1995

Linear Time Optimization Algorithms for P4-Sparse Graphs

Beverly Jamison

Stephan Olariu
Old Dominion University

Original Publication Citation

Jamison, B., & Olariu, S. (1995). Linear-time optimization algorithms for P_4-sparse graphs. *Discrete Applied Mathematics, 61*(2), 155-175. doi:10.1016/0166-218x(94)00012-3

ELSEVIER

Discrete Applied Mathematics 61 (1995) 155–175

DISCRETE APPLIED MATHEMATICS

Linear time optimization algorithms for P_4-sparse graphs

Beverly Jamison[a], Stephan Olariu[b,*,1]

[a] *Department of Computer Sciene, Marymount University, Arlington, VA 22347, USA*
[b] *Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, USA*

Received 24 November 1992; revised 24 November 1993

Abstract

Quite often, real-life applications suggest the study of graphs that feature some local density properties. In particular, graphs that are unlikely to have more than a few chordless paths of length three appear in a number of contexts. A graph G is P_4-sparse if no set of five vertices in G induces more than one chordless path of length three. P_4-sparse graphs generalize both the class of cographs and the class of P_4-reducible graphs. It has been shown that P_4-sparse graphs can be recognized in time linear in the size of the graph. The main contribution of this paper is to show that once the data structures returned by the recognition algorithm are in place, a number of NP-hard problems on general graphs can be solved in linear time for P_4-sparse graphs. Specifically with an n-vertex P_4-sparse graph as input the problems of finding a maximum size clique, maximum size stable set, a minimum coloring, a minimum covering by clique, and the size of the minimum fill-in can be solved in $O(n)$ time, independent of the number of edges in the graph.

Keywords: Optimization; NP-hard problems; Resource allocation; Local density; Linear algorithms; Optimal algorithms

0. Introduction

Numerous computational problems in LAN technology [16, 18], group-based cooperation [17], cluster analysis [2, 12], scheduling [2–6], computational learning [8], and resource allocation [6] have suggested the study of graphs featuring local density properties. In these applications it is typical to equate local density with the absence of chordless paths of length three (referred to as P_4's). Several natural attempts to capture the notion of local density have motivated the study of the class of cographs [2–4, 13–15] and P_4-reducible graphs [9] corresponding, respectively, to the

*Corresponding author.

[1] This author was supported, in part, by the National Science Foundation under grants CCR-8909996 and CCR-9407180.

SSDI 0166-218X(94)00012-3

local density metrics described below:
(μ1) the graph contains no induced P_4;
(μ2) every vertex of the graph belongs to at most one induced P_4.

Recently, a new local density metric in graphs has been proposed [7, 10, 11]:
(μ3) every set of five vertices induces at most one P_4.

In practical applications, the metric (μ3) is less restrictive and thus more realistic than both (μ1) and (μ2). The class of graphs that naturally corresponds to this metric (the P_4-sparse graphs) features a number of remarkable properties [10, 11]. Recently, the authors have shown [10] that just as the cographs, the P_4-sparse graphs can be recognized in time linear in the size of the graph at hand. The purpose of this paper further exploit the data structures obtained during the recognition algorithm for the purpose of obtaining linear time solutions to a number of combinatorial optimization problems as we are about to explain.

It is well known that for a general graph G, the task of evaluating the following parameters

- $\omega(G)$ – the clique number of G – standing for the largest number of pairwise adjacent vertices in G,
- $\chi(G)$ – the chromatic number of G – representing the smallest number of colors used in a coloring of G such that adjacent vertices always receive distinct colors,
- $\alpha(G)$ – the stability number of G – defined as the largest number of pairwise nonadjacent vertices in G,
- $\theta(G)$ – the clique cover number of G – denoting the smallest number of cliques which cover all the vertices of G,
- $\phi(G)$ – the minimum fill-in of G – defined as the smallest number of edges that have to be added to G to obtain a chordal graph

is *hard.* More precisely, each of the problems of recognizing graphs G and integers k with $\omega(G) \geqslant k$, $\chi(G) \leqslant k$, $\alpha(G) \geqslant k$, $\theta(G) \leqslant k$, and $\phi(G) \leqslant k$ is NP-complete. Furthermore, the second and fourth problems are NP-complete even if the value of k is fixed, as long as $k \geqslant 3$, [5, 20]. We shall refer to these problems as *optimization problems.* Although the optimization problems defined above are intractable for general graphs, they are solvable in polynomial time for various particular classes of graphs that happen to be of import in practical applications. The reader is referred to Golumbic's comprehensive work where many of these results are summarized [6].

This paper proposes to solve the above optimization problems for the class of P_4-sparse graphs. More precisely, we are interested in solving instances of the following template problem:

Given a P_4-sparse graph G, compute a certain parameter $\pi(G)$.

The parameter $\pi(G)$ will be instantiated with "clique number", "chromatic number", "stability number", "clique cover number", and "minimum fill-in". In addition to returning the corresponding value of the desired parameter, we shall also return a "certificate" validating the value of the parameter. Typically, the certificate will be provided in the form of a set of vertices that achieve the desired parameter.

We note that the class of P_4-sparse graphs is a subclass of weak bipolarizable graphs [18] for which the first four parameters discussed above can be solved in $O(n + m)$ time for graphs with n vertices and m edges. However, there are a number of problems with these algorithms:

(a) they run in $O(n^2)$ time for dense graphs;
(b) they are not self-contained, in the sense that the algorithms depend on machinery designed for a different class of graphs;
(c) the technique used to solve them [18] does not seem to apply to computing other parameters, the minimum fill-in being a prime example.

By contrast, all the optimization algorithms discussed in this paper run in $O(n)$ time once the data structures returned by the recognition algorithm are in place. Furthermore, the technique that we propose in this paper applies not only to the first four parameters but also to other problems including computing the minimum fill-in, a maximum matching, the scattering number, among others.

