Improved Algorithms for Weakly Chordal Graphs

RYAN B. HAYWARD

University of Alberta

JEREMY P. SPINRAD

Vanderbilt University

AND

R. SRITHARAN

The University of Dayton

Abstract. We use a new structural theorem on the presence of two-pairs in weakly chordal graphs to develop improved algorithms. For the recognition problem, we reduce the time complexity from $O(mn^2)$ to $O(m^2)$ and the space complexity from $O(n^3)$ to $O(m+n)$, and also produce a hole or antihole if the input graph is not weakly chordal. For the optimization problems, the complexity of the clique and coloring problems is reduced from $O(mn^2)$ to $O(n^3)$ and the complexity of the independent set and clique cover problems is improved from $O(n^4)$ to $O(mn)$. The space complexity of our optimization algorithms is $O(m + n)$.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms

Additional Key Words and Phrases: Perfect graphs, weakly chordal, recognition, coloring, graph algorithms

ACM Reference Format:

Hayward, R. B., Spinrad, J. P., and Sritharan, R. 2007. Improved algorithms for weakly chordal graphs. ACM Trans. Algor. 3, 2, Article 14 (May 2007), 19 pages. DOI = 10.1145/1240233.1240237 http://doi.acm.org/10.1145/1240233.1240237

A preliminary version of this article appeared in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000. The first author was supported in part by NSERC, the second author by the NSF, and the third author by the Research Council, UD.

Authors' addresses: R. B. Hayward, Department of Computing Science, University of Alberta, Edmonton Canada T6G 2H1, e-mail: hayward@cs.ualberta.ca; J. P. Spinrad, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville TN 37235, e-mail: spin@vuse.vanderbilt.edu; R. Sritharan (contact author), Computer Science Department, University of Dayton, 300 College Park, Dayton OH 45469-2160, e-mail: srithara@notes.udayton.edu.

1. *Introduction and Motivation*

A *hole* is an induced cycle with five or more vertices and an *antihole* is the complement of a hole. A graph is *weakly chordal* (also called weakly triangulated) if it contains no holes and no antiholes. The class of weakly chordal graphs, introduced in Hayward et al. [1985], is a well-studied class of perfect graphs. Hayward et al. [1989] 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 as follows.

THEOREM 1.1 [HAYWARD ET AL. 1989]. *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 [Spinrad and Sritharan 1995] and for solving a variety of optimization problems on weakly chordal graphs [Spinrad and Sritharan 1995; Hayward et al. 1989] (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 in a sparse graph is due to Arikati and Rangan [1991] and runs in $O(mn)$ time. The current best algorithm to find a two-pair in a dense graph is due to Kratsch and Spinrad [2003] and runs in $O(n^{2.83})$ time; their algorithm, similar to the one in Arikati and Rangan [1991], can actually list all the two-pairs in the given graph. In this article 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 [Spinrad and Sritharan 1995] takes $O(n^4)$ time and $O(mn)$ space. We improve on this algorithm by taking $O(m^2)$ time and $O(m+n)$ space, and producing as a certificate a hole or antihole whenever the input graph is not weakly chordal (the previous algorithm does not do so). This is analogous to the well-known $O(m + n)$ time chordal graph recognition algorithm [Rose et al. 1976], which can also produce in this time an induced cycle of length four or more whenever the input graph is not chordal. We note that since a preliminary version of this article [Hayward et al. 2000] appeared Berry et al. [2000] have developed an $O(m^2)$ algorithm to recognize weakly chordal graphs that is quite different in principle; their algorithm uses $O(m^2)$ space. Also, very recently, Nikolopoulos and Palios presented [2004] an algorithm that can find a hole in a graph in $O(m^2)$ time using $O(mn)$ space. In summary, currently, the recognition algorithm presented in this article is the only one requiring $O(m^2)$ time and only a linear amount of space.

The previously best optimization algorithms for weakly chordal graphs [Hayward et al. 1989; Spinrad and Sritharan 1995], based on the algorithms of Hayward et al. [1989] and an $O(mn)$ time algorithm for finding a two-pair [Arikati and Rangan 1991], repeat the following process: Find a two-pair and modify the graph, either by adding an edge between vertices of a two-pair or by "collapsing a two-pair" (defined later) to form a single vertex. In the unweighted case, the clique and coloring algorithms [Hayward et al. 1989] take $O(mn^2)$ time, while the algorithms

for independent set and clique cover problems take $O(n^4)$ time; the latter problems are essentially solved by running the former algorithm on the complement of the graph. The weighted optimization algorithms [Spinrad and Sritharan 1995], which are based on the algorithms in Hayward et al. [1989], run in $O(n^4)$ time. We present algorithms that can solve the unweighted independent set and clique cover problems, which only involve collapsing two-pairs of the complement, in $O(mn)$ time. Running the same algorithm on the complement of the graph, we can solve the clique and coloring problems in $O(n^3)$ time. While our optimization algorithms correctly solve the problems on any given weakly chordal graph, they are actually "robust" [Raghavan and Spinrad 2003] in the sense that they either correctly solve the problem on any given input (which need not be weakly chordal) or correctly declare that the input is not a weakly chordal graph. We next introduce the notation used in the article.

1.1. NOTATION. P_k and C_k denote, respectively, the induced path and cycle with k vertices. $\overline{C}_k$ is the complement of C_k. When $k \geq 5$, we use *hole of size at least* k to refer to an induced cycle on at least k vertices. 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. We use xy to denote the edge incident on the vertices x and y. 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. For graph G and edge xy of G, $G - xy$ refers to the graph obtained from G by deleting the edge xy, but retaining vertices x and y.

2. *An Overview and a New Tool*

A *co-two-pair* of a graph, or simply a *copair*, is a two-pair of the complement of the graph. The previous use of a two-pair (or a copair) in recognition [Spinrad and Sritharan 1995] and optimization [Hayward et al. 1989] of weakly chordal graphs essentially relies on the facts given next.

LEMMA 2.1 [HAYWARD ET AL. 1989]. *Suppose G is a weakly chordal graph and $\{x, y\}$ is a two-pair of G. Let G^* be the graph obtained from G by deleting vertices x and y and introducing vertex z whose neighborhood is the union of the neighborhoods of x and y in G. Then, G^* is weakly chordal. Further, the chromatic number and clique number of G equal the corresponding numbers for G^*.*

We refer to the operation used in Lemma 2.1 as "collapsing a two-pair".

