Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (2000) pp 42-49

Weakly Chordal Graph Algorithms via Handles*

Ryan B. Hayward† Jeremy Spinrad‡ R. Sritharan§

Abstract

We use the notion of *handle*, introduced by Hayward, to improve algorithms for weakly chordal graphs. For recognition we reduce the time complexity from $O(n^2m)$ to $O(m^2)$ and the space complexity from $O(n^3)$ to $O(n+m)$, and also produce a hole or antihole if the input graph is not weakly chordal. For the optimization problems clique, independent set, coloring, and clique covering, we reduce the time complexity from $O(n^4)$ to $O(nm)$.

1 Introduction and Motivation

P_k and C_k denote respectively the induced path and cycle with k vertices. $\overline{C}_k$ is the complement of C_k. A *hole* is an induced cycle with five or more vertices and an *antihole* is the complement of a hole. We use n and m respectively for the number of vertices and edges of a graph. *Sees* and *misses* mean "is adjacent to" and "is not adjacent to" respectively. A vertex *misses an edge* if it misses both vertices of the edge. For a subset S of the vertex set $V(G)$ of a graph G, $G[S]$ is the subgraph induced by S and $N(S)$ is the neighborhood of S, namely the vertices of $V(G) - S$ which see some vertex of S.

A graph is *weakly chordal* (also called weakly triangulated) if it contains no holes and no antiholes. Weakly chordal graphs, introduced by Hayward [2], are a well studied class of perfect graphs. Hayward, Hoàng, and Maffray characterized weakly chordal graphs via the presence of a *two-pair*, namely a pair of nonadjacent vertices such that every induced path between them has exactly two edges. Their theorem is:

THEOREM 1.1. [5] *A graph is weakly chordal if and only if each induced subgraph either is a clique or contains a two-pair of the subgraph.*

A number of algorithms on weakly chordal graphs work by repeatedly finding a two-pair $\{x, y\}$ and modifying the neighborhoods of x and/or y. This is the basis of the previously best algorithms for recognizing weakly chordal graphs [8] and for solving a variety of optimization problems on weakly chordal graphs [8, 5] (namely weighted and unweighted versions of maximum clique, minimum vertex coloring, maximum independent set and minimum clique covering) The fastest algorithm for finding a two-pair [1] runs in $O(nm)$ time and works on general graphs. In this paper we give a structural result specifying where a two-pair can be located in a weakly chordal graph. While this does not lead to an improvement in the time to find a single two-pair, it does lead to an improvement in the time to find a sequence of two-pairs in a graph as the graph is modified during weakly chordal recognition and optimization algorithms.

The previously best recognition algorithm for weakly chordal graphs [8] takes $O(n^4)$ time and $O(nm)$ space. We improve on this algorithm by taking only $O(m^2)$ time and $O(n+m)$ space, and producing as a certificate a hole or antihole whenever the input graph is not weakly chordal (the previous algorithm does not do this). This is analogous to the well known $O(m+n)$ time chordal graph recognition algorithm [6], which can also produce in this time an induced cycle of length four or more whenever the input graph is not chordal.

The previously best optimization algorithms for weakly chordal graphs [8], based on the algorithms of [5] and an $O(nm)$ time algorithm for finding a two-pair [1], repeat the following process: find a two-pair and modify the graph, either by adding an edge to a two-pair or by collapsing a two-pair to form a single vertex. These algorithms take $O(n^2m)$ and $O(n^4)$ time in the unweighted and weighted cases respectively. Our algorithms can solve the unweighted optimization problems, which only involve collapsing two-pairs, in $O(nm)$ time.

Our algorithms rely on the following notion due to Hayward [3, 4]: a *handle* in a graph G is a proper vertex subset H with size at least two, such that $G[H]$ is connected, some component $J \neq H$ of $G - N(H)$ satisfies $N(J) = N(H)$, and each vertex of $N(H)$ sees at least one vertex of each edge of $G[H]$.

J is called a *cohandle* of H. Notice that $N(H)$ is a

*Supported by NSERC, NSF, URC/Indiana State University

†Department of Computing Science, University of Alberta, Edmonton Canada T6G 2H1, hayward@cs.ualberta.ca

‡Department of Computer Science, Vanderbilt University, Nashville TN 37235, spin@vuse.vanderbilt.edu

§Computer Science Department, University of Dayton, 300 College Park, Dayton OH 42469-2160, rst@cps.udayton.edu

minimal separator of H and J.