In addition, our strategy for solving instances of the above template problem for P_4-sparse graphs is quite natural: we exploit the fact that the P_4-sparse graphs are precisely those C_5-free graphs for which a certain greedy algorithm always returns an induced cograph unique up to isomorphism.

As it turns out, once this cograph (referred to as the *canonical cograph*) is known, the optimization problems on P_4-sparse graphs can be solved elegantly by solving weighted versions of the same optimization problems on the canonical cograph. Therefore, we begin by providing extensions of known algorithmic results on cographs to handle the weighted case. Our contribution here is to also show how a certificate can be obtained efficiently. To the best of our knowledge these results are novel.

Next, the solution to the weighted optimization problem on the canonical cograph will be used in conjunction with the data structures produced in the linear-time recognition of P_4-sparse graphs [10]. The main contribution of this paper, as we see it, is to show that this combined information yields $O(n)$, and thus optimal, algorithms to solve the optimization problems mentioned above.

The remainder of this paper is organized as follows: Section 1 presents extensions of algorithmic problems on cographs that are instrumental in obtaining our linear time optimization algorithms for P_4-sparse graphs; Section 2 reviews fundamental results about P_4-sparse graphs; Section 3 details the proposed optimization algorithms; finally Section 4 summarizes the results.

1. The tools

All the graphs in this work are finite, with no loops nor multiple edges. In addition to standard graph-theoretical terminology compatible with Bondy and Murty [1], we use some new terms that we are about to define. In the context of trees, vertices will be called *nodes*. A *clique* is a set of pairwise adjacent vertices; a *stable set* is a set of pairwise nonadjacent vertices.

For a node w in a tree T, we let $p(w)$ stands for the patent of w in T, we let $T(w)$ stand for the subtree of T rooted at w; similarly, $L(w)$ denotes the set of leaves in $T(w)$; we let $G(w)$ represent the subgraph of G induced by $L(w)$; finally, we let $N(w)$ denote the number of vertices in $G(w)$.

For convenience, we shall assume that all the trees in this work are binary. To convince the reader that this is a reasonable assumption, we now point out an easy transformation that, with an arbitrary rooted tree T as input, returns a full binary tree BT (i.e. every internal node of BT has precisely two children). We proceed as follows: if a node x has degree k in T then, in BT, we add $k-2$ identical copies of x, namely $x_1, x_2, \ldots, x_{k-2}$ in such a way that, with x_0 standing for x,

- the parent of x_i is x_{i-1} whenever $i \geqslant 1$;
- the left child of x_i is the $(i+1)$st child of x in T;
- the right child of x_i is x_{i+1} in case $i \leqslant k-3$, and the kth child of x in T otherwise.

Now an easy counting argument shows that the number of nodes in BT is linear in the size of T, and that the conversion can be done in $O(|T|)$ time.

A graph G is called totally decomposable if there exists a rooted tree $T(G)$, unique up to isomorphism, whose leaves are the vertices of G and whose internal nodes correspond to certain graph operations. If $T(G)$ can be obtained efficiently, then a number of optimization problems can be solved efficiently for G [2–4, 15].

Lerchs [15] has shown that the cographs are totally decomposable; specifically, we can associate with every cograph G a unique rooted tree $T(G)$ called the *cotree* of G, featuring the following properties:

every internal node, except possibly for the root, has at least two children; furthermore, the root has only one child if, and only if, the underlying graph G is disconnected; (1.1)

the internal nodes are labeled by either 0 (0-nodes) or 1 (1-nodes) in such a way that the root is always a 1-node, and such that 1-nodes and 0-nodes alternate along every path in $T(G)$ starting at the root; (1.2)

the leaves of $T(G)$ are precisely the vertices of G; vertices x and y are adjacent in G if, and only if, the lowest common ancestor of x and y in $T(G)$ is a 1-node. (1.3)

For further reference, we note the following simple observation that follows directly from (1.3).

Observation 1.1. Let u be an arbitrary node in $T(G)$. Every leaf in $L(\text{root}(T)) - L(u)$ is either adjacent to all the leaves in $L(u)$ or to none of them.

In their comprehensive papers on cograhs, Corneil et al. [2, 3] noted that many algorithmic problems for cographs can be solved elegantly by performing a certain computation on the corresponding cotree. As a rule, they propose associating the leaves of the cotree with a weight of 1, while the internal nodes are associated with

various operators. This reduces the problem at hand to evaluating an algebraic expression on the cotree.

It is easy to generalize the results in [2, 3] in the following way: let G be a cograph and let $T(G)$ be the corresponding cotree. Let A be an arbitrary finite-carrier algebra and let $\square$ and $\bigcirc$ be two associative operations on A. We assume unit-time operations between the elements of A. If the leaves of $T(G)$ are labeled with elements of A, the 1-nodes are labeled by $\square$, and the 0-nodes of $T(G)$ are labeled by $\bigcirc$, then the cotree $T(G)$ can be perceived as computing an algebraic expression in the obvious way. Computationally, the corresponding expression can be evaluated efficiently by traversing the cotree in postorder, and performing the prescribed operation at every internal node. It is important to note that the postorder traversal guarantees that when it comes to evaluating the expression corresponding to an internal node, the corresponding operands are available.

The details of this simple procedure are spelled out as follows:

Procedure Evaluate(root(T), $\square$, $\bigcirc$);
1. **begin**
2. $v \leftarrow$ initial node of T in postoder;
3. **while** $v \neq$ root(T) **do begin**
4. **if** $p(v)$ is a 1-node **then**
5. value($p(v)$) $\leftarrow \square$(value(v), value($p(v)$))
6. **else**
7. value($p(v)$) $\leftarrow \bigcirc$(value(v), value($p(v)$));
8. $v \leftarrow$ next node in postorder
9. **end**
10. **end**; {Evaluate}

A simple inductive argument on the height of a generic node v in $T(G)$ proves the following lemma.

