On the Recognition of P_4-Comparability Graphs

Stavros D. Nikolopoulos and Leonidas Palios

Department of Computer Science, University of Ioannina
GR-45110 Ioannina, Greece
{stavros,palios}@cs.uoi.gr

Abstract. We consider the problem of recognizing whether a simple undirected graph is a P_4-comparability graph. This problem has been considered by Hoàng and Reed who described an $O(n^4)$-time algorithm for its solution, where n is the number of vertices of the given graph. Faster algorithms have recently been presented by Raschle and Simon and by Nikolopoulos and Palios; the time complexity of both algorithms is $O(n+m^2)$, where m is the number of edges of the graph.
In this paper, we describe an $O(n\,m)$-time, $O(n+m)$-space algorithm for the recognition of P_4-comparability graphs. The algorithm computes the P_4s of the input graph G by means of the BFS-trees of the *complement* of G rooted at each of its vertices, without however explicitly computing the complement of G. Our algorithm is simple, uses simple data structures, and leads to an $O(n\,m)$-time algorithm for computing an acyclic P_4-transitive orientation of a P_4-comparability graph.

Keywords: Perfectly orderable graph, comparability graph, P_4-comparability graph, recognition, P_4-component, P_4-transitive orientation.

1 Introduction

We consider simple non-trivial undirected graphs. Let $G = (V, E)$ be such a graph. An *orientation* of G is an antisymmetric directed graph obtained from G by assigning a direction to each edge of G. An orientation (V, F) of G is called *transitive* if it satisfies the following condition: $\overrightarrow{ab} \in F$ and $\overrightarrow{bc} \in F$ imply $\overrightarrow{ac} \in F$, for all $a, b, c \in V$, where by $\overrightarrow{uv}$ or $\overleftarrow{vu}$ we denote an edge directed from u to v [8]. An orientation of a graph G is called P_4*-transitive* if it is transitive when restricted to any P_4 (chordless path on 4 vertices) of G; an orientation of such a path $abcd$ is transitive if and only if the path's edges are oriented in one of the following two ways: $\overrightarrow{ab}$, $\overleftarrow{bc}$ and $\overrightarrow{cd}$, or $\overleftarrow{ab}$, $\overrightarrow{bc}$ and $\overleftarrow{cd}$.

A graph which admits an acyclic transitive orientation is called a *comparability graph* [7,8,9]; Figure 1(a) depicts a comparability graph. A graph is a P_4-*comparability graph* if it admits an acyclic P_4-transitive orientation [11,12]. In light of these definitions, every comparability graph is a P_4-comparability graph. However, the converse is not always true; the graph depicted in Figure 1(b) is a P_4-comparability graph but it is not a comparability graph (it is often referred to as a pyramid). The graph shown in Figure 1(c) is not a P_4-comparability graph. The class of P_4-comparability graphs was introduced by Hoàng and Reed, along

L. Kučera (Ed.): WG 2002, LNCS 2573, pp. 355–366, 2002.

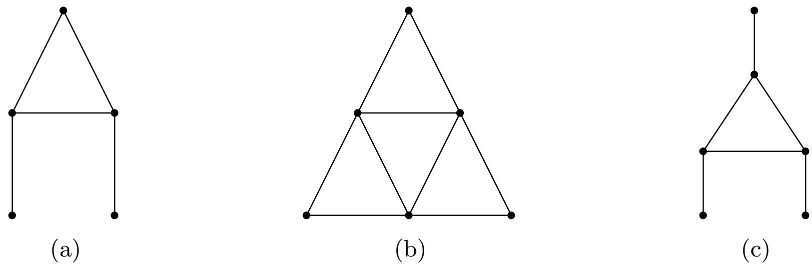

Fig. 1. (a) a comparability graph; (b) a P_4-comparability graph (which is not comparability); (c) a graph which is not P_4-comparability.

with the classes of the P_4-indifference, the P_4-simplicial, and the Raspail graphs, and all four classes were shown to be perfectly orderable [12].

The class of *perfectly orderable* graphs was introduced by Chvátal in the early 1980s [4]; it is a very important class of graphs since a number of problems which are NP-complete in general can be solved in polynomial time on its members [2,8,10]; unfortunately, it is NP-complete to decide whether a graph is perfectly orderable [15]. Chvátal showed that the class of perfectly orderable graphs contains the comparability and the triangulated graphs [4]. It also contains a number of other classes of perfect graphs which are characterized by important algorithmic and structural properties, such as, the classes of 2-threshold, brittle, co-chordal, weak bipolarizable, distance hereditary, Meyniel $\cap$ co-Meyniel, P_4-sparse, etc. [3,8]. Finally, since every perfectly orderable graph is strongly perfect [4], the class of perfectly orderable graphs is a subclass of the well-known class of perfect graphs.

Algorithms for many different problems on almost all the subclasses of perfectly orderable graphs are available in the literature. The comparability graphs in particular have been the focus of much research which culminated into efficient recognition and orientation algorithms [3,8,14]. On the other hand, the P_4-comparability graphs have not received as much attention, despite the fact that the definition of the P_4-comparability graphs is a direct extension of the definition of comparability graphs [6,11,12,17].