THEOREM 1.2. [3, 4] *A graph has a handle if and only if the graph has a* $\overline{P}_3$*, and a handle and its cohandle can be found in O(nm) time.*

A *co-two-pair* of a graph, or simply a *co-pair*, is a two-pair of the complement of the graph. Our algorithms are based on searching for co-pairs. A set X of vertices is maintained with the property that a co-pair of G can be found in X whenever G is weakly chordal. X is refined until each edge in $G[X]$ induces a co-pair of G. The graph is then modified as required by the top-level (recognition or optimization) algorithm, and the co-pair search resumes in the modified graph. The advantage of our method is that when the co-pair search resumes in the modified graph, vertices are considered in essentially the same order as in the original search. Thus unnecessary recomputation can be avoided by maintaining a stack of vertices eliminated from X. Upon search resumption, it suffices to restore X by popping the stack back to the first level at which X contains an edge.

We condense the relevant algorithms from [3, 4] into the following algorithm whose details are needed for the analysis of our algorithms. We use *with* as a synonym for "containing".

Algorithm *find-handle*
Input: A graph G
Output: Either a message 'no handle' or
a handle H of G and a cohandle J of H

begin
/* *initialization* */
search for vertex v and edge e so that v misses e
if no v, e **then** return 'no handle' and exit **endif**
$Y \leftarrow$ component of $G - N(e)$ with v
$X \leftarrow$ component of $G - N(Y)$ with e
/*$N(X) = N(Y)$ minimally separates X and Y*/

/* *handle finding* */
while some v in $N(X)$ misses some e in X **do**
$Y' \leftarrow$ component of $G - N(e)$ with v
$X' \leftarrow$ component of $G - N(Y')$ with e /*(+)*/
push vertices of $X - X'$ onto the stack
$X \leftarrow X'$ $Y \leftarrow Y'$ **endwhile**
$H, J \leftarrow X, Y$
end *find-handle*

We also use an operation called *restarting*, namely reentering *find-handle* in the handle finding phase after the modification of either the graph or the set X. When restarting we assume that we have access to the stack which was used when *find-handle* last exited.

A key fact [3, 4] used in proving the correctness of *find-handle* is that X' is a proper subset of X after the step marked (+). Thus *find-handle* deletes at least one vertex from X at a cost of $O(m)$ during each iteration of the while loop. The initialization phase takes $O(m)$ time. An initial v and e can be found if and only if G is not a complete multipartite graph. A simple complete multipartite recognition algorithm is to pick any vertex x, let C be $V - \{x\} - N(x)$, and check whether $N(y) = N(x)$ for every y in C. If so, remove C and repeat the process. If not, some z has been found in $N(y) - N(x)$ or $N(x) - N(y)$, so $\{x, y, z\}$ induces $\overline{P}_3$ and yields the desired v and e.

2 Finding co-pairs

Given a graph, our approach in finding a co-pair is to find a handle of the graph, and then a handle of the handle, and so on, until we cannot find a handle in the last handle. It turns out that if a graph is weakly chordal then every edge of the last handle induces a co-pair of the graph. A handle of a handle may not be a handle of the graph, but that property is not needed. After performing operations on a co-pair as required by the relevant recognition or optimization algorithm, we search for the next co-pair by restarting *find-handle*, thus doing less work than in the original call to *find-handle*.

Algorithm *find-pair*
Input: A weakly chordal graph G with at least one edge
Output: A co-pair of G

begin
$j \leftarrow 0$ $H_0 \leftarrow V(G)$
while $G[H_j]$ has a handle H **do**
$j \leftarrow j + 1$ $H_j \leftarrow H$
endwhile
output any edge of $G[H_j]$
end *find-pair*

To prove the correctness of *find-pair* we use the following lemma.

LEMMA 2.1. *Let H be a handle of a weakly chordal graph G and let $\{x, y\}$ be a co-pair of $G[H]$. Then $\{x, y\}$ is a co-pair of G.*

Proof. Let J be a cohandle of H in G, let $I = N(H) = N(J)$, and let $R = V(G) - H - I$. Suppose that $\{x, y\}$ is a co-pair in $G[H]$ but not in G.