Lemma 1.2. *When procedure Evaluate terminates, for every node v in $T(G)$,* value(v) *represents the value of the subexpression corresponding to $G(v)$. Furthermore, the computation takes time proportional to the size of $T(v)$.*

Lemma 1.2 will be instantiated in a number of different ways, tailored to suit specific computational needs. For this purpose, we assume that the vertices of the cograph G have been weighted by nonnegative integers. Note that properties (1.1)–(1.3) together with Lemma 1.2 imply that for cographs the tasks of computing a maximum weight clique and a maximum weight stable set are dual to one another in the following strong sense.

Corollary 1.3. *For every node v in T(G), the following statements hold:*

- value(v) *returned by* Evaluate(root(T), +, max) *represents the size of a maximum weight clique in* $G(v)$;
- value(v) *produced by* Evaluate(root(T), max, +) *represents the size of a maximum weight stable set in* $G(v)$.

To begin, we provide the details of a simple algorithm to exhibit a maximum weight clique in a cograph. Our algorithm proceeds in two stages. The first stage involves the call Evaluate(root(T), +, max) as discussed above. In the second stage, we provide a certificate in the form of a set of vertices that constitute a maximum weight clique in the graph. The idea of the second stage is very simple: we traverse $T(G)$, starting at the root, in a way reminiscent of depth-first search. At every 1-node u, the search proceeds recursively for every child of u; at a 0-node, the search continues in one subtree only (this will be justified later). The details of the procedure implementing the second stage follow.

```
Procedure Max_Weight_Clique(v);
1.  begin
2.    if v is a leaf then
3.      mark v
4.    else
5.      if v is a 1-node then
6.        for every child w of v do
7.          Max_Weight_Clique(w)
8.      else begin {v is a 0-node}
9.        w ← left_child(v);
10.       while value(w) ≠ value(v) do
11.           w ← right_child(v);
12.         Max_Weight_Clique(w)
13.   end; {if}
14. end; {Max_Weight_Clique}
```

The correctness of the above procedure hinges on the following technical result.

Lemma 1.4. *Let u be an arbitrary* 1*-node in* $T(G)$. *If a maximum weight clique* C *contains leaves from* $L(u)$, *then* C *contains leaves of nonzero weight from every subtree of* $T(u)$. *Similarly, let* v be an arbitrary 0-node in $T(G)$. *If a maximum weight clique* C *contains leaves from* $L(v)$, *then these leaves belong to exactly one subtree of* $T(v)$.

Proof. Let C be an arbitrary maximum weight clique in the cograph G. Let u be a 1-node in $T(G)$ such that for children v and w of u, C contains leaves from $T(v)$ but

not from $T(w)$. Observation 1.1 guarantees that if $L(w)$ contains a leaf x of nonzero weight, then C can be augmentd by the addition of x, contradicting its maximality. Similarly, if v is a 0-node then, by property (1.3), no clique can contain nodes from distinct subtrees of $T(v)$. The conclusion follows. □

Theorem 1.5. *Once the cotree $T(G)$ of an n-vertex cograph G is available, procedure* Max_Weight_Clique *correctly returns a maximum weight clique in G in $O(n)$ time.*

Proof. To begin, we note that

any leaves marked by procedure Max_Weight_Clique are adjacent in G. (1.4)

[To justify this claim, let u, v be arbitrary marked leaves. Now lines 8–12 in the procedure guarantee that the lowest common ancestor of u and v cannot be a 0-node. The conclusion is implied by (1.3).]

By virtue of (1), the set C of marked leaves is indeed a clique in G. The fact that C is a clique of maximum weight follows directly from the fact that, by Corollary 1.3, the effect of the call Evaluate(root(T), +, max) performed in the first stage of the algorithm is to compute for every node v *in* $T(G)$, the size of the maximum weight clique in $T(v)$.

To complete the proof of Theorem 1.5, we only need observe that stage 1 of the algorithm runs in time linear in the size of $T(G)$; in the second stage the while loop in lines 10–11 takes time proportional to the number of children of node v in $T(G)$. Therefore, the overall time is linear in the size of $T(G)$, as claimed. □

The following result follows by a mirror argument.

Theorem 1.6. *Once the cotree $T(G)$ is available, the maximum weight stable set in an n-vertex cograph G can be computed in $O(n)$ time.*

Next, we discuss a linear time algorithm to compute a weighted coloring of a cograph. Again, we assume an n-vertex cograph, the corresponding cotree $T(G)$, and an assignment of nonnegative integer weights to the vertices of G. For the purpose of this paper, the weighted version of the graph coloring problem involves assigning to each vertex a number of colors equal to the weight of that vertex (in particular, vertices of zero weight receive no colors). We further assume that the colors are represented by positive integers. Our weighted coloring algorithm assigns to every vertex in the graph a range of colors. When the algorithm terminates, every leaf w of $T(G)$ stores an ordered pair $(\text{weight}(w), \text{color}(w))$ with the implication that w is assigned consecutive colors from $\text{color}(w) - \text{weight}(w) + 1$ through $\text{color}(w)$. As we are about to demonstrate, this color assignment is optimal in the sense that is uses the least number of colors. It is also helpful to notice that the coloring we obtain is an interval coloring.

Again, our algorithm proceeds in two stages: the first stage involves calling Evaluate(root(T), +, max), while the second traverses the tree from the root down to the leaves assigning ranges of colors to the nodes of the tree. Note that by Corollary 1.3, for every node x in $T(G)$, value(v) computed by the call Evaluate(root(T), +, max) equals the maximum weight of a clique in $G(v)$. In particular, value(root(T)) contains the maximum weight of a clique in G. It is relatively straightforward to see that this also represents the least number of colors in a coloring of the cograph G' whose cotree is obtained from $T(G)$ by replacing every leaf w of weight h with h pairwise adjacent siblings of weight 1.