LEMMA 2.2 [SPINRAD AND SRITHARAN 1995]. *Suppose $\{x, y\}$ is a two-pair of graph G. Let G^+ be the graph obtained from G by adding the edge xy. Then, G is weakly chordal if and only if G^+ is weakly chordal.*

As a graph is weakly chordal if and only if its complement is weakly chordal, the following is immediate.

COROLLARY 2.3 [SPINRAD AND SRITHARAN 1995]. *Suppose $\{x, y\}$ is a copair of graph G. Let G^- be the graph obtained from G by deleting the edge xy (leaving vertices x and y to remain). Then, G is weakly chordal if and only if G^- is weakly chordal.*

Lemma 2.1 suggests the following algorithm [Hayward et al. 1989] for optimizing weakly chordal graphs: Find a two-pair, collapse it, and solve the problem recursively. Similarly, Corollary 2.3 suggests the following algorithm [Spinrad and Sritharan 1995] for recognizing weakly chordal graphs: Repeatedly find a copair and delete the edge involved until there are no edges left. Thus, our algorithms find a copair of the given graph, modify the graph by either deleting the edge that corresponds to the copair or by replacing the two vertices of the copair by a new vertex with an appropriate neighborhood, and then iterate this process on the remaining graph. In order to facilitate the repeated search for copairs, we make use of a structure called "handle" (defined later) from Hayward [1997a, 1997b]. In essence, a handle H of a graph G is a connected set of at least two vertices with some special properties. It is known Hayward [1997a, 1997b] that any graph containing the complement of a P_3 has a handle. An important fact we will prove later is that if H is a handle of a weakly chordal graph G, then any copair of $G[H]$ is also a copair of G. Therefore, we find a copair of the given graph G with $H_0 = V(G)$ by first finding a handle H_1 of G, and then repeatedly finding a handle H_i of $G[H_{i-1}]$, should it exist. It will turn out that each H_i is a proper subset of H_{i-1}. Also, any edge of the final handle H_k is guaranteed to be a copair of G when G is weakly chordal. We then modify the graph around the copair, as required by the recognition or optimization algorithms, and then proceed to find a copair of the remaining graph.

There are several reasons as to why the use of a sequence of handles aids in making our algorithms efficient. Essentially, after a local change is made to the graph around the found copair, a sequence of handles to use for the location of a next copair is almost fully available inside the sequence of handles already computed. Hence, by restarting the algorithm that finds a copair at an appropriate state, we are able to continue to find the next copair without starting over. In order to support this restart, we maintain a stack of sets of vertices. When a handle H_i of $G[H_{i-1}]$ is computed by "refining" H_{i-1} into H_i, we stack the set $H_{i-1} \setminus H_i$ of vertices. We note that the computation of a handle H_i of $G[H_{i-1}]$ itself is done by starting with an appropriate candidate set X of vertices and successively refining it until we eventually finish with the handle H_i. During each such refinement, we stack the set of vertices that is removed from the candidate set X. On a restart, we pop the stacked sets of vertices appropriately and restore the state of the algorithm so that the search for the next copair can continue.

Next, we present the definition of a handle and some related necessary facts. A *handle* in a graph G [Hayward 1997a, 1997b] 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. Note that $N(H)$ is a minimal separator of H and J.

When the vertex subset H of G with $|H| \geq 2$ induces a connected component of G, as $N(H) = \emptyset$, H is trivially a handle of G; any other connected component of G can be considered a cohandle of H. In this case, it is easily seen that when G is a weakly chordal graph, any copair of $G[H]$ is a copair of G also. We shall prove later that this holds for any handle H of G when G is a weakly chordal graph.

THEOREM 2.4 [HAYWARD 1997A, 1997B]. *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 polynomial time.*

We first present an overview of our strategy by describing the connection between the ideas of a copair, a handle, and our algorithms. Then, in the next section, we discuss the algorithm to find a copair. We then describe our recognition and optimization algorithms. As noted earlier, the latter algorithms need to perform repeated searches for copairs. In order to perform these searches efficiently, a set X of vertices of the given graph G is maintained ensuring that a copair of G can always be found within X (i.e., in $G[X]$) whenever G is weakly chordal. The set X is refined appropriately and repeatedly until each edge in $G[X]$ induces a copair of G. The graph is then modified as required by the top-level (recognition or optimization) algorithm, and the search for a copair resumes in the modified graph. The advantage of our method is that when the search for a copair resumes in the modified graph, vertices are considered in essentially the same order as in the original search. This enables us to avoid unnecessary recomputation by maintaining a stack of vertices eliminated from X during each refinement operation. It turns out that upon resumption of the search, the set X can be appropriately restored using the information stored on the stack.

We condense the relevant algorithms for finding a handle from Hayward [1997a, 1997b] into the following algorithm whose details are needed for the analysis of our algorithms. The statements tagged "r1:" and "r2:" push a set of vertices onto the stack so that *find-handle* can be used by other algorithms. These statements are not critical to understanding the working of *find-handle* and we comment on the use of the stack later.

Algorithm *find-handle*

Input: graph G
Output: either the message 'no handle'
or a handle H of G and a cohandle J of H

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

r1: push the set $V(G) - X$ of vertices onto the stack

Restart-point:
/* *handle finding* */
while some v in $N(X)$ misses some e in X **do**
$Y' \leftarrow$ component of $G - N(e)$ containing v
$X' \leftarrow$ component of $G - N(Y')$ containing e /*(+)*/

r2: push the set $X - X'$ of vertices onto the stack

/* $N(X') = N(Y')$ minimally separates X' from Y' */

```
        X ← X′;  Y ← Y′
    endwhile
    H ← X;  J ← Y
end find-handle
```

We also use an operation called *restarting*, namely, reentering *find-handle* in the handle-finding phase after 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.

2.1. IMPLEMENTATION OF ALGORITHM *find-handle*. We now show that the algorithm *find-handle* can be implemented to run in $O(mn)$ time. We first show that the initialization phase can be implemented to run in $O(m)$ 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 $V(G) - \{x\} - N(x)$, and check whether $N(y) = N(x)$ for every y in C. If so, we remove C and repeat the process. If not, then 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.

It can be shown [Hayward 1997a, 1997b] that $Y \subseteq Y'$, $X' \subseteq X$, and $(Y \cup N(X) - N(e)) \subseteq Y'$ throughout the execution of the algorithm *find-handle*. A key fact [Hayward 1997a, 1997b] 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 during each iteration of the while loop. It is clear that, barring the operation of finding vertex v in $N(X)$ and edge e in X such that v misses e, an iteration of the handle-finding phase can easily be done in $O(m)$ time. Next, we introduce data structures which are used to ensure that the total cost of finding such pairs v, e over the entire run of the algorithm is $O(mn)$. We note that these data structures are also used by other algorithms that invoke *find-handle*.