Then in $\overline{G}$ there is an induced path $P = (x \ldots y)$ with at least four vertices. P does not intersect R, since in $\overline{G}$ each vertex in R sees both x and y. Thus P intersects I. P has no subpath (uvw) such that u

and w are in H but v is in I, for otherwise in G some v of I misses edge (u, w) of $G[H]$ and H is not a handle, contradiction. Thus in $\overline{G}$ P has at least one edge of I.

In $\overline{G}$ consider a subpath $P' = (x_2x_3x_4 \ldots x_r)$ of P with $r \geq 4$ such that x_2 and x_r are in H but x_3 through x_{r-1} are in I. Observe that x_3 misses x_4 in G. Since I is a minimal separator for H and J in G and since G has no holes, in G every two nonadjacent vertices of I have a common neighbor in J. In particular x_3 and x_4 see some x_1 of J in G. Thus in $\overline{G}$ x_1 sees x_2, x_1 misses x_3, x_1 misses x_4, and so $(x_1x_2x_3x_4)$ is a P_4. Let x_s be the first vertex in P' after x_4 such that in $\overline{G}$ x_1 sees x_s; x_s exists since in $\overline{G}$ x_1 sees x_r. Then in $\overline{G}$ $\{x_1, x_2, \ldots, x_s\}$ induces a hole of size at least five and G is not weakly chordal, contradiction. □

THEOREM 2.1. *Algorithm find-pair is correct.*

Proof. Assume that *find-pair* terminates after performing p loop iterations. Then $G[H_p]$ has no $\overline{P}_3$, since a graph with a $\overline{P}_3$ has a handle. Thus $G[H_p]$ is a complete multipartite graph, so every edge of $G[H_p]$ induces a co-pair of $G[H_p]$. Thus by Lemma 2.1 every edge of $G[H_p]$ induces a co-pair of $G[H_{p-1}]$, since H_p is a handle of $G[H_{p-1}]$. Continuing this argument, every edge of $G[H_p]$ induces a co-pair of G. □

THEOREM 2.2. *Algorithm find-pair runs in O(nm) time.*

Proof. The time spent by an execution of *find-pair* is the total of the time spent by the resulting calls of *find-handle*. Since *find-handle* is called at most n times, the total time spent in its initialization phase over these calls is $\mathrm{O}(nm)$. The total time spent in its handle finding phase is also $\mathrm{O}(nm)$, since each loop iteration takes $O(m)$ time and with each iteration at least one vertex is deleted. □

3 Recognizing weakly chordal graphs

Since our algorithm uses the basic structure of the $\mathrm{O}(n^4)$ algorithm [8] for recognizing weakly chordal graphs, we review that first. Given an arbitrary graph G as input, the algorithm in [8] repeatedly finds a two-pair and adds an edge between the vertices of the two-pair. When the algorithm eventually fails to find a two-pair, the original graph is weakly chordal if and only if the final graph is a clique.

A *co-pair edge* is the edge induced by a co-pair. Our algorithm can be thought of as the execution of the above algorithm in the complement: repeatedly find a co-pair and delete the induced edge (but not the vertices). Since a graph is weakly chordal if and only its complement is weakly chordal, the input graph is weakly chordal if and only if our algorithm deletes all of its edges. We use *find-pair* as the fundamental subroutine. After a co-pair is found and its edge deleted, the next co-pair is found by restarting *find-pair* if the current set X still contains an edge and by popping back to the previous set X otherwise. In this section we show that our recognition algorithm is correct and takes $\mathrm{O}(m^2)$ time.

Algorithm *wc-recognize*
Input: A graph G
Output: 'yes' if G is weakly chordal, otherwise 'no'

```
begin
  H ← V(G)
  while (G[H] has a co-pair) do
    {x, y} ← find-pair(G[H])
    if {x, y} not co-pair of G then return 'no'
    else
      delete (x, y) from G
      if G[H] has no edge then
        if stack empty then return 'yes'
        else H ← H∪ set on top stack endif endif
    endif
    restart on each connected component of H
  endwhile
end
```

It is not difficult to test in $\mathrm{O}(n+m)$ time whether an edge induces a co-pair, as we now explain. Nonadjacent vertices form a two-pair if and only if removing their common neighbors leaves the vertices in different components, so adjacent vertices form a co-pair if and only if removing their common nonneighbors leaves the vertices in different components of the complement. Given two vertices, it is easy to remove their common nonneighbours in $\mathrm{O}(n)$ time, and to determine in $\mathrm{O}(n+m)$ time whether in the reduced graph the vertices are in the same component of the complement, for example by starting from one of the two vertices and constructing a breadth first search tree of the reduced graph's complement. Since testing whether an edge induces a co-pair takes $\mathrm{O}(n+m)$ time, and since each edge is deleted immediately after it is tested, the total time spent on this task over the whole algorithm is $\mathrm{O}(m^2)$.