The second stage of the algorithm proceeds top-down from the root to the leaves: at every 1-node, the range of colors inherited from its parent is partitioned among its children; at every 0-node the same range is shared by its children. The details are spelled out by the following pseudo-code that we present for a generic node v of $T(G)$. Here, the parameter k represents the first available color that can be used on $G(v)$.

```
Procedure Weighted_Color(v, k);
1.  begin
2.    if v is a leaf and value(v) > 0 then begin
3.      k ← k + value(v);
4.      color(v) ← k
5.      end
6.    else
7.      if v is a 0-node then begin
8.        oldk ← maxk ← k;
9.        for every child w of v do begin
10.           k' ← oldk;
11.           Weighted_Color(w, k');
12.           if k' > maxk then
13.             maxk ← k'
14.       end; {for}
15.       k ← maxk
16.       end {0-node handling}
17.     else {handle a 1-node}
18.       for every child w of v do
19.         Weighted_Color(w, k)
20. end; {Weighted_Color}
```

Theorem 1.7. *Procedure* Weighted_Color *correctly produces a weighted coloring of* $G(v)$ *using the least number of colors. Furthermore, the running time is linear in the size of* $T(v)$.

Proof. To argue for the running time, note that procedure Weighted_Color is called recursively once for each node in the tree. Other than the recursive calls, each node is processed in a constant number of steps.

To settle the correctness, it is helpful to mentally replace every leaf w of weight h, by h siblings $w_1, w_2, \ldots, w_h$ of w each carrying a weight of 1. Therefore, the weighted coloring problem reduces to the standard coloring problem on the new cograph G'. Since the cographs are perfect in the sense of Berge [6], it follows that for every internal node v, value(v) returned by Evaluate(root(T), +, max) equals the least number of colors in a coloring of $G(v)$.

Therefore, we only need argue that this is the number of colors used. For this purpose, we proceed by induction on the size of $T(v)$. The statement holds trivially for leaves. If v is a 0-node, the reassignment of k in line 10 to the initial value associated with the 0-node together with the induction hypothesis guarantees that the number of colors used will not exceed value(v). Clearly, in the case of a 1-node no colors can be reused. The conclusion follows. □

Note that for a cograph there is a strong duality between weighted coloring and the problem of assigning ranges of clique numbers in a minimum clique cover of the complement of the graph. This is easily obtained by running Evaluate(root(T), max, +) to obtain value(v) for every node of T, followed by running the procedure Weighted_Color. What results is a "weighted coloring" of the complement of G: a range of colors at a leaf should be interpreted as a range of clique numbers in a minimum weight clique cover of G.

Therefore, we have the following result.

Theorem 1.8. *Once the cotree $T(G)$ is available, the minimum weight clique cover problem in an n-vertex cograph G can be solved in $O(n)$ time.*

As another example, consider the problem of chordal graph completion or *minimum fill-in*: the problem asks for the least number of vertices that have to be added to make an arbitrary graph into a chordal graph. Yannakakis [19] showed that the decision version of this problem is NP-complete. However, when the input is restricted for particular classes of graphs, the problem becomes polynomial. In particular, Corneil et al. [3] demonstrated that the cotree associated with a cograph can be used to compute the minimum fill-in efficiently. Recall that we assume that the cotree is a binary tree (should this not be the case, we can binarize it in time linear in the size of the tree).

In order to compute the minimum fill-in for cographs Corneil et al. [3] introduced two new parameters. Specifically, with every node v of the cotree they associate nonnegative numbers $Q(v)$, and $F(v)$: $Q(v)$ is the number of nonadjacencies in $G(v)$, while $F(v)$ is the size of the minimum fill-in corresponding to $G(v)$. In [3] the following formulas are given for computing these parameters:

- for every leaf v, $F(v) = Q(v) = 0$ and $N(v) = 1$;

- for every 1-node v with children $v_1, v_2, \ldots, v_k$, set

$$N(v) = \sum_{i=1}^{k} N(v_i); \qquad Q(v) = \sum_{i=1}^{k} Q(v_i);$$

$$F(v) = \min_{i=1}^{k} \left\{ F(v_i) + \sum_{j=1; j \neq i}^{k} Q(v_j) \right\}; \tag{1.5}$$

- for every 0-node v with children $v_1, v_2, \ldots, v_k$, set

$$N(v) = \sum_{i=1}^{k} N(v_i); \qquad Q(v) = \frac{1}{2} \sum_{i=1}^{k} N(v_i) * [N(v) - N(v_i)];$$

$$F(v) = \sum_{i=1}^{k} F(v_i). \tag{1.6}$$

As it turns out, formulas (1.5) and (1.6) become more tractable in the context of binary trees. By way of motivation, and for further reference, consider an *arbitrary* graph G whose vertex set partitions into nonempty, disjoint sets A and B such that every vertex in A is adjacent to all the vertices in B. Let $Q(A)$ and $F(A)$ (resp. $Q(B)$ and $F(B)$) stand for the number of nonadjacencies in A and for the size of the minimum fill-in corresponding to A (resp. B). The following simple results will be instrumental in understanding how the minimum fill-in algorithm works.

Observation 1.9. G is a chordal graph if and only if at least one of the graphs induced by A and B is a clique.

Observation 1.10. The following hold for G:

$$Q(G) = Q(A) + Q(B) \quad \text{and} \quad F(G) = \min\{F(A + Q(B), F(B) + Q(A)\}.$$

(The first part is trivial; the second part follows directly from Observation 1.9.)