We keep an ordered list NX of vertices. Vertices that at any time become part of $N(X)$ are added to this list, with new vertices that become part of $N(X)$ being added to the end of this list. For each vertex u in NX, we maintain $NN(u)$, the number of neighbors that u has in the current set X. Therefore, a vertex u belongs to $N(X)$ with respect to the current set X if and only if $NN(u) > 0$.

The list NX is used to efficiently find vertex v of $N(X)$ and edge e of X such that v misses e. For each edge e, we will want to scan NX once during the entire course of the algorithm so that the total cost of finding such pairs v and e is $O(mn)$. In order to facilitate this, for each edge e in X, we keep a pointer in NX as to how far we have already scanned NX with respect to e. The idea is that a particular vertex v and an edge e are used to refine the set X (at most) once; since the refining of set X leaves edge e in X and vertex v outside of $X \cup N(X)$. Also, for an edge e, once a vertex v is found to miss (not miss) e, v will always miss (not miss) e. Therefore, if a future refinement were to occur with respect to e, we only have to scan further down the list NX to find an appropriate vertex that misses e. The list NX and the $NN(u)$ values can easily be updated in $O(m)$ time at every instance where the set X changes its value. Thus, over the entire course of the algorithm, for every edge e, the list NX is scanned at most once and we have the following theorem.

THEOREM 2.5. *Algorithm find-handle can be implemented to run in $O(mn)$ time.*

Our algorithms modify the subgraph induced by a handle by deleting edges from it. We conclude this section by presenting some properties of handles with respect to such changes; the properties are easy to verify.

Property P1. Suppose H is a handle of graph G and x and y are adjacent vertices of H. Supposing that H induces a connected subgraph of $G - xy$, then H is a handle of $G - xy$.

Property P2. Suppose H is a handle of graph G and x and y are adjacent vertices of H. Suppose H induces a disconnected subgraph of $G - xy$ and H' is a component of $(G - xy)[H]$ with at least two vertices. Then, H' is a handle of $G - xy$.

3. *Finding Copairs*

Given a graph, our approach to finding a copair is to first find a handle of the graph, and then a handle of the subgraph induced by the handle found, and so on, until we cannot repeat this process. It turns out that if a graph is weakly chordal then every edge of the last handle found induces a copair 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 copair as required by the relevant recognition or optimization algorithm, we search for the next copair by restarting *find-handle*, thus doing less work than in the original call to *find-handle*.

Algorithm *find-copair*

Input: graph G with at least one edge
Output: a copair of G, when G is weakly chordal

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 $\{x, y\}$ such that xy is an edge of $G[H_j]$
end *find-copair*

We need the following lemma in order to prove the correctness of algorithm *find-copair*.

LEMMA 3.1. *Suppose H is a handle of a weakly chordal graph G and $\{x, y\}$ is a copair of $G[H]$. Then, $\{x, y\}$ is a copair of G.*

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

Then, there exists an induced path $P = x \cdots y$ with at least four vertices in $\overline{G}$. As each vertex in R sees both x and y in $\overline{G}$, P does not involve any vertex in R; therefore, P has at least a vertex from I. Now, P cannot have a segment uvw such

that u and w are in H but v is in I, for otherwise, vertex v of I misses edge uw of $G[H]$ in G, contradicting that H is a handle of G. Thus, at least two consecutive vertices of P are in I and P involves at least an edge of $\overline{G}$ with both endpoints in I.

In $\overline{G}$, consider a segment $P' = x_2x_3x_4 \cdots x_r$ 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 G has no holes, in G every two nonadjacent vertices of I must have a common neighbor in J. In particular, x_3 and x_4 see some vertex 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 $x_1x_2x_3x_4$ is a P_4. Let x_k be the first vertex in P' after x_4 such that x_1 sees x_k in $\overline{G}$; such an x_k exists, as x_1 sees x_r in $\overline{G}$. Then, $\{x_1, x_2, \ldots, x_k\}$ induces a hole of size at least five in $\overline{G}$, contradicting G being weakly chordal. □

THEOREM 3.2. *Algorithm find-copair is correct.*

PROOF. Assume that *find-copair* terminates after performing p iterations of the loop. Then, $G[H_p]$ has no $\overline{P}_3$, as any graph with a $\overline{P}_3$ has a handle. Thus $G[H_p]$ is a complete multipartite graph, and therefore every edge of $G[H_p]$ induces a copair of $G[H_p]$. Then, by Lemma 3.1, every edge of $G[H_p]$ induces a copair 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 copair of G. □

THEOREM 3.3. *Algorithm find-copair runs in O(mn) time.*

PROOF. The time spent by an execution of *find-copair* is dominated by the total time spent by the resulting calls to *find-handle*. Since *find-handle* is called at most n times, the total time spent in its initialization phase over these calls is $O(mn)$. As discussed in Section 2.1, the total cost of finding vertex v in $N(X)$ and edge e in X such that v misses e is $O(mn)$. Finally, the total time spent in the rest of its handle-finding phase is also $O(mn)$, as each iteration of the loop takes $O(m)$ time and in each iteration at least one vertex is deleted from the set X. □

Next we describe a basic strategy employed by our algorithms. Recall that at any point during its execution, algorithm *find-copair* maintains a sequence of sets $V(G) = H_0, H_1, H_2, \ldots, H_k$ such that each H_i is a handle of the subgraph induced by H_{i-1}. We refer to the last set in the sequence as the "current handle" at that point in execution. When $G[H_k]$ does not have a handle, *find-copair* outputs a pair of adjacent vertices from H_k. Consider the time at which the very first copair $\{x, y\}$ is output. Referring back to algorithms *find-handle* and *find-copair*, $X = H_k$ induces a complete multipartite graph in G and each H_i is a handle of $G[H_{i-1}]$. Our algorithms then modify the graph G essentially by deleting one or more edges from the graph. We would now like to find a copair of the modified graph G, without having to repeat the entire process of finding a new sequence of handles. Deletion of edges may disconnect $G[H_i]$; however, by Property P2, any connected subset of H_i with at least two vertices is still a handle of $G[H_{i-1}]$. It then turns out that we have a new sequence of handles existing within the old sequence and we would like to extend this new sequence when possible. Note that we can consider any connected subset H' of H_{k-1} with at least two vertices as the current handle of G. We compute a subset of H' as the value for the set X so that we can reenter the algorithm *find-handle* at the entry point Restart-point and continue with the computation of a handle of the current handle. Referring back to the algorithm *find-handle*, what we thus need to ensure is that there is a connected subset Y of H' minimally separated from X in

H' by $N(X) = N(Y)$. Therefore, when we restart the computation to find the next copair, we always make sure that the set computed as the value of X satisfies this condition. We note that our algorithms do not explicitly compute the set H'. Instead, the computation of set X implies the existence of an appropriate sequence of handles such that X is a subset of the current handle. In particular, for each H_i, if we take the vertex set of the connected component of $G[H_i]$ that includes X, then the sequence of such sets will be a required sequence of handles. Our algorithms then basically grow and shrink such a sequence of sets. Also, our algorithms process edges in $G[H_i]$ before moving on to process the remaining edges in $G[H_{i-1}]$. Hence, a stack plays a vital role in controlling this process. We now present the invariant maintained by our algorithms that guarantees their correctness. Given the preceding discussion, for the sake of simplicity, when we say "in graph G, H_i is a handle of $G[H_j]$", we actually mean that any connected subset of H_i with at least two vertices is a handle of $G[H_j]$.