Our main objective in this paper is to study the recognition problem on the class of P_4-comparability graphs. This problem along with the problem of producing an acyclic P_4-transitive orientation have been addressed by Hoàng and Reed who described an $O(n^4)$- and an $O(n^5)$-time algorithm respectively for their solution [11,12], where n is the number of vertices of the input graph. Improved results on these problems were provided by Raschle and Simon [17]. Their algorithms work along the same lines, but focus on the P_4-components of the graph; both algorithms run in $O(n + m^2)$, where m is the number of edges of the input graph. Recently, Nikolopoulos and Palios described different $O(n+m^2)$-time algorithms for these problems [16]. Their approach relies on the construction of the P_4-components by means of BFS-trees of the input graph.

In this paper, we present an $O(n\,m)$-time recognition algorithm for P_4-comparability graphs, where n and m are the number of vertices and edges of the input graph. The algorithm computes the P_4s of the input graph G by means of the BFS-trees of the *complement* of G rooted at each of its vertices, without however explicitly computing the complement of G. Instrumental for the algorithm are the observations that the complement of a P_4 is also a P_4 and that for a graph G, the number of vertices in all the levels, but the 0th and the 1st, of the BFS-tree of the complement of G rooted at a vertex v does not exceed the degree of v in G. The proposed recognition algorithm is simple, uses simple data structures and requires $O(n+m)$ space. Along with the result in [16], it leads to an $O(n\,m)$-time algorithm for computing an acyclic P_4-transitive orientation of a P_4-comparability graph.

2 Theoretical Framework

We consider simple non-trivial undirected graphs. Let $G = (V, E)$ be such a graph. A *path* in G is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $v_{i-1}v_i \in E$ for $i = 1, 2, \ldots, k$; we say that this is a path from v_0 to v_k and that its *length* is k. A path is called *simple* if none of its vertices occurs more than once; it is called *trivial* if its length is equal to 0. A simple path $(v_0, v_1, \ldots, v_k)$ is *chordless* if $v_iv_j \notin E$ for any two non-consecutive vertices v_i, v_j in the path. Throughout the paper, the chordless path on n vertices is denoted by P_n. In particular, a chordless path on 4 vertices is denoted by P_4.

Two P_4s are called *adjacent* if they have an edge in common. The transitive closure of the adjacency relation is an equivalence relation on the set of P_4s of a graph G; the subgraphs of G spanned by the edges of the P_4s in the equivalence classes are the *P_4-components* of G. With slight abuse of terminology, we consider that an edge which does not belong to any P_4 belongs to a P_4-component by itself; such a component is called *trivial*. A P_4-component which is not trivial is called *non-trivial*; clearly a non-trivial P_4-component contains at least one P_4.

The definition of a P_4-comparability graph requires that such a graph admits an acyclic P_4-transitive orientation. However, Hoàng and Reed [12] showed that in order to determine whether a graph is a P_4-comparability graph one can restrict one's attention to the P_4-components of the graph. What they proved ([12], Theorem 3.1) can be paraphrased in terms of the P_4-components as follows:

Lemma 1. [12] *Let G be a graph such that each of its P_4-components admits an acyclic P_4-transitive orientation. Then G is a P_4-comparability graph.*

Our recognition algorithm relies on the following important lemma in order to achieve its stated time complexity.

Lemma 2. *Let G be an undirected graph and let $T_{\overline{G}}(v)$ be the BFS-tree of the complement $\overline{G}$ of G rooted at a vertex v. Then, the number of vertices in all the levels of $T_{\overline{G}}(v)$, except for the 0th and the 1st, does not exceed the degree of v in G.*

Proof. Clearly true, since the vertices in all the levels of $T_{\overline{G}}(v)$, except for the 0th and the 1st, are vertices which are not adjacent to v in $\overline{G}$. ∎

3 Recognition of P_4-Comparability Graphs

The algorithm works by constructing and orienting the P_4-components of the given graph, say, G, and then by checking whether they are acyclic (Lemma 1). The P_4-components are constructed as follows: the algorithm considers initially m (partial) P_4-components, one for each edge of G; then, it locates the P_3s of all the P_4s of G, and whenever the edges of such a P_3 belong to different (partial) P_4-components it unions and appropriately orients these P_4-components. Since we are interested in a P_4-transitive orientation of each P_4-component, the edges of such a P_3 need to be oriented either towards their common endpoint or away from it.

As stated earlier, the P_4s of the graph G are computed by means of processing the BFS-trees of the complement $\overline{G}$ of G rooted at each of its vertices. It is important to observe that if $abcd$ is a P_4 then its complement is the P_4 $bdac$ and it belongs to the complement $\overline{G}$ of G. Let us consider the BFS-tree $T_{\overline{G}}(b)$ of $\overline{G}$ rooted at b. Since $bdac$ is a P_4 of $\overline{G}$, the vertices b, d, and a have to belong to the 0th, 1st, and 2nd level of $T_{\overline{G}}(b)$ respectively; the vertex c belongs to the 2nd or 3rd level, but not to the 1st, since c is not adjacent to b in $\overline{G}$. These two cases are shown in Figure 2.