Note that Observations 1.9 and 1.10 along with property (1.3) afford us the following simpler versions of (1.5) and (1.6) in the presence of a binarized cotree. As before, consider a generic node v with left and right children u and w, respectively. Then $N(v)$, $Q(v)$, and $F(v)$ are computed as follows:

- if v is a 1-node then

$$N(v) = N(u) + N(w); \qquad Q(v) = Q(u) + Q(w);$$

$$F(v) = \min\{F(u) + Q(w), F(w) + Q(u)\}; \tag{1.7}$$

- if v is a 0-node then

$$N(v) = N(u) + N(w); \qquad Q(v) = Q(u) + Q(w) + N(u) * N(w);$$

$$F(v) = F(u) + F(w). \tag{1.8}$$

Note that we can define weighted versions of the above formulas: the only difference is in the original assignment of weights at the leaves. Furthermore, once the assignment

of weights at the leaves is done, computing $N(v)$, $Q(v)$, and $F(v)$ for an arbitrary node v of the cotree is easy. To wit, Evaluate(root(T), +, +) determines $N(v)$; to compute $Q(v)$ we use a new procedure Evaluate_Q(root(T), +, +) that is exactly like Evaluate except that line 5 (i.e. processing of 0-nodes) is altered by adding $N(u)N(w)$ to the partial result computed by Evaluate(root(T), +, +); finally to compute $F(v)$ we use a new procedure Evaluate_F(root(T), +, +) that is exactly like Evaluate except that we replace line 7 (i.e. processing of 1-nodes) with $\min\{F(u) + Q(w), F(w) + Q(u)\}$. To summarize our findings we state the following result.

Theorem 1.11. *Once the cotree $T(G)$ of an n-vertex cograph G is available, the weighted version of the minimum fill-in can be computed in G in $O(n)$ time.*

In the particular case where the weights at the leaves are $N(v) = 1$; $Q(v) = F(v) = 0$ we obtain the result in [3]. The usefulness of the general weighted case will become apparent in computing the minimum fill-in for P_4-sparse graphs.

2. P_4-sparse graphs, an overview

Let G be an arbitrary graph. We let the following greedy procedure return a cograph obtained by removing an arbitrary endpoint of every P_4 in G.

Procedure Greedy(G);
{Input: an abitrary graph G;
Output: a cograph $C(G)$}
begin
  $C(G) \leftarrow G$;
    while there exist P_4's in $C(G)$ **do begin**
      pick a P_4 $uvxy$ in $C(G)$;
      pick z abitrarily in $\{u, y\}$;
      $C(G) \leftarrow C(G) - \{z\}$
    end;
    return($C(G)$)
end;

The following surprising result will be instrumental in understanding how our algorithms work.

Proposition 2.1 (Jamison and Olariu [11]). *Let G be a graph with no induced C_5. G is P_4-sparse if, and only if, for every induced subgraph H of G, $C(H)$ is unique up to isomorphism.*

Proposition 2.1 justifies referring to the cograph $C(G)$ of a P_4-sparse graph as the *canonical* cograph of G.

An arbitrary graph G is said to have a *special partition* if there exists a family $\Sigma = \{S_1, S_2, \ldots, S_q\}$ $(q \geqslant 1)$ of disjoint stable sets of G with $|S_i| \geqslant 2$ $(1 \leqslant i \leqslant q)$ and an injection

$$f: \bigcup_{i=1}^{q} S_i \to V - \bigcup_{i=1}^{q} S_i$$

such that the following two conditions are satisfied:

$$K_i = \{z \mid z = f(s) \text{ for some } s \text{ in } S_i\} \text{ is a clique for all } i\ (1 \leqslant i \leqslant q): \tag{2.1}$$

a set A of vertices induces a P_4 in G if, and only if, there exists a subscript i $(1 \leqslant i \leqslant q)$ and distinct vertices x, y in S_i such that $A = \{x, y, f(x), f(y)\}$. (2.2)

For further references, we let S and K stand for $\bigcup_{i=1}^{q} S_i$ and $\bigcup_{i=1}^{q} K_i$, respectively. The following results provide insight into the structure of P_4-sparse graphs that will be used for the purpose of developing our linear time algorithms.

Proposition 2.2 (Jamison and Olariu [10]). *A graph is P_4-sparse if, and only if, it is a cograph or it has a special partition.*

Proposition 2.3 (Jamison and Olariu [10]). *Let G be a P_4-sparse graph with a special partition. For every i $(1 \leqslant i \leqslant q)$, the following conditions are satisfied:*
either
- $N(s) \cap K_i = \{f(s)\}$ *for every* $s \in S_i$,

or else
- $N(s) \cap K_i = K_i - \{f(s)\}$ *for every* $s \in S_i$.

Proposition 2.4 (Jamison and Olariu [10]). *Let $G = (V, E)$ be a P_4-sparse graph. For every i $(1 \leqslant i \leqslant q)$, if a vertex in $V - K_i$ is adjacent to a vertex in S_i, then it is adjacent to all the vertices in $K_i \cup S_i$.*

It is easy to see that given an arbitrary P_4-sparse graph $G = (V, E)$, the canonical cograph $C(G)$ contains all the vertices in $V - S$, along with precisely one vertex in every S_i $(1 \leqslant i \leqslant q)$. The optimal recognition algorithm for P_4-sparse graphs [10] relies crucially on the fact that we can uniquely retrieve a P_4-sparse graph G from the tuple $(C(G), SK)$. Here, SK can be thought of as an array such that for every i $(1 \leqslant i \leqslant q)$, $SK[i]$ contains the pair (S_i, K_i) of the special partition of G, in a way that naturally associates with every s in S_i its unique image $f(s)$, along with a bit indicating whether or not s and $f(s)$ are adjacent.

3. Algorithms for optimizing P_4-sparse graphs

Throughout this section, $G = (V, E)$ will represent an arbitrary P_4-sparse graph. We assume that the ordered pair $(C(G), SK)$ is available. Furthermore, we assume that $C(G)$ is represented by its (binarized) cotree $T(G)$. Due to the fact that $C(G)$ contains all of the vertices from K and only some of the vertices from S, the duality that was present in cographs no longer holds. In particular, the canonical cograph of the complement of G is not necessarily the complement of the canonical cograph of G. We can compensate for this loss of duality by assigning appropriate weights to the leaves of the canonical cotree.