Invariant $\mathcal{I}$. For any set X computed by the algorithm *find-handle*, the following are true: $X \subseteq H$, where H is a current handle of graph G when X is computed, $|X| \geq 2$, X induces a connected subgraph of $G[H]$, and there exists a component Y of $G[H] - N(X)$ such that $N(X) \cap H = N(Y) \cap H$. In other words, X and Y are minimally separated by $N(X) = N(Y)$ in $G[H]$.

We note that when X (with $|X| \geq 2$) induces a connected component of $G[H]$, it trivially satisfies the invariant. The following lemma shows the relevance of the invariant to algorithms that use *find-handle* via *find-copair*.

LEMMA 3.4. *Suppose an algorithm uses find-copair on a weakly chordal graph G, ensuring that the invariant $\mathcal{I}$ is never violated. Then, any pair $\{x, y\}$ output by find-copair is a copair of G.*

PROOF. Referring back to the working of algorithm *find-handle*, the fact that set X in current handle H of graph G satisfies the invariant implies that a handle of $G[H]$, when it exists, will be found inside X. This in turn implies that *find-copair* computes a sequence of handles for G. Let $V(G) = H_0, H_1, \ldots, H_k$ be the sequence of handles already computed when *find-copair* outputs $\{x, y\}$. Then, at that instant (referring back to *find-handle*), $X = H_k$, H_k is a handle of $G[H_{k-1}]$, H_k induces a complete multipartite graph in $G[H_{k-1}]$, and $\{x, y\}$ is a copair of subgraph induced by H_k in $G[H_{k-1}]$. Since X maintains the invariant $\mathcal{I}$, we can use Lemma 3.1 with $G[H_{k-1}]$ in place of G and H_k in place of H to conclude that $\{x, y\}$ is a copair of $G[H_{k-1}]$. It then follows from Theorem 3.2 that $\{x, y\}$ is a copair of G also. □

Referring to the algorithm *find-handle*, we call the operation of obtaining a proper subset of X as the new value of X as "refining the set X". It is obvious that refinement of X is done around the statement tagged "r2:". Note that X is refined around the statement tagged "r1:" also. This happens when a handle $X = H_i$ in the sequence of handles for G is found, and *find-copair* proceeds to compute a handle H_{i+1} of $G[H_i]$ with the invocation *find-handle*$(G[H_i])$. Also, during the invocation *find-handle*$(G[H_0 = V(G)])$, when set X is computed for the first time, $V(G) - X$ is pushed onto the stack. Therefore, at any time that a set X is refined to the new value Z, algorithm *find-handle* pushes the set $X - Z$ of vertices onto the stack.

4. *Recognizing Weakly Chordal Graphs*

Since our algorithm uses the basic structure of the $O(n^4)$ algorithm [Spinrad and Sritharan 1995] for recognizing weakly chordal graphs, we review this first. Given an arbitrary graph G as input, the algorithm in Spinrad and Sritharan [1995] repeatedly finds a two-pair and adds an edge between its vertices. 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 *copair edge* is the edge induced by a copair. Our algorithm can be thought of as the execution of the aforementioned algorithm in the complement: repeatedly find a copair and delete the induced edge (but not the vertices). Since a graph is weakly chordal if and only if 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-copair* as the fundamental subroutine. After a copair is found and its edge deleted, the next copair is found by restarting *find-copair* if the current set X still contains an edge, and by popping back to an appropriate previous level to compute a new value for the set X otherwise. In this section we show that our recognition algorithm is correct and takes $O(m^2)$ time.

Algorithm *wc-recognition*

Input: graph G with at least one edge
Output: 'yes' if G is weakly chordal, 'no' otherwise
Shared variables: the following variables of *find-handle*:
the stack,
the set X,
the ordered list NX, and
$NN(u)$ for each vertex u in NX

```
begin
    {x, y} ← find-copair(G)   /* the first one */
    repeat
        if {x, y} is not a copair of G then
            return 'no'
        endif
        G ← G − (edge xy)   /* vertices x, y remain */
        if G has at least an edge then
            restart find-copair after adjusting value of X in find-handle
            to compute the next pair {x, y} as explained below
        endif
    until (G has no edges)
    return 'yes'
end wc-recognition
```

We now explain how the restarting of *find-copair* is done. This operation, as it is, will be used by our optimization algorithms also. Let $V(G) = H_0, H_1, \ldots, H_k$ be the sequence of handles computed just before the edge xy was deleted from G. After the edge xy is deleted, $X = H_k$ may no longer be a handle of $G[H_{k-1}]$. We compute a subset of H_{k-1} as value for X such that in $G[H_{k-1}]$, some connected subset Y is minimally separated from X by $N(X)$. This enables *find-copair* to

resume computing a handle of $G[H_{k-1}]$ by reentering *find-handle* at the entry point Restart-point. Next we provide the details of this operation.

Algorithm for restarting *find-copair*

```
begin
    X_old ← X
    if X induces an independent set then
        repeat
            X_old ← X_old ∪ (set popped from stack)
        until (stack is empty) or (X_old is not an independent set)
    endif
    if X_old does not induce an independent set then
        X_1 ← a component of G[X_old] with |X_1| ≥ 2
        X ← X_1
        push the set X_old − X of vertices onto stack
        Resume execution of find-copair by reentering find-handle at the
                                                    entry point Restart-point
    else  /* graph has no edges left */
        return
    endif
end
```

Note that when X_{old} is not an independent set, some connected component of $G[X_{old}]$ with at least an edge, possibly X itself, becomes the new value for X. Also, supposing that the stack becomes empty and set X_{old} is not an independent set, then all edges in the sequence of handles have been processed and $G[X_{old}]$ is the graph remaining at this instant. We must then process this graph by computing a new sequence of handles. This will be accomplished as some connected component of $G[X_{old}]$ with at least two vertices is computed as the value for X. This set will subsequently be found to be a handle of the graph at that instant in algorithm *find-handle*. Finally, suppose the stack becomes empty, but the set X_{old} induces an independent set. Then, it means that every edge in the original graph has been processed already.