Let e be a co-pair edge of G returned by *find-pair* and let G' be $G - e$. Recall that *find-pair* maintains a set X such that e is an edge of $G[X]$. We consider three possibilities: $G'[X]$ is connected, $G'[X]$ is disconnected but has at least one edge, and $G'[X]$ has no edges.

LEMMA 3.1. *Let $G'[X]$ be connected. If G is weakly chordal then every co-pair of $G[X]$ is a co-pair of G'.*

Proof. Recall that X is a handle of G. Suppose that we start algorithm *find-handle* on G' with any edge of $G[X]$ as our initial edge e and an initial vertex v from the cohandle Y of X. It is not difficult to see that at the end of the initialization stage, the set X found by algorithm *find-handle* is the same as our set X. The algorithm will find Y as the cohandle, with $N(Y)$ consisting of all separators of handles which contained Y and X as well as the separator between Y and X, with X as a handle in this graph. Since X remains a handle in G', Lemma 2.1 says that we may continue our search for co-pairs of G' by looking within the subgraph induced by X. Thus, we restart our algorithm with X as our current set. □

The situation is quite similar if G' is disconnected but contains at least one edge. Each connected component of $G[X]$ could be produced as X in G' by the same argument given above. Thus we restart our algorithm looking for co-pairs separately within each connected component of $G[X]$.

LEMMA 3.2. *Let e be the last edge in $G[X]$ and let X_2 be the union of X with the top of the stack. If G is weakly chordal then every co-pair of each connected component of $G[X_2]$ is a co-pair of G'.*

Proof. Restart the algorithm with any connected component C of X_2 as our current X set. We show that C could have been obtained as the set X by an appropriate set of choices in algorithm *find-pair* on G'.

Consider any handle H already found by the algorithm such that X is a subset of H. H - X is connected; *find-handle* removes vertices from the X set only if they are adjacent to vertices already removed from the X set. Thus, H remains a handle in G', with the only possible change being the removal of vertices of X which become isolated with respect to H and do not affect our task of finding co-pairs of G.

Thus, we only need to consider a single call to procedure *find-handle*. No edge of X could have been chosen as e except in the last step; this would lead to finding the handle X immediately. Run algorithm *find-handle* using the same choices of v and e made originally. When computing the connected component which contains v, the only vertices which the algorithm can use but could not use previously are vertices of X. However, any vertex of X can only be reached via a path through $N(X)$. No vertex of $N(X)$ is added to Y in the original algorithm, so no vertex of X can ever be added to Y in this way. Thus, the only change from the previous run of the algorithm which can occur is that some set X_i found previously can become disconnected.

First, we will ignore vertices which are isolated in X_2, and argue that nonisolated vertices will never go into separate connected components. There must be some path between the components in X, since X was connected. Consider a shortest path $(x_1'x_1 \ldots x_2x_2')$ between components of X_2, where x_1 and x_2 are in X, x_1' and x_2' are in X_2 - X, and the subpath P from x_1 to x_2 only uses vertices of X. P must be empty, since if there is an edge between x_1 and x_2 then x_1' would have to be adjacent to at least one endpoint, either connecting two components of X_2 or making a shorter path between components. Since x_1' and x_2' were among the last set of vertices removed from X in the earlier iterated procedure, this was caused by a refinement using some v and e, where e was an edge from X. Both x_1' and x_2' are in $N(e)$ for any edge e of X, so these are removed from X only if they are in the set $N(Y)$. The shortest path from x_1' to x_2' through Y gives us a hole of length at least five together with x_1 and x_2.