To begin, we address the problem of exhibiting a maximum clique in G. For later reference, we shall make an observation that will justify our algorithmic approach.

Lemma 3.1. *G contains a maximum clique with no vertices from S.*

Proof. To see this, let M be a maximum clique in G. If M contains a vertex s_i from S_i, then by virtue of Propositions 2.3 and 2.4 combined, the set obtained by replacing s_i with a vertex k_i in K_i, nonadjacent to s_i, is a clique of cardinality $|M|$. The conclusion follows. □

Lemma 3.1 guarantees the existence of a maximum clique that contains no vertices from S. This, in fact, justifies assigning weights to the vertices of G (and implicitly to the leaves of $T(G)$) as follows:

- every vertex in S receives a weight of 0;
- all other vertices of G receive a weight of 1.

In order to find a maximum clique in G, we proceed in the following two stages:

- first, we run procedure Evaluate(root(T), +, max) with the leaves weighted as above;
- next, we run the procedure Max_Weight_Clique on $T(G)$.

The details are spelled out in procedure Maximum_Clique that we present next.

Procedure Maximum_Clique(G);
1. **begin**
2. **for** all vertices v of G **do**
3. **if** $v \in S$ **then**
4. weight(v) ← 0
5. **else**
6. weight(v) ← 1;
7. Evaluate(root(T), +, max);
8. Max_Weight_Clique(root(T))
9. **end**; {Maximum_Clique}

Theorem 3.2. *Procedure* Maximum_Clique *correctly returns a maximum clique in an n-vertex P_4-sparse graph G in* $O(n)$ *time.*

Proof. Follows immediately from Corollary 1.3, Theorem 1.6, and Lemma 3.1 combined. □

Next, we shall present an optimal coloring algorithm for P_4-sparse graphs. Our coloring strategy is as follows: we begin by producing an optimal coloring of the subgraph of G induced by $V - S$. We then complete the coloring by assigning colors to the vertices in S. As it turns out, this can be done without the need of introducing new colors. The details are contained in procedure SK_Color.

Procedure SK_Color(T);
1. **begin**
2. **for** every i **do begin**
3. **for** every s in S_i **do begin**
4. **if** s is adjacent to $f(s)$ **then begin**
5. choose a vertex k_i in $K_i - \{f(s)\}$;
6. color(s) ← color(k_i)
7. **end**
8. **else**
9. color(s) ← color($f(s)$)
10. **end**
11. **end**
12. **end**; {SK_Color}

Note that lines 6 and 9 in procedure SK_Color guarantee that the colors assigned to the vertices in S_i occur among the colors received by the vertices in K_i, and so no new colors are used. Therefore we state the following obvious result.

Lemma 3.3. *If an optimal coloring of the graph induced by $V - S$ is available, then the coloring returned by procedure* SK_Color *is an optimal coloring of G.*

We are now in a position to show how all the parts fit together. We begin by assigning weights to the nodes of $T(G)$ using the same scheme as before: nodes in S receive a weight of 0, all the other nodes receive a weight of 1. This ensures that procedure Weighted_Color will return an optimal coloring of the desired subgraph. To see that this is the case, note that line 2 in procedure Weighted_Color disqualifies nodes with a weight of 0 from receiving any colors in the coloring. The correctness now follows from Theorem 1.7.

Finally, Lemma 3.3 asserts that the coloring produced can be extended to the whole graph. The fact that no new colors are used, guarantees the optimality of the coloring of G. It is important to note that the entire computation can be performed in linear time.

Procedure Optimal_Coloring(G);
1. **begin**
2. **for** all vertices v of G **do**
3. **if** $v \in S_i$ **then**
4. weight(v) $\leftarrow 0$
5. **else**
6. weight(v) $\leftarrow 1$;
7. Evaluate(root(T), + , max);
8. $k \leftarrow 0$;
9. Weighted_Color(root(T), k);
10. SK_Color(T)
11. **end**; {Optimal_Coloring}

To summarize our findings we state the following result.

Theorem 3.4. *Procedure* Optimal_Coloring *correctly returns an optimal coloring of a P_4-sparse graph G in time linear in the size of the graph.*

Proof. Follows directly from Corollary 1.3, Lemma 3.3 and Theorem 1.7 combined. □

We now turn to the problem of computing a maximum stable set in a P_4-sparse graph. Note that the duality arguments relating maximum cliques and maximum stable sets that used to work in the context of cographs do not extend to P_4-sparse graphs. The reason lies in the fact that the sets S and K that describe the structure of a P_4-sparse graph G are not perfectly symmetric. Clearly, the canonical cotree of the complement $\bar{G}$ of G need not coincide with the tree obtained by interchanging 0's and 1's in the internal nodes of $T(G)$.

Lemma 3.5. *G contains a maximum stable set with no vertices from K.*

Proof. To justify this claim, let I be a maximum stable set in G. If I contains a vertex k_i from K_i, then the contrapositive of Proposition 2.4 guarantees that the set obtained by replacing k_i with a vertex s_i in S_i, adjacent to k_i, is a stable set of cardinality $|I|$. The conclusion follows. □

Lemma 3.5 guarantees the existence of a maximum stable set that contains no vertices from K. As a consequence of Proposition 2.4, once a vertex in S_i belongs to a maximum stable set, the whole set S_i is included in the stable set. We plan to use this fact, along with Lemma 3.5 to provide an appropriate weighting scheme to compensate for the lack of duality between maximum cliques and maximum stable sets discussed above. Specifically, we assign weights to the vertices of G (and implicitly to the leaves of $T(G)$) as follows:

- $|K_i|$ to the unique vertex from S_i present in $T(G)$;
- 0 to all vertices in K;
- 1 to all remaining vertices.