It is not difficult to test in $O(m+n)$ time whether an edge induces a copair, 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 copair 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 nonneighbors in $O(n)$ time, and to determine in $O(m+n)$ 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 copair takes $O(n+m)$ time, and since each edge is deleted immediately after being tested, the total time spent on this task over the whole algorithm is $O(m^2)$.

THEOREM 4.1. *Algorithm wc-recognition is correct.*

PROOF. In view of Theorem 1.1 and Corollary 2.3, we only have to argue that when *wc-recognition* is invoked on a weakly chordal graph, every pair $\{x, y\}$ output

by *find-copair* is a copair of the graph at that instant. Again, given Lemma 3.4, we can accomplish this by showing that any set X computed by *find-handle* during the execution of *wc-recognition* is consistent with the invariant $\mathcal{I}$. The correctness of algorithm *find-copair* guarantees that up to the point at which the very first pair is output, the invariant $\mathcal{I}$ is maintained.

Suppose at a particular instant G is a weakly chordal graph and $V(G) = H_0, H_1, \ldots, H_k$ is the sequence of handles already computed when pair $\{x, y\}$ is output by *find-copair*. Recall that we compute $G' = G - xy$ and restart *find-copair*. Given that the invariant $\mathcal{I}$ is maintained thus far, we need to show that the invariant is maintained when a new value for X is computed from $X = H_k$ after the edge xy is deleted. First, (by Property P2) for $1 \leq i \leq k$, any connected component of $G'[H_i]$ that has at least two vertices is still a handle of $G'[H_{i-1}]$. Therefore, we only need to concern ourselves with H_k, the current handle for G. We consider the various ways in which the algorithm could have progressed. Note that whenever X equals the current handle of the present graph, the invariant $\mathcal{I}$ is maintained; simply take the cohandle of the current handle (with respect to the subgraph induced by the previous handle) as the set Y.

Suppose $G'[X]$ is connected. Then, by Property P1, X is a handle of $G'[H_{k-1}]$, and therefore X is the current handle of G' and $\mathcal{I}$ is maintained.

Suppose $G'[X]$ is disconnected and X_1 is a component of $G'[X]$ with at least two vertices. Then, the algorithm proceeds with $X = X_1$. By Property P2, X_1 is a handle of $G'[H_{k-1}]$, and therefore X is the current handle of G' and $\mathcal{I}$ is maintained.

Suppose X induces an independent set. Then, the algorithm repeatedly pops the stack, combining the sets popped with X to eventually arrive at set X_{old}. Let X_1 be a connected component of $G'[X_{old}]$ with at least two vertices. The algorithm proceeds with $X = X_1$. Next we want to show that $X = X_1$ is consistent with the invariant $\mathcal{I}$. Let G_{old} be the graph when X_{old} was first computed as the value of X. Let Y_{old} be the connected set minimally separated from X_{old} by $N(X_{old})$ in the handle current at that instant.

If X_{old} was ever found to be a current handle of G_{old}, then by property P2, X_1 is a current handle for G' and therefore $\mathcal{I}$ is maintained.

Therefore, we can assume that X_{old} did not become a current handle for G_{old} and X_{old} was refined leaving set R on the stack. Let H_r be the handle that was current when X_{old} was first computed as the value for X. Clearly, X_{old}, Y_{old}, and $N(X_{old})$ are vertex subsets of H_r. Note that Y_{old} induces a connected subgraph of $G'[H_r]$ and every vertex in $N(X_{old})$ sees some vertex in Y_{old} in $G'[H_r]$. Therefore, any connected component of $G'[H_r]$ that includes a vertex of $N(X_{old})$ must include all the vertices in Y_{old}. Observe that any path in $G'[H_r]$ from a vertex in X_1 to a vertex not in X_1 must pass through some vertex in $N(X_{old})$. Let H' be the connected component of $G'[H_r]$ that includes X_1. Then, by Property P2, the vertex set of H' is a handle of $G'[H_{r-1}]$ and a current handle of G'. In H', let S be the set of those vertices of $N(X_{old})$ each of which sees some vertex in X_1. If $S = \emptyset$, then X_1 is the vertex set of H' and therefore is a current handle of G'. Then, $X = X_1$ satisfies the invariant. Now suppose S is a nonempty set. Then, S separates X_1 from Y_{old} in H'. Also, as $S \subseteq N(X_{old})$, and $N(X_{old})$ was a minimal separator for X_{old} and Y_{old} when X_{old} was first refined, every vertex of S sees some vertex of Y_{old}. Let Y' be the component of $H' - S$ that contains all the vertices of the connected set Y_{old}. Then, in H', X_1 is minimally separated from Y' by S satisfying the conditions of the invariant $\mathcal{I}$. □

THEOREM 4.2. *Algorithm wc-recognition determines whether G is weakly chordal in $O(m^2)$ time.*

PROOF. Recall that the algorithm maintains a sequence of sets $V(G) = H_0$, $H_1, \ldots, H_i$, and that at any point in time, we are dealing with $G[H = H_i]$; we call H the current context. Note that it is easy to maintain the current context so that adjacencies are computed with respect to the current context. When a new handle H_j is computed, the context changes to H_j. Also, when a handle $X = H_j$ is refined in the initialization step of *find-handle*, we can mark that so that during the restart, while popping the stack, if X_{old} ever equals some H_i, then context can be changed to H_{i-1}, since X_{old} was computed to be X in the context H_{i-1}. We use the data structures NX and $NN(u)$ as maintained by the algorithm *find-handle*. Therefore, a vertex u of H belongs to $N(X)$ with respect to the current set X if and only if $NN(u) > 0$.

As noted earlier, whether a pair $\{x, y\}$ is a copair can be tested in O(m) time. As such a test is done O(m) times, the total cost of these operations is O(m^2).