Thus, the only difference that can occur while refining G' is that isolated vertices of X_2 may go into a different connected component. This will only affect the algorithm if for some isolated vertex i which has already been separated from the component containing nonisolated vertices of X_2, a refinement step of G chooses v and an edge (i, j) as its initial vertex and edge, where j is in $N(X)$. Vertex j cannot have edges to any vertices of X other than i, or i would not be isolated at this stage. Thus, X must induce a star centered at i, since j must see every edge in X. Let x',j' be any edge of X_2 with x in X and j' in V - X. If j is adjacent to j', i will be in the same component as x' at this stage, while if j and j' are nonadjacent we can complete the P_4 (j, i, x', j') to be a hole by using only vertices of the cohandle. Therefore, each step of the refinement procedure of G' will match the same step in the refinement of G. □

THEOREM 3.1. *Algorithm wc-recognize correctly determines whether G is weakly chordal in $O(m^2)$ time.*

Proof. The correctness of the algorithm is a consequence of the preceding lemmas. Each time a set X is refined to a set X', the edges of $G[X']$ will never be in a subgraph together with edges outside of $G[X']$. We may think of each step as subdividing a set of edges: each refinement step increases the number of subsets of edges, so there at most m refinement steps. Since each refinement step takes $\mathrm{O}(m)$ time, the total time spent by the algorithm is $\mathrm{O}(m^2)$. □

4 On finding a proof that a graph is not weakly chordal

The recognition algorithm of the previous section, like the algorithm of [8] it is based on, does not provide the most natural evidence that a graph is not weakly chordal, namely a hole in either the graph or its complement. In this section, we provide an algorithm to find a hole or antihole in a graph which is not weakly chordal, such that the running time of the algorithm is dominated by the running time of the recognition algorithm. The first vertex pair found which is not a co-pair of the graph gives us a starting position for finding the hole or antihole easily. By contrast, if the $O(n^4)$ recognition algorithm of [8] is used, we get no information about where the hole might be if the input graph contains no two-pairs.

Let us consider the behavior of the recognition algorithm when a graph G that is not weakly chordal is given as input. The algorithm takes $O(n^4)$ time and turns G into a graph that is not a clique and has no two-pairs. Let us refer to this resulting graph that has no two-pairs as G'. It is an obvious consequence of Theorem 1.1 that G' has either a hole or an antihole. More importantly, it follows from the correctness of the recognition algorithm that every hole and antihole of the input graph is also present in G' and vice versa. Therefore, if we were to look for holes or antiholes in the input graph G, it suffices to look for them in the graph G' constructed as above. This is precisely what our algorithm does. In the following, we refer to the recognition algorithm from [8] as *wt-recognition*.

Our algorithm takes the graph G given as input and runs *wt-recognition* on $\overline{G}$. If $\overline{G}$ turns into a clique the algorithm reports that the input is weakly chordal. Otherwise, from the graph G with no co-pair the algorithm in $O(n^3)$ time either finds a proper vertex subset H such that $G[H]$ has no co-pair or finds a hole or an antihole in G. In the former case, the algorithm proceeds to work on $G[H]$ to find a hole or antihole; in the latter case the algorithm obviously terminates. We give a description of our algorithm below and then prove its complexity.

Algorithm *find-hole-or-antihole*
Input: A graph G
Output: Either the message 'G weakly chordal' or
a hole or antihole of G

begin
wt-recognition$(\overline{G})$
if ($\overline{G}$ is a clique) **then** output 'G weakly chordal'
else *find-proof*(G) **endif**
end *find-hole-or-antihole*

Algorithm *find-proof*
Input: A graph G with no co-pair
Output: A hole or antihole of G

begin
/* G has a $\overline{P}_3$ since G has no co-pair */
find handle H of G and cohandle J of H

/* by handle definition, $\overline{G}[H]$ is not a clique */
if $G[H]$ has a co-pair $\{x, y\}$ of $G[H]$ **then**
apply the algorithm in the proof of Theorem 4.1
else
/* $H \subset V(G)$ and $G[H]$ has no co-pair */
find-proof$(G[H])$ **endif**
end *find-proof*

The ideas used in the proof of the following theorem are similar to those used in the proof of Lemma 2.1.

THEOREM 4.1. *Let G be a graph with no co-pair, let H be a handle of G, and let $\{x, y\}$ be a co-pair of $G[H]$. Then a hole or an antihole of G can be found in $O(n^2)$ time.*

Proof. Let J, I, R be as defined in the proof of Lemma 2.1, namely J is a cohandle of H, $I = N(H) = N(J)$, and $R = V(G) - H - I$.

Since $\{x, y\}$ is not a co-pair, in $\overline{G}$ there is an induced path with at least four vertices between x and y. Choose such a path P as follows: in $\overline{G}$ delete the common neighbors of x and y and find a shortest path between x and y in the resulting graph.

