices. 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 given graph on n vertices and m edges contains a hole on k or more vertices, for some fixed value of $k \geq 4$, is solved in $O(n^k)$ time (Hayward [6]). Spinrad [12] reduced the time complexity of the problem to $O(n^{k-3}M)$, 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

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 $N[x] := \{x\} \cup N(x)$. The neighborhood of a subset A of vertices is defined as $N(A) := \left(\bigcup_{x \in A} N(x)\right) - A$ and its closed neighborhood as $N[A] := A \cup N(A)$. 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:

$A(e; x) = N(x) - N(y)$,

$A(e; y) = N(y) - N(x)$,

$A(e) = N(x) \cap N(y)$;

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 [3, 8, 9].

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 $u_1u_2 \ldots u_k$, where $k \geq 5$, such that $u_iu_{i+1}u_{i+2}u_{i+3}$ for each $i = 1, 2, \ldots, k-3$, and $u_{k-2}u_{k-1}u_ku_1$ 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 $v_1v_2 \ldots v_h$ be the shortest such cycle. Then, this cycle is a hole:

a) by definition, it is of length at least equal to 5;

b) it 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_kv_ℓ, where $k < \ell$, be the chord with minimum length. Note that $\ell \geq k+4$; this follows from the fact that $\ell \geq 5$ (because $v_1v_2v_3v_4$ is a P_4) and the fact that $v_{\ell-3}v_{\ell-2}v_{\ell-1}v_\ell$ is a P_4. Then, $v_{\ell-2}v_{\ell-1}v_\ell v_k$ is a P_4 in G because it is a path in G, $v_{\ell-2}v_k \notin E(G)$ (note that $v_{\ell-3}v_{\ell-2}v_{\ell-1}v_\ell$ is a P_4), and $v_kv_{\ell-2} \notin E(G)$ and $v_kv_{\ell-1} \notin E(G)$ for otherwise these would be chords whose *length*-value would be smaller than that of the chord v_kv_ℓ, in contradiction to the minimality of $length(v_kv_\ell)$. Additionally, $v_iv_{i+1}v_{i+2}v_{i+3}$ is a P_4 for all $i = k, k+1, \ldots, \ell-3$. Thus, the cycle $v_kv_{k+1} \ldots v_\ell$ would meet the conditions of the lemma and would be shorter, which would contradict the fact that the cycle $v_1v_2 \ldots v_h$ is the shortest such cycle. Hence, the cycle $v_1v_2 \ldots v_h$ 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 [2], 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 has proceeded to traverse a sequence

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) should 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 if 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 us first assume that the input graph 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 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 pair of edges *ab* and *cd* of the graph 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

Then, for the next value of i, the condition of the while loop is no longer true, and the procedure `get_hole()` reports that the vertices $u_2, u_3, u_4, u_5, u_6, u_7$ form a hole, which is correct. It must be noted that the hole reported is not necessarily the shortest hole in the graph of Figure 1.

The correctness of the computation follows from Lemma 3.4.

Lemma 3.4. *The vertices printed by procedure* `get_hole()` *induce a hole in the input graph G.*

Proof: Clearly, these vertices induce a cycle. Moreover, its length is at least equal to 5; note that $h \geq i + 4$, which implies that $i_{max} \geq i_{min} + 4$. Finally, we show that the cycle is chordless. Suppose for contradiction that there existed a chord and suppose that it was incident on the vertices `pathvertex[`ℓ`]` and `pathvertex[`r`]`, where $i_{min} \leq \ell < r \leq i_{max}$ and at least one of $\ell \neq i_{min}$ and $r \neq i_{max}$ holds, which implies that $r - \ell < i_{max} - i_{min}$. The definition of the P_4-DFS traversal implies that every four consecutive vertices in the array `pathvertex[]`; form a P_4, and thus $r - \ell \geq 4$. Then, $\ell \leq r - 4 \leq i_{max} - 4$, which, along with the fact that the value of i_{max} never increases, implies that the while loop of the procedure `get_hole()` has been executed for $i = \ell$. Since $\ell + 4 \leq r \leq i_{max}$, the condition of the if statement would be found true; then, i_{min} would have been set to ℓ and i_{max} to an integer not exceeding r. This, however, is a contradiction, in light of the fact that $r - \ell < i_{max} - i_{min}$ for the final values of i_{min} and i_{max} and that the value of i_{min} and of i_{max} never decreases and never increases respectively. ∎

It is not difficult to see that the certificate computation requires $O(n + m)$ time; note that all vertices in the array `pathvertex[]` are distinct, their neighbors can be accessed in constant time per neighbor using the adjacency list representation of the input graph, and that finding whether a vertex belongs to the current P_4-DFS path and its position in it takes constant time using the (modified) array `in_path[]`. The space required is linear in the size of the input graph. Therefore, we have the following result:

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 necessitate $\Theta(n^4)$ 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 for a graph on n vertices and m edges takes $O(n+m^2)$ time and $O(nm)$ space; the algorithm applies the P_4-DFS traversal on the complement of the input graph without however computing the complement explicitly, and takes advantage of the fact that a P_3 abc in the complement of a graph G is equivalent to the pair (ac, b), where ac is an edge of G and b is not adjacent to a nor to b in G.