We next consider the operation of finding a vertex v from $N(X)$ and an edge e in X such that v misses e (in order to refine the set X). With respect to a particular vertex v and an edge e, refinement is done at most once; for the refinement leaves edge e in X and vertex v outside of $X \cup N(X)$. In the future if v were to return to $N(X)$, edge e would already have been deleted from the graph. Also, for an edge e, once a vertex v is found to miss (not miss) e, v will always miss (not miss) e. Therefore, when a future refinement occurs with respect to e, we only have to scan further down the list NX to find an appropriate vertex that misses e. Thus, for each edge e of X, we will already have checked to see whether e has missed neighbors of X up through some vertex w on NX; we start our search from w and continue. Since each edge is checked against each vertex once, the total work in finding v and e through the whole algorithm is O(mn).

A straightforward implementation of the rest of the refinement operation can be done in O(m) time. 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 are at most m refinement steps. Thus, total time spent on all the refinements is O(m^2).

We now consider the initialization step of *find-handle*. Barring the very first invocation (on the entire input graph), any invocation of *find-handle* is on some set $X = H$ that is found to be a current handle. Therefore, the number of invocations of *find-handle* is bounded by the number of refinement steps and is O(m). As the time spent in the initialization phase for each such invocation is O(m), the total time spent in the initialization steps is O(m^2). Also, when a new value for set X is computed in the initialization step, we add $N(X)$ computed to the list NX, updating the $NN(u)$ value for each vertex u of $N(X)$ and we set the context to H.

Finally, we show that a restart after the deletion of a copair edge can be done in O(m) time. As vertices in X are marked, we can scan the adjacencies of vertices in $X_{old} = X$ to check whether X induces an independent set. If it does, then we repeatedly pop a set of vertices from the stack and combine it with X_{old}, until X_{old} does not induce an independent set. When we pop a set of vertices from the stack, we update the context as required. Also, once we pop a set of vertices and mark its members to belong to X_{old}, we scan the adjacencies of each

of the vertices thus added to check whether it has a neighbor in X_{old}. Thus, arriving at a set X_{old} that does not induce an independent set can be done in O(m) time. Clearly, computing a connected subset of X_{old}, with at least two vertices, as the new value of X can be done in O(m) time. Then, we simply allow ourselves O(m) time to update the $NN(u)$ value for each neighbor u of X. As a restart is done only after deletion of a copair edge, the overall time thus spent on restarts is O(m^2). □

5. *Finding a Proof that a Graph is Not Weakly Chordal*

The recognition algorithm of the previous section, like that of Spinrad and Sritharan [1995] on which it is based, does not provide the most natural evidence that a graph is not weakly chordal, namely, a hole in either the graph, or its complement when it returns 'no'. In this section we show how the algorithm *wc-recognition* can be modified to find a hole or antihole in a graph that is not weakly chordal such that the running time of the algorithm is dominated by that of the recognition algorithm. In contrast, if the O(n^4) recognition algorithm of Spinrad and Sritharan [1995] is used, we get no information about where a hole or antihole might be when the input graph is not weakly chordal.

Note that by Corollary 2.3, deletion of the edge corresponding to a copair from a graph neither creates nor destroys any holes or antiholes. Our basic idea is that the very first pair of vertices encountered by algorithm *wc-recognition* which is not a copair of the graph at that point can be used as a starting point to find a hole or antihole of the input graph. We need the following lemma whose proof uses ideas similar to those used in the proof of Lemma 3.1.

LEMMA 5.1. *Suppose H is a handle of graph G and $\{x, y\}$ is a copair of $G[H]$ but not a copair of G. Then, a hole or an antihole of G can be found in O(n^2) time.*

PROOF. Let J be a cohandle of H, $I = N(H) = N(J)$, and $R = V(G) - H - I$.

Since $\{x, y\}$ is not a copair of G, In $\overline{G}$ there is an induced path with at least four vertices connecting 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 copair of $G[H]$, P must involve vertices from I.

Now, P cannot have a segment uvw such that u and w are in H, but v is in I. Otherwise, in G, vertex v of I will miss the edge uw of $G[H]$, contradicting the assumption that H is a handle of G. Therefore, in $\overline{G}$, the 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 that x_3 and x_4 do not have a common neighbor in J. Compute P^*, namely, 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 three 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. Then, union of P^* and P^{**} is 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_k be the

first vertex that comes after x_4 in P' such that x_1 sees x_k; there must be such a vertex, as x_1 sees x_r in $\overline{G}$. Then, $\{x_1, x_2, \ldots, x_k\}$ induces a hole of size at least five in $\overline{G}$. Every step of the previous algorithm takes $O(n^2)$ time and there is a constant number of steps in the algorithm, making the time complexity $O(n^2)$. □

We next show how to find an appropriate graph G and its handle H satisfying conditions of Lemma 5.1 when algorithm *wc-recognition* returns 'no'.

THEOREM 5.2. *Suppose algorithm wc-recognition returns 'no' on an input graph and $\{x, y\}$ is the pair returned by the algorithm find-copair at that point. Then, graph G and its handle H satisfying conditions of Lemma 5.1 can be found in O(mn) time.*

PROOF. Refer to algorithm *wc-recognition*. Let G' be the graph present at the instant the algorithm returns 'no'. Referring back to the algorithm *find-copair*, let $V(G') = H_0, H_1, H_2, \ldots, H_p$ be the sequence of handles computed up to this point. As xy is an edge of $G'[H_p]$ and $G'[H_p]$ is a complete multipartite graph, $\{x, y\}$ is a copair of $G'[H_p]$. Also, as the algorithm *wc-recognition* returns 'no', $\{x, y\}$ is not a copair of G'. Find the largest k such that $\{x, y\}$ is a copair of $G'[H_k]$ but not a copair of $G'[H_{k-1}]$. Note that H_k is a handle of $G'[H_{k-1}]$. We then take $G = G'[H_{k-1}]$ and $H = H_k$. It is easy to maintain the sets H_i so that such H_{k-1} and H_k can be found in $O(mn)$ time. □

6. *Optimization Problems*

In this section, we demonstrate that algorithm *find-copair* can be used 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 Hayward et al. [1989] and is based on the following operation: Repeatedly find a two-pair $\{x, y\}$, and collapse the two-pair 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 Arikati and Rangan [1991] this takes $O(mn^2)$ time. We show that we can solve the independent set and clique cover problems on G in $O(mn)$ time essentially by simulating the aforementioned algorithm for clique and coloring problems on the complement of G.

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 add a new vertex z such that $N(z) = N(x) \cap N(y)$, and then delete the vertices x and y. However, an equivalent way of viewing the operation is to consider it as a sequence of deletion of edges as follows: Mark x to be z in the new graph, delete every edge incident on y, mark y as deleted, and delete every edge xw such that $w \in N(x) - N(y)$.