In $\overline{G}$, as every vertex in R sees both x and y, the induced path P cannot involve vertices from R. Moreover, since $\{x, y\}$ is a co-pair of G, P *must* involve vertices from I.

Now, P cannot have a segment uvw on three vertices such that u and w are in H but v is in I. Otherwise, in G, vertex v of I will miss both the endpoints of the edge $[u, w]$ of H, contradicting the assumption that H is a handle. Therefore, in $\overline{G}$, P must have at least an edge in I.

In $\overline{G}$, compute a segment $P' = x_2x_3x_4 \ldots x_r$, $r \geq 4$, of P such that x_2 and x_r are in H but x_3 through x_{r-1} are in I; observe that x_3 misses x_4 in G.

Suppose in G, x_3 and x_4 do not have a common neighbor in J. Compute P^*, a shortest path in G between x_3 and x_4 such that all vertices of P^* except x_3 and x_4 belong to J; observe that P^* must have at least 3 edges. Then, compute P^{**}, a shortest path in G between x_3 and x_4 such that all vertices of P^{**} except x_3 and x_4 belong to H. Glue P^* with P^{**} to get a hole in G.

On the other hand, suppose x_3 and x_4 see vertex x_1 of J in G. Then, in $\overline{G}$, x_1 sees x_2, x_1 misses x_3, and x_1 misses x_4, making $x_1x_2x_3x_4$ a P_4. In $\overline{G}$, let x_s be the first vertex that comes after x_4 in P' such that x_1 sees x_s; there must be such a vertex as x_1

sees x_r in $\overline{G}$. Then, the set of vertices $\{x_1, x_2, \ldots, x_s\}$, induce a hole of size at least five in $\overline{G}$. Every step of the above algorithm takes $O(n^2)$ time and there are a constant number of steps in the algorithm making the time complexity $O(n^2)$. □

The correctness of algorithm *find-hole-or-antihole* follows from our discussions and the proof of the theorem above.

THEOREM 4.2. *Algorithm find-hole-or-antihole runs in $O(n^4)$ time.*

Proof. As *wt-recognition* runs in $O(n^4)$ time, we only need to prove that *find-proof*(G) takes $O(n^4)$ time. Note that in each step of *find-proof*, the algorithm either decides to iterate on a proper induced subgraph $G[H]$ of G, or terminates after spending $O(n^2)$ time (see Theorem 4.1). Now, the decision to work on $G[H]$ involves computing a handle followed by looking for a two-pair in the complement of the graph induced by the handle; each of these can be done in $O(n^3)$ time [1, 3]. Therefore, algorithm *find-proof* performs at most n iterations each of which takes $O(n^3)$ time making its time complexity $O(n^4)$. □

5 Optimization Problems

In this section, we show that it is possible to use algorithm *find-pair* to improve the time complexity of unweighted optimization problems on weakly chordal graphs.

The algorithm for solving unweighted independent set, coloring, clique, and clique cover problems on weakly chordal graphs was first discovered in [5] and is based on the following operation: repeatedly find a two-pair x,y, and 'identify' x and y by collapsing them to a single vertex z with $N(z) = N(x) \cup N(y)$. This works for the clique and coloring problems; for independent set and clique cover, the algorithm is run on the complement graph, which is also weakly chordal. The running time is dominated by at most n two-pair finding iterations; using the algorithm of [1] this takes $O(n^2m)$ time. We will reduce the time complexity to $O(nm)$.

Since we find a two-pair of the complement in our algorithm, the effect of an identification operation of a two-pair x, y in $\overline{G}$ is to remove the vertices x and y, and add a new vertex z such that $N(z) = N(x) \cap N(y)$.

Algorithm *wc-opt* can be defined as follows. If x,y is not the last edge of the current handle, we simply restart our handle finding procedure with z replacing x and y in the current X set; if x,y is the only edge of the handle, let z replace x and y, pop the stack to add the last set of vertices deleted from X to our current X set, and restart the handle-finding procedure.

We will show that *wc-opt* can be implemented to run in $O(nm)$ time. This optimization algorithm implementation is more complex than our previously discussed $O(m^2)$ time recognition algorithm implementation.

The correctness of *wc-opt* relies on the following invariant: for any set X created by algorithm *wc-opt*, there is a set Y of V - X - $N(X)$ such that $G[Y]$ is connected and every vertex of $N(X)$ in the current handle is adjacent to a member of Y.