Antihole-Detection Algorithm

Input: an undirected graph G on n vertices and m edges.

Output: yes, if G contains an antihole; otherwise, no.

1. Initialize the entries of the 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`]` $\leftarrow 1$;

(ii) the vertices in the current P_4-DFS path are stored in a stack `pathvertex[]`;

(iii) 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 uses an auxiliary array `mark[1..`n`]` and computes the range $[i_{min}, i_{max}]$ of indices of the array `pathvertex[]` which store the vertices inducing an antihole in G.

```
get_antihole()
    i_min ← j;      i_max ← k;
    for i = 1, 2, ..., n do
        mark[i] ← 0;      {initialize to 0}
    i ← i_min;
    repeat
        for each vertex x adjacent to the vertex in pathvertex[i] in G do
            if in_path[x] > 0      {note that x = pathvertex[in_path[x]]}
            then mark[in_path[x]] ← 1;      {mark neighbors in path}
        h ← i + 4;
        while h ≤ i_max and mark[h] = 1 do
            h ← h + 1;
        if h ≤ i_max
        then i_min ← i;      i_max ← h;
        for each vertex x adjacent to the vertex in pathvertex[i] in G do
            if in_path[x] > 0
            then mark[in_path[x]] ← 0;      {clear array mark[]}
        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 an antihole in G.

It is not difficult to see that the body of the repeat-until loop for each value of i locates the smallest-index entry of the array `pathvertex[]` corresponding to a non-neighbor of `pathvertex[`i`]`; note that, at the beginning of each iteration of the repeat-until loop, the entries of the array `mark[]` are all equal to 0. The correctness of the procedure `get_antihole()` follows from an argument similar to that used to prove Lemma 3.4; 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 $O(n)$ time. During the execution of the repeat-until loop for a vertex, say, u, stored in `pathvertex[`i`]`, the for loops take $O(N(u))$ time (thanks to the adjacency list representation of G), so does the while loop as well (since it stops at the first non-neighbor of u encountered), while $O(1)$ time suffices for the remaining assignments. Since all 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.*

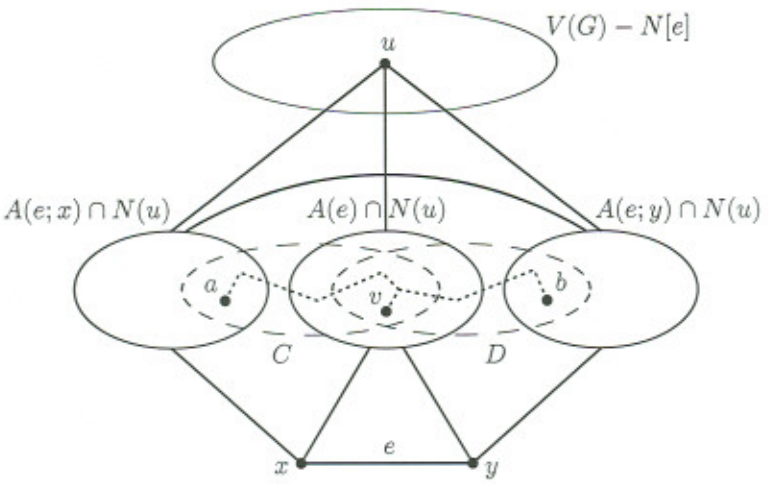

Figure 2

1.2 for each edge $e = xy$ of G such that $x, y \notin N[u]$ do
  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
      `mark1[cc(`$N_{u,x}$`;`w`)]` $\leftarrow 1$; {*mark the representative*}
  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
      `mark2[cc(`$N_{u,y}$`;`w`)]` $\leftarrow 1$; {*mark the representative*}
  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 5.1 and from the fact that for an edge $e = xy$ of G and a vertex $u \in V(G) - N[e]$ the following hold:

(i) $A(e;x) \cap N(u) = N_{u,x} - N_{u,y}$, $A(e;y) \cap N(u) = N_{u,y} - N_{u,x}$, and $A(e) \cap N(u) = N_{u,x} \cap N_{u,y}$;

(ii) the condition "if `mark1[cc(`$N_{u,x}$`;`v`)]` $= 1$ and `mark2[cc(`$N_{u,y}$`;`v`)]` $= 1$" along with the fact that the vertex v belongs to $N_{u,x} \cap N_{u,y}$ implies 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 $A(e;y)$; thus, in the complement of the subgraph of G induced by $N(u) \cap N(e)$ there exists a path from a vertex in $A(e;x)$ to a vertex in $A(e;y)$.

Time and Space Complexity. We will show that the above mentioned antihole-detection algorithm runs in $O(n+m^2)$ time; it suffices to show that this time complexity holds for connected input graphs. Step 1.1.1 can be completed in $O(n)$ time, while the construction of $G[N_{u,v}]$ and the computation of its co-components [3, 8, 9] can be done in $O(|N_{u,v}|^2)$ time. Since $|N_{u,v}| \leq \min\{|N(u)|, |N(v)|\}$, we have that $|N_{u,v}|^2 \leq |N(u)| \cdot |N(v)|$; thus, for a vertex u of G, Step 1.1.2 takes $O(n) + \sum_v O(|N(u)| \cdot |N(v)|) = O(m\,|N(u)|)$ time. (Note that working on the subgraph $G[N_{u,v}]$ requires re-indexing of vertices; this can be done in constant time per re-indexing request using two