Algorithm *wc-opt* can be defined as follows: When *find-copair* returns the pair $\{x, y\}$ we identify x and y into vertex z, as explained before. We also replace x and y with z in the set X maintained by *find-handle*. We then use the operation of restarting *find-copair* as explained in Section 4 and proceed with finding a copair. As computing (in the complement) a largest clique and optimum coloring of the larger graph from those of the smaller graph is explained in Hayward et al. [1989],

we omit these details and concern ourselves only with the implementation of the successive identification operations.

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

THEOREM 6.1. *Algorithm wc-opt is correct.*

PROOF. In view of Lemma 2.1, we need only argue that when *wc-opt* is invoked on a weakly chordal graph, every pair $\{x, y\}$ output by *find-copair* is a copair of the graph at that instant. Again, given Lemma 3.4, we can accomplish this by showing that any set X computed by *find-handle* during the execution of *wc-opt* is consistent with the invariant $\mathcal{I}$. The correctness of algorithm *find-copair* guarantees that up until the point at which the very first pair is output, the invariant $\mathcal{I}$ is maintained.

Suppose at a particular instant that G is a weakly chordal graph and $V(G) = H_0$, $H_1, \ldots, H_k$ is the sequence of handles already computed when pair $\{x, y\}$ is output by *find-copair*. Given that the invariant $\mathcal{I}$ is maintained thus far, we need to show that the invariant is maintained when a new value for X is computed from $X = H_k$ after the copair $\{x, y\}$ is identified and algorithm *find-copair* is restarted. Let G' be the graph obtained from G by identifying the copair $\{x, y\}$ into vertex z.

Any edge that is deleted clearly has at least one endpoint in X. First consider the effect of deleting an edge, one of whose endpoints is not in X. In particular, let us consider the case when one of the endpoints is in $N(X)$ in $G[H_{k-1}]$. It is now possible for a vertex $w \in N(X)$ not to have a neighbor in X any more. Since such a vertex w has a neighbor in Y (the connected set that is minimally separated from X in $G[H_{k-1}]$), we can instead treat w as part of Y. In this case, as the endpoint not in X must belong to a separating set for some set computed as the value of X in the past, essentially the same logic applies to every set computed as X thus far, including those computed to be a current handle at some point. Thus, logically, we can think of $N(X)$ in $G[H_{k-1}]$ as being the set resulting after moving all such vertices to Y, and likewise for all sets computed as X thus far, including the sets H_i in the sequence of handles.

Further, consider H_{i-1} and H_i such that zw is an edge of $G'[H_i]$. Clearly, w sees both x and y in G. In $G'[H_{i-1}]$, for any vertex u in $N(H_i)$, suppose u misses w. Then, as H_i was a handle of $G[H_{i-1}]$, u must see both x and y in G. Thus, u sees z in G'. Therefore, for any vertex u in $N(H_i)$ in $G'[H_{i-1}]$, and edge vw with both endpoints in H_i, u sees at least one of v, w. Finally, as a result of moving vertices, for a set Z computed to be the value of X at some point in some H_j, suppose $N(Z)$ in $G'[H_j]$ becomes empty. Then, Z induces the union of connected components of $G'[H_j]$, and hence any connected subset of Z with at least two vertices is trivially a handle of $G'[H_j]$. Therefore, after deletion of edges with exactly one endpoint in X, (by Property P2), for $1 \leq i \leq k$, any connected component of $G'[H_i]$ that has at least two vertices is still a handle of $G'[H_{i-1}]$. Therefore, we only need to concern ourselves with H_k, the current handle for G, and the effect of deleting edges with both endpoints in X. The effect of deletion of such an edge on maintaining the invariant was dealt with in the proof of Theorem 4.1. Therefore, regardless of whether $G'[X]$ is connected, disconnected but has a connected component with at least two vertices, or induces an independent set, the arguments of Theorem 4.1 can be used to establish that invariant $\mathcal{I}$ is maintained after the restart also. □

We now show that the algorithm *wc-opt* can be implemented to run in O(mn) time. Recall that the algorithm maintains a sequence of sets $V(G) = H_0, H_1, \ldots, H_i$ and that at any point in time, we are dealing with $G[H = H_i]$; we call H the current context. Similar to the case of the recognition algorithm, we will argue that the total time spent finding a vertex v in $N(X)$ and an edge e in X such that v misses e is O(mn). However, in previous arguments, our time bound was derived from the fact that the rest of a single refinement of the set X in the current context took O(m) time. There are several such steps that take O(m) time; finding vertex v that misses edge e in $G[H]$ in the initialization phase of *find-handle* followed by the computation of sets X and Y, finding the connected component Y' of v in $G[H] - N(e)$, marking neighbors of Y', and finding the connected component containing e in the subgraph of $G[H]$ induced by $X - N(Y')$ (whose vertex set becomes the new value for X). Then, there are the operations of identifying a copair and restarting the algorithm *find-copair*.

We will show that if we ignore a total cost of O(mn), a single refinement step can be made to run in 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 O(mn) upper bound on the entire work done by the optimization algorithm.

We use the data structures NX and $NN(u)$ as maintained by the algorithm *find-handle*. Therefore, a vertex u of H belongs to $N(X)$ with respect to the current set X if and only if $NN(u) > 0$.

We will want to scan NX for each edge e once during the entire course of the algorithm so that the total cost of finding such pairs v and e is O(mn). Again, the idea is that with respect to a particular vertex v and an edge e, refinement is done at most once; for the refinement leaves edge e in X and vertex v outside of $X \cup N(X)$. If in the future v were to return to $N(X)$, edge e would already have been deleted from the graph. Therefore, when a future refinement occurs with respect to e, we need only scan further down the list NX to find an appropriate vertex that misses e. Also, for an edge e none of whose endpoints is ever found to be part of a copair, once a vertex v is found to miss (not miss) e, v will always miss (not miss) e. However, for an edge zw, where z is the vertex obtained by identifying a copair $\{x, y\}$, a vertex that previously saw one of the edges xw or yw could now miss the edge zw. We show later that this can be handled by scanning the list NX from the beginning for such an edge zw.

As far as implementation of identifying a copair $\{x, y\}$ into vertex z is concerned, we simply delete vertices x and y from the graph and from set X, introduce vertex z into the graph and into set X, and make $N(z) = N(x) \cap N(y)$.

THEOREM 6.2. *Algorithm wc-opt can be implemented to run in O(mn) time.*

PROOF. As noted in the proof of Theorem 4.2, it is easy to maintain the current context so that adjacencies are computed with respect to the current context.