Note that this invariant is not affected by either the identification of co-pairs in X, which can potentially remove vertices from the separator set and into the set Y, or by adding vertices back to the set X; in the latter case, we simply revert to an earlier set Y of vertices.

The proofs below are essentially equivalent to Lemma 2.1. For convenience, we will call a set a *recursive handle* if this set was found to be a handle at some point during the running of the algorithm.

Let G' be obtained from G via the identification of a co-pair $\{x, y\}$.

LEMMA 5.1. *Let H_0 be a recursive handle of a weakly chordal graph G. If $\{u, v\}$ is a co-pair of $G'[H_0]$, then $\{u, v\}$ is a co-pair of G'.*

Proof. Assume that $G[H_i]$ is a handle of $G[H_{i+1}]$ for i = 0,...,k. Consider an induced path P from u to v of length greater than two in $\overline{G'}$. This path must leave H_0, since $\{u, v\}$ is a co-pair of $G'[H_0]$.

Let i be the largest index such that P uses a vertex which is in H_i, but not H_{i-1}. Let h_{i_1} be the first vertex from H_i - H_{i-1} on P. Note that h_{i_1} must be in $N(H_{i-1})$, since h_{i_1} is adjacent to at least one of u, v. Let g be the vertex which precedes h_{i_1} on P, and let h_{i_2} be the vertex which succeeds h_{i_1} on P. Since h_{i_1} cannot miss an edge in H_{i-1}, h_{i_2} must be in H_i - H_{i-1}.

Vertices h_{i_1} and h_{i_2} must have a common neighbor s in V - $N(H_{i-i})$; this follows from the invariant above and the fact that G has no holes. Thus, there is a hole in $\overline{G'}$, of the form s, g, h_{i_1}, h_{i_2},..,j, where j is the first vertex after h_{i_2} on P which is nonadjacent to s.

Thus, G' is not weakly chordal, contradiction, since G' was obtained from G by identifying a co-pair. □

THEOREM 5.1. *Algorithm wc-opt is correct.*

Proof. Let G be a weakly chordal graph, and let u,v be the first pair of vertices found by algorithm *wc-opt* which are not a co-pair of the current graph G'.

By the preceding lemma, we only need to concern ourself with refinement procedures in a single recursive handle. However, within a single recursive handle, the proof of lemma 1 can be repeated essentially unchanged using the invariant given above, so every vertex pair found must be a co-pair of G'. □

We now show that the algorithm can be implemented to run in $\mathrm{O}(nm)$ time. In previous arguments, our time bound was derived from the fact that a single refinement of the set X took $\mathrm{O}(m)$ time. There are several steps which take $\mathrm{O}(m)$ time; finding vertices v and e to start the refinement procedure, finding the connected component C of v in $V-N(e)$, marking neighbors of C, and finding the connected component of e in the graph induced by $X-N(C)$. We will show that if we ignore a total cost of $\mathrm{O}(nm)$, a single refinement step can be made to run in $\mathrm{O}(m_X)$ time, where m_X is the number of edges in the subgraph induced by X. We will then use the new bound on an individual step to get an $\mathrm{O}(nm)$ upper bound on the entire work done in the optimization algorithm.

We will maintain some extra information to speed up these steps. Keep the current set $N(X)$ as an ordered list, with vertices added to $N(X)$ being placed at the end of this list, and maintain with each member u of $N(X)$ the number of neighbors of u in X. We also maintain for each vertex x in X the number of neighbors of x in $N(X)$.

THEOREM 5.2. *Algorithm wc-opt can be implemented to run in O(nm) time.*

Proof. Our first step which takes $\mathrm{O}(m)$ time in a straightforward implementation is finding a vertex v from $N(X)$ and edge e in X such that v misses e. Note that once v has been found to (not) miss e, v will always (not) miss e. For each edge e of X, we will have already have checked to see whether e has missed neighbors of X up through some vertex j; we start our search from j and continue. Since each edge is checked against each vertex once, the total work in finding v and e through the whole algorithm is $\mathrm{O}(nm)$. In the full version of the paper, we address certain lower level details not treated here; these involve deleting vertices from $N(X)$ and assignment of costs when vertices x and y are identified in the algorithm.