In order to find a maximum stable set in G, we proceed in the following stages:

- first, we assign weights to the vertices of G as described above;
- next, we run Evaluate(root(T), max, +);
- finally, we run procedure Max_Weight_Clique.

The details are spelled out by procedure Maximum_Stable_Set that we present next.

```
Procedure Maximum_Stable_Set(G);
1.  begin
2.    for all vertices v of C(G) do
3.      if v ∈ S_i then
4.        weight(v) ← |K_i|
5.      else
6.        if v ∈ K then
7.          weight(v) ← 0;
8.        else
9.          weight(v) ← 1;
10.   Evaluate(root(T), max, +);
12.   Max_Weight_Clique(root(T))
13. end; {Maximum_Stable_Set}
```

The following result follows immediately from the previous discussion along with Lemma 3.5 and Theorem 2.5.

Theorem 3.6. *Procedure* Maximum_Stable_Set *correctly returns, in linear time, a maximum stable set in a P_4-sparse graph G.*

Next, we shall present an algorithm to exhibit a minimum clique cover for a P_4-sparse graph. Note that this problem is not the dual of optimal coloring: the problem is that most of the vertices of S are absent from $T(G)$. However, the weighted form of

the cograph coloring algorithm discussed in Section 2, permits us to obtain a range of clique numbers for the single vertex in every S_i present in the canonical cotree. We can then distribute the clique assignments among the vertices in S. By weighting the vertices in K by 0, we can postpone the assignment of clique numbers to these vertices until after the clique numbers are distributed to the vertices in S. Our strategy of implementing ideas is as follows: we use the same weighting scheme as for the maximum stable set problem discussed above. Once the leaves of the canonical cotree have been weighted, we call Evaluate(root(T), max, +) and then procedure Weighted_Color(root(T), 0); notice that in this case, procedure Weighted_Color assigns a range of clique numbers to the unique vertex in every S_i that is present in $T(G)$, while no numbers are assigned to vertices in K. Next, we distribute clique numbers to all vertices in S. Finally, once this is done, we assign clique numbers to the vertices in K, by associating every vertex in S_i with a vertex in K_i adjacent to it. Therefore, no new clique numbers are used in this operation. The details of assigning clique numbers to vertices in $S \cup K$ follow.

Procedure SK_Cover(T);
1. **begin**
2. **for** every i **do begin**
{assume that s_i belongs to $T(G)$)};
3. distribute the range of colors of s_i to all vertices in S_i;
4. **for** every k in K_i **do**
5. **if** k is not adjacent to $f^{-1}(k)$ **then**
6. clique(k_i) ← clique($f^{-1}(k_i)$)
7. **else** {now k is adjacent to $f^{-1}(k)$}
8. **for** every k_i in K_i **do begin**
9. find $s_j \in S_i$ distinct from $f^{-1}(k_i)$
10. clique(k_i) ← clique(s_j))
11. **end** {{for}
12. **end** {for}
13. **end**; {SK_Cover}

Note that lines 6 and 10 in procedure SK_Cover guarantee that the clique numbers assigned to the vertices in K_i occur among the clique numbers received by the vertices in S_i, and so no new clique numbers are used. Therefore we state the following result whose proof is immediate.

Lemma 3.7. *If, after having executed line* 3 *in* SK_Cover, *we have a minimum clique cover of the subgraph of G induced by* $V - K$, *then when* SK_Cover *terminates, it returns a minimum clique cover of G itself.*

We are now in a position to show how all the parts fit together. Our strategy is to produce a minimum clique cover of the subgraph of G induced by $V - K$, and then to extend this cover to a minimum clique cover of G. We begin by assigning weights to the nodes of $T(G)$ using the same scheme as before: nodes in S_i receive a weight of $|K_i|$, the nodes in K_i receive a weight of 0, while all the other nodes receive a weight of 1. Next, we invoke Evaluate(root(T), max, +) and then procedure Weighted_Color obtaining an assignment of ranges of clique numbers to the nodes in S present in $T(G)$, no numbers assigned to nodes in K and a unique number assigned to all other nodes. Furthermore, after line 3 in procedure SK_Cover has been executed, we obtain a minimum clique cover of the subgraph induced by $V - K$. Now Lemma 3.7 guarantees that when SK_Cover ends we have the desired result. The details are spelled out next.

```
Procedure Minimum_Clique_Cover(G);
1.  begin
2.    for all vertices v of G do
3.      if v ∈ S_i then
4.        weight(v) ← |K_i|
5.      else
6.        if v ∈ K then
7.          weight(v) ← 0;
8.        else
9.          weight(v) ← 1;
10.   Evaluate(root(T), max, +);
11.   k ← 0;
12.   Weighted_Color(root(T), k));
13.   SK_Cover(T)
14. end; {Minimum_Clique_Cover}
```

In summary, we can state the following result.

Theorem 3.8. *Procedure* Minimum_Clique_Cover *correctly results in linear time a minimum clique cover for a P_4-sparse graph G.*

To compute the size of the minimum fill-in for P_4-sparse graphs we note that as a consequence of Proposition 2.4, once a vertex in S_i is incident to an edge added in a minimum fill-in, in the whole set S_i has this property. We plan to use this observation along with Observations 1.9 and 1.10 to provide an appropriate weighting scheme to compensate for the absence of vertices in S_i in the canonical cograph.

Specifically, we assign weights to the vertices of G (and implicitly to the leaves of $T(G)$) as follows:

- if v is the unique vertex from S_i present in $T(G)$ then we weight v as follows:

  $N(v) \leftarrow |K_i|$;

  $Q(v) \leftarrow |K_i| * (|K_i| - 1)/2$;

  $F(v) \leftarrow 0$;

- with all remaining leaves we associate

  $N(v) \leftarrow 1$;

  $Q(v) \leftarrow 0$;

  $F(v) \leftarrow 0$.