We first consider the initialization step of *find-handle*. Barring the very first invocation (on the entire input graph), any invocation of *find-handle* is on some set $X = H$ that is found to be a current handle. For one such invocation, the time spent in the initialization phase is O(m_X). Therefore, the total time spent in initialization steps is bounded by the total number of edges induced by all the X sets, plus an extra cost of O(m) for the first invocation. Also, when a new value for set X is computed in the initialization step, we add $N(X)$ computed to the list NX, updating the $NN(u)$ value for each vertex u of $N(X)$, and we set the context to H.

We next consider the operation of finding a vertex v from $N(X)$ and an edge e in X such that v misses e (in order to refine the set X). For each edge e of X, we will already have checked to see whether e has missed neighbors of X up through some vertex w on NX; we start our search from w and continue. Since each edge is checked once against each vertex, the total work in finding v and e through the whole algorithm is $O(mn)$.

The next task which uses $O(m)$ time is computing Y', the component of $G[H] - N(e)$ containing vertex v, and then subsequently finding $N(Y')$. Set X', the component of $G[H] - N(Y')$ containing e, then becomes the new value of X.

We do not explicitly recompute the component Y'. Instead, we implement this by moving vertices from X. We make sure that vertices which must go to Y' will be placed in Y'. Also, those vertices that must be placed in $N(X')$ are placed there. We can then simply break the subgraph induced by the remaining set X to find X', namely the connected component containing e.

First, note that every member of $N(X)-N(e)$ will be added to Y'; these can easily be found in $O(n)$ time. If a vertex u 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 done in finding this component can all be charged to the edges in X, and hence this cost is $O(m_X)$.

We now must find neighbors of Y'. When a vertex y is moved into Y', we scan its adjacency list, appropriately updating the values maintained (if y is moved from X, all neighbors in $N(X)$ have their number of neighbors in X decreased by one), 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. Note that Y' subsequently becomes the set Y. A vertex can be moved into Y at most n-1 times, since a vertex will never return to X or $N(X)$ until identification of a copair is performed. Similarly, a vertex can be moved into $N(X)$ at most n-1 times, since it will not return to X before an identification of a copair is performed. Thus, the total time spent scanning adjacency lists while moving vertices to Y and $N(X)$ is $O(mn)$.

The final step of refinement, for which we allowed $O(m)$ time, was finding connected components in the new set X so that we can locate the component X' containing e; but this step clearly takes $O(m_X)$ time.

When a copair $\{x, y\}$ is identified into vertex z, we scan the neighbors of each of x and y to update the $NN(u)$ value for every neighbor u. Also for each edge zw, we set-up zw to point to the beginning of the list NX to facilitate a scan from the beginning. Scanning NX for such edges will cost us $O(\text{degree}(z)*n)$. However, as $\text{degree}(z) \leq \min(\text{degree}(x), \text{degree}(y))$, and as a vertex is deleted from the graph whenever a copair is identified, we can charge this cost to the vertex deleted. Thus, the overall cost of scans associated with all such edges zw is $O(mn)$.

Finally, as argued in the proof of Theorem 4.2, a restart after the identification of a copair can be done in $O(m)$ time. As a restart is done only after an identification of a copair, the overall time thus spent on restarts is $O(mn)$.

Therefore, the running time of the algorithm is $O(mn)$ plus the sum of the number of edges in all X sets created during the algorithm.

Each edge is part of (at most) n such X sets, as each time that an edge is placed in an X set, the size of X decreases by at least one. Thus the total size of all sets of edges induced by X sets is $O(mn)$, and our algorithm runs in $O(mn)$ time. □

Note that our optimization algorithm runs faster than the recognition algorithm. Thus, we can imagine a potential for the algorithm to make mistakes if the input is not weakly chordal, as 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 pair output by *find-copair* is a copair of the graph at that instant; this 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.

7. *Conclusions and Open Problems*

We have demonstrated the use of the notion of a handle in a graph, that is, a combination of a special type of connected component and separator, to design efficient algorithms for the class of weakly chordal graphs. We conclude the article by proposing some problems for further work.

With the recent result in Nikolopoulos and Palios [2004], disregarding the space complexity, the time complexities of the currently best algorithms for recognizing weakly chordal graphs and for finding a hole in a graph are the same. This seems counterintuitive: Given the structural properties exhibited by weakly chordal graphs, we would expect that recognizing such graphs would be computationally easier than determining whether an arbitrary graph has a hole. On top of this, we have demonstrated that optimizing on weakly chordal graphs can currently be done faster than recognizing weakly chordal graphs. Thus a natural problem is: Design an algorithm with a complexity of $o(m^2)$ for recognizing weakly chordal graphs. Another problem is improving the efficiency of the algorithms for weighted optimization problems on weakly chordal graphs.

REFERENCES

ARIKATI, S., AND RANGAN, C. 1991. An efficient algorithm for finding a two-pair, and its applications. *Discrete Appl. Math. 31*, 71–74.

BERRY, A., BORDAT, J. P., AND HEGGERNES, P. 2000. Recognizing weakly triangulated graphs by edge separability. *Nordic J. Comput. 7*, 164–177.

HAYWARD, R. B. 1985. Weakly triangulated graphs. *J. Combinatorial Theory Series B 39*, 200–209.

HAYWARD, R. B. 1997a. Meyniel weakly triangulated graphs. I. Co-Perfect orderability. *Discrete Appl. Math. 73*, 199–210.

HAYWARD, R. B. 1997b. Meyniel weakly triangulated graphs. II. A theorem of Dirac. *Discrete Appl. Math. 78*, 283–289.

HAYWARD, R. B., HOÀNG, C. T., AND MAFFRAY, F. 1989. Optimizing weakly triangulated graphs. *Graphs Combinatorics 5*, 339–349; erratum in *6* 1990, 33–35.

HAYWARD, R. B., SPINRAD, J. P., AND SRITHARAN, R. 2000. Weakly chordal graph algorithms via handles. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA) 42–49.

KRATSCH, D., AND SPINRAD, J. P. 2003. Between $O(mn)$ and $O(n^\alpha)$. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA). 158–167.

NIKOLOPOULOS, S. D., AND PALIOS, L. 2004. Hole and antihole detection in graphs. *In Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA). 843–852.

RAGHAVAN, V., AND SPINRAD, J. P. 2003. Robust algorithms for restricted domains. *J. Alg. 48*, 160–172.

ROSE, D. J., TARJAN, R. E., AND LEUKER, G. S. 1976. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comp. 5*, 266–283.

SPINRAD, J. P., AND SRITHARAN, R. 1995. Algorithms for weakly triangulated graphs, *Discrete Appl. Math. 19*, 181–191.

RECEIVED MARCH 2005; REVISED JUNE 2006; ACCEPTED JULY 2006