The next task which used $\mathrm{O}(m)$ time is computing Y', which is the component of $G-N(e)$ containing v. First, note that every member of $N(X)-N(e)$ will be added to Y'; these can easily be found in $\mathrm{O}(n)$ time. If a vertex x of X is not in $N(e)$ and is adjacent to a vertex moved from $N(X)$ to Y', it is added to Y'; note that we will also add its connected component in $X-N(e)$ to Y', but the work finding this component can all be charged to m_X.

We now must find neighbors of Y'. When a vertex y is moved into Y', we scan its adjacency list, updating appropriate maintained values (if y is moved from X, all neighbors in $N(X)$ have their number of X-neighbors decreased by 1), and moving neighbors of y from X to $N(X)$. When neighbors of y are moved to $N(X)$, we also update the appropriate values. A vertex can be moved into Y at most n-1 times, since a vertex will never return to X or $N(X)$ until a collapse operation is performed. Similarly a vertex can be moved into $N(X)$ at most n-1 times, since it will not return to X before a collapse operation is performed. Thus, the total time spent scanning adjacency lists while moving vertices to Y and $N(X)$ is $\mathrm{O}(nm)$.

The final step for which we allowed $\mathrm{O}(m)$ time was finding connected components in the new set X, but this step clearly takes $\mathrm{O}(m_X)$ time. Thus the running time of the algorithm is $\mathrm{O}(nm)$ plus the sum of the number of edges in all X sets created during the algorithm.

Each edge is part of at most n edge sets, since each time it is placed in an X set the size of X decreases by at least 1. Thus the total size of all sets of edges induced by X sets is $\mathrm{O}(nm)$, and our algorithm runs in $\mathrm{O}(nm)$ time. $\square$

Note that in this case, the optimization algorithm runs faster than the recognition algorithm. Thus, one can imagine a potential for the algorithm to make mistakes if the input is not weakly chordal; there is no time to run a recognition algorithm to check whether the input is in the class. Fortunately, the correctness of the optimization algorithm depends only on the fact that each edge induces a co-pair of G, which can be checked within the time bounds given. Thus, the algorithm works correctly on all weakly chordal graphs, and will also correctly solve the problems in some cases when the input is not weakly chordal.

6 Conclusions and Open Problems

We have demonstrated the use of the notion of a handle in a graph, a special type of component/separator, to design efficient algorithms for certain classes of graphs without holes. We conclude the paper by proposing some problems for further work. Designing an $\mathrm{O}(n^2)$ algorithm to find a two-pair in an arbitrary graph would improve almost every algorithm known for weakly chordal graphs. We also propose designing an $\mathrm{O}(n^2)$ algorithm to find a two-pair in a given weakly chordal graph as a problem; this would suffice to improve the recognition and optimization algorithms. In view of the proof of Theorem 2.1, a faster algorithm to find a handle in an arbitrary (weakly chordal) graph would in turn imply a faster algorithm to find a two-pair in an arbitrary (weakly chordal) graph. Finally, we propose designing an $\mathrm{O}(n^4)$ algorithm to find a hole in an arbitrary graph as a problem.

References

[1] S. Arikati and C. Rangan, An Efficient Algorithm for Finding a Two-Pair, and its Applications, *Discrete Applied Mathematics*, **31** (1991), 71-74.

[2] R. B. Hayward, Weakly Triangulated Graphs, *Journal of Combinatorial Theory Series B*, **39** (1985), 200-209.

[3] R. B. Hayward, Meyniel Weakly Triangulated Graphs. I. Co-Perfect Orderability, *Discrete Applied Mathematics*, **73** (1997), 199-210.

[4] R. B. Hayward, Meyniel Weakly Triangulated Graphs. II. A Theorem of Dirac, *Discrete Applied Mathematics*, **78** (1997), 283-289.

[5] R.B. Hayward, C.T. Hoàng, and F. Maffray, Optimizing weakly triangulated graphs, *Graphs and Combinatorics*, **5** (1989), 339-349; erratum in **6** (1990), 33-35.

[6] D.J. Rose, R.E. Tarjan, and G.S. Leuker, Algorithmic Aspects of Vertex Elimination on Graphs, *SIAM Journal on Computing*, **5** (1976), 266-283.

[7] J. Spinrad, Finding Large Holes, *Information Processing Letters*, **39** (1991), 227-229.

[8] J. Spinrad and R. Sritharan, Algorithms for Weakly Triangulated Graphs, *Discrete Applied Mathematics*, **59** (1995), 181-191.