In order to compute the size of the minimum fill-in in G, we proceed in the following stages:

- first, we assign weights to the vertices of G as described above;
- run Evaluate(root(T), + , +) to compute $N(v)$ at all internal nodes v;
- run Evaluate_Q(root(T), + , +) to compute $Q(v)$ at all internal nodes v;
- run Evaluate_F(root(T), + , +) to compute $F(v)$ at all internal nodes v.

The details are spelled out by procedure Minimum_Fill_In that we present next.

Procedure Minimum_Fill_In(G);
1. **begin**
2. **for** all vertices v of G **do**
3. **if** $v \in S_i$ **then begin**
4. $N(v) \leftarrow |K_i|$;
5. $Q(v) \leftarrow |K_i| * (|K_i| - 1)/2$;
6. $F(v) \leftarrow 0$
7. **end**
8. **else begin**
9. $N(v) \leftarrow 1$;
10. $Q(v) \leftarrow 0$;
11. $F(v) \leftarrow 0$
12. **end**
13. Evaluate(root(T), + , +); {compute $N(v)$}
14. Evaluate_Q(root(T), + , +); {compute $Q(v)$}
15. Evaluate_F(root(T), + , +); {compute $F(v)$}
16. **end**; {Minimum_Fill_In}

The following results follows immediately from the previous discussion along with Proposition 2.4 and Observations 1.9 and 1.10.

Theorem 3.9. *Procedure* Minimum_Fill_In *correctly returns, in linear time, the size of the minimum fill-in in a P_4-sparse graph G.*

4. Conclusions and open problems

We have presented optimal time algorithms to solve a number of optimization problems for P_4-sparse graphs, by relying on the data structures returned by the optimal recognition algorithm in [10].

Our approach is motivated by the fact that a P_4-sparse graph is uniquely determined by an ordered pair $(C(G), SK)$ where $C(G)$ is a cograph obtained by removing one endpoint of every P_4 in the graph. A surprising characterization of P_4-sparse graphs guarantees that the result is always unique up to isomorphism. The SK component contains information about the P_4 structure of the original graph.

We note that the class P_4-sparse graphs is a subclass of weak bipolarizable graphs [18] for which the first four parameters discussed above can be solved in $O(n + m)$ time for graphs with n vertices and m edges. However there are a number of problems with these algorithms:

(a) they run in $O(n^2)$ time for dense graphs;
(b) they are not self-contained, in the sense that the algorithms depend on machinery designed for a different class of graphs;
(c) the technique used to solve them [18] does not seem to apply to computing other parameters, the minimum fill-in being a prime example.

The recognition algorithm in [10] captures the structure imposed by the local density property ($\mu 3$) into a tree representation that can be exploited to obtain optimization algorithms running in time proportional to the number of vertices in the graph, irrespective of the number of edges. Moreover, these techniques work not only for the first four parameters but also for computing the minimum fill-in, a maximum matching, the scattering number, a minimum path cover, hamiltonicity, among others.

It would be interesting to see what other algorithmic problems on P_4-sparse graphs can be solved in $O(n)$ time using similar techniques. This promises to be an exciting area for further work.

Acknowledgement

The authors are indebted to two anonymous referees for many constructive comments.

References

[1] J.A. Bondy and U.S.R. Murty, Graph Theory with Applications (North-Holland, Amsterdam, 1976).
[2] D.G. Corneil, H. Lerchs and L.S. Burlingham, Complement reducible graphs, Discrete Appl. Math. 3 (1981) 163–174.

[3] D.G. Corneil, Y. Perl and L.K. Stewart, Cographs: recognition, applications, and algorithms, Congr. Numer. 43 (1984) 249–258.
[4] D.G. Corneil, Y. Perl and L.K. Stewart, A linear recognition algorithm for cographs, SIAM J. Comput. 14 (1985) 926–934.
[5] M.R. Garey and D.S. Johnson, Computers and Intractability, a Guide to the Theory of NP-completeness (W.H. Freeman, San Francisco, CA, 1979).
[6] M.C. Golumbic, Algorithm Graph Theory and Perfect Graphs (Academic Press, New York, 1980).
[7] C. Hoàng, Doctoral Thesis, McGill University, Montreal (1985).
[8] L. Hellenstein and M. Karpinski, Learning read-once formulas using membership queries, in: Proceedings 2nd Annual Workshop on Computational Learning Theory (1989) 146–161.
[9] B. Jamison and S. Olariu, P_4-reducible graphs, a class of uniquely tree representable graphs, Stud. Appl. Math. 81 (1989) 79–87.
[10] B. Jamison and S. Olariu, A linear-time recognition algorithm for P_4-sparse graphs, SIAM J. Comput. 21 (1992) 381–406.
[11] B. Jamison and S. Olariu, A tree representation for P_4-sparse graphs, Discrete Appl. Math. 35 (1992) 115–129.
[12] H.A. Jung, On a class of posets and the corresponding comparability graphs, J. Combin. Theory Ser. B 24 (1978) 125–133.
[13] A. Kelmans, The number of trees in graph I, Automat. Remote Control 26 (1965) 2194–2204.
[14] A. Kelmans, The number of trees in graph II, Automat. Remote Control 27 (1966) 56–65.
[15] H. Lerchs, On the clique-kernel structure of graphs, Department of Computer Science, University of Toronto (October 1972).
[16] J.O. Limb and C. Flores, Description of Fasnet – A unidirectional local area communication network, BSTJ 61 (1982) 1413–1440.
[17] P.K. McKinley and J.W.S. Liu, Multicast tree construction in bus-based networks, Comm. ACM 33 (1990) 29–42.
[18] S. Olariu, Weak bipolarizable graphs, Discrete Math. 74 (1989) 159–171.
[19] M. Yannakakis, Computing the minimum fill-in is NP-complete, SIAM J. Comput. 2 (1981) 77–79.