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 however results into an $O(n^4)$-time and $O(n^2)$-space algorithm. The time complexity can be improved if the operation of the algorithm on $\overline{G}$ is interpreted in terms of G so that $\overline{G}$ is not constructed explicitly.

In general terms, the algorithm processes all the P_3s of G; for each such P_3 xuy, it tries to determine whether there exists a vertex w that is not adjacent to u, x, or y, and there exists a path from w in $G[M_{u,x}]$ to a vertex adjacent to y and a path from w in $G[M_{u,y}]$ to a vertex adjacent to x, where $M_{u,x}$ (resp. $M_{u,y}$) is the set of vertices which are adjacent neither to u nor to x (resp. neither to u nor to y), i.e., $M_{u,x} = \big(V(G) - N[u]\big) \cap \big(V(G) - N[x]\big)$ and $M_{u,y} = \big(V(G) - N[u]\big) \cap \big(V(G) - N[y]\big)$. Note that such a vertex w exists iff there exists a chordless path $v_1 v_2 \ldots v_k$, where $k \geq 3$, such that $v_1 \in N(y) - \big(N(u) \cup N(x)\big)$, $v_k \in N(x) - \big(N(u) \cup N(y)\big)$, and $v_i \notin N(u) \cup N(x) \cup N(y)$ for all $i = 2, \ldots, k-1$; this is equivalent to the vertices $x, u, y, v_1, \ldots, v_k$ inducing a hole of length at least equal to 6.

In detail, the algorithm is given below. It is a variant of the antihole algorithm presented earlier in this section, where all the "adjacencies" have been replaced by "non-adjacencies" and vice versa.

Hole-Detection Algorithm for Graphs not containing a C_5

Input: an undirected graph G on n vertices and m edges which does not contain a C_5.

Output: yes, if G contains a hole; otherwise, no.

1. For each vertex u of G do

   1.1 for each vertex v adjacent to u in G do
   - 1.1.1 compute the set $M_{u,v} = (V(G) - N[u]) \cap (V(G) - N[v])$;
   - 1.1.2 compute the connected components of $G[M_{u,v}]$;
   - 1.1.3 store the set $M_{u,v}$ as a list of vertex records, ordered by vertex index, where each vertex $w \in M_{u,v}$ is associated with the representative `comp`$(M_{u,v}; w)$ of the component to which it belongs;

   1.2 for each vertex x adjacent to u in G do
   for each vertex y adjacent to u and not adjacent to x in G do
   {*mark the conn.components of $G[M_{u,x}]$ containing a neighbor of y in G*}
   for each vertex $w \in M_{u,x}$ do
   `mark1[`w`]` $\leftarrow 0$;
   for each vertex $w \in M_{u,x} - M_{u,y}$ do
   if w is adjacent to y in G
   then `mark1[comp(`$M_{u,x}; w$`)]` $\leftarrow 1$; {*mark the representative*}
   {*mark the conn.components of $G[M_{u,y}]$ containing a neighbor of x in G*}
   for each vertex $w \in M_{u,y}$ do
   `mark2[`w`]` $\leftarrow 0$;
   for each vertex $w \in M_{u,y} - M_{u,x}$ do
   if w is adjacent to x in G
   then `mark2[comp(`$M_{u,y}; w$`)]` $\leftarrow 1$; {*mark the representative*}
   for each vertex $v \in M_{u,x} \cap M_{u,y}$ do
   if `mark1[comp(`$M_{u,x}; v$`)]` $= 1$ and `mark2[comp(`$M_{u,y}; v$`)]` $= 1$
   then print that G contains a hole; Stop;

2. Print that G does not contain a hole.

References

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

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

[3] E. Dahlhaus, J. Gustedt, and R.M. McConnell, Efficient and practical modular decomposition, *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)*, 26–35, 1997.

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

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

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

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

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

[9] S.D. Nikolopoulos and L. Palios, A co-connectivity algorithm with application to the parallel recognition of weakly triangulated graphs, *Technical Report 32-01*, Department of Computer Science, University of Ioannina, 2001.

[10] S.D. Nikolopoulos and L. Palios, Recognizing P_4-comparability graphs, *Proc. 28th Intern. Workshop on Graph Theoretic Aspects of Computer Science (WG'02)*, 2002.

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

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

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

[14] 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.