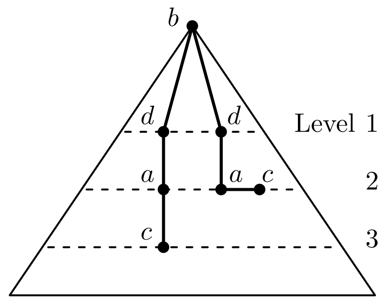

Fig. 2. The two positions of the P_4 $bdac$ in the BFS-tree $T_{\overline{G}}(b)$.

The algorithm is described in more detail below. We consider that the input graph is connected; the case of disconnected graphs is addressed in Section 3.3. Additionally, we assume that initially each edge of G belongs to a P_4-component by itself and is assigned an arbitrary orientation.

P_4-comparability Graph Recognition Algorithm. Input: a connected graph G on n vertices and m edges. *Output:* yes, if G is a P_4-comparability graph; otherwise, no.

1. Initialize to 0 all the entries of an array $M[\,]$ which is of size n;
2. For each vertex v of the graph G, do
   - 2.1 compute the sets L_1, L_2, and L_3 of vertices in the 1st, 2nd, and 3rd level respectively of the BFS-tree of the complement $\overline{G}$ rooted at v;
   - 2.2 partition the set L_2 into subsets of vertices so that two vertices belong to the same subset iff they have (in $\overline{G}$) the same neighbors in L_1;
   - 2.3 for each vertex x in L_2, do

2.3.1 for each vertex w adjacent to x in G, do
$M[w] \leftarrow 1$; $\{$*mark in* $M[\,]$ *the neighbors of* x *in* $G\}$
2.3.2 for each vertex y in L_3 do
if $M[y] = 0$
then $\{xvy$ *is a* P_3 *in a* P_4 *of* $G\}$
If the edges xv and vy belong to the same P_4-component and do not both point towards v or away from it, then the P_4-component cannot admit a P_4-transitive orientation and we conclude that the graph G is not a P_4-comparability graph.
If the edges xv and vy belong to different P_4-components, then we union these components into a single component and if the edges do not both point towards v or away from it, we invert (during the unioning) the orientation of all the edges of the unioned P_4-component with the fewest edges.
2.3.3 for each vertex y in L_2 do
if $M[y] = 0$ and the vertices x and y belong to different partition sets of L_2 (see Step 2.2)
then $\{xvy$ *is a* P_3 *in a* P_4 *of* $G\}$
process the edges xv and vy as in Step 2.3.2;
2.3.4 for each vertex w adjacent to x in G, do
$M[w] \leftarrow 0$; $\{$*clear* $M[\,]\}$

3. After all the vertices have been processed, we apply topological sorting on the directed graph spanned by the directed edges associated with each of the non-trivial P_4-components; if the topological sorting succeeds then the component is acyclic, otherwise there is a directed cycle. If any of the P_4-components contains a directed cycle, then the graph is not a P_4-comparability graph.

For each P_4-component, we maintain a linked list of the records of the edges in the component, and the total number of these edges. Each edge record contains a pointer to the header record of the component to which the edge belongs; in this way, we can determine in constant time the component to which an edge belongs and the component's size. Unioning two P_4-components is done by updating the edge records of the smallest component and by linking them to the edge list of the largest one, which implies that the union operation takes time linear in the size of the smallest component. As mentioned above, in the process of unioning, we may have to invert the orientation in the edge records that we link, if the current orientations are not compatible.

Correctness of the Recognition algorithm. The correctness of the algorithm follows (i) from the fact that in Steps 2.3.2 and 2.3.3 it processes precisely the P_3s participating in P_4s of the input graph G (Lemmata 3 and 4) and that it assigns correct orientations on the edges of these P_3s, (ii) from the correct construction of the P_4-components by unioning partial P_4-components whenever a P_3 is processed whose edges belong to more than one such partial components, and (iii) from Lemma 1 in conjunction with Step 2 of the algorithm.

Note that the initial assignment of 0 to all the entries of the array $M[\,]$ and the clearing of all the set entries at Step 2.3.4 of the algorithm guarantee that the only entries of the array which are set at any iteration are precisely those corresponding to the vertices adjacent in G to the vertex processed at Step 2.3.

Lemma 3. *Every P_3 in a P_4 of the input graph G is considered at Steps 2.3.2 or 2.3.3 of the recognition algorithm.*

Proof. Let $abcd$ be a P_4 of the graph G; we will show that the P_3 abc is considered at Step 2.3.2 or 2.3.3 of the recognition algorithm. Since the algorithm processes each vertex v of G in Step 2 and considers the BFS-tree of $\overline{G}$ rooted at v, it will process b, it will consider the BFS-tree $T_{\overline{G}}(b)$ of $\overline{G}$ rooted at b, and it will compute the sets L_1, L_2, and L_3 of vertices in the 1st, 2nd, and 3rd level of $T_{\overline{G}}(b)$ respectively. Let us consider the two cases of Figure 2. In the first case, the vertices a and c belong to the 2nd and 3rd level of $T_{\overline{G}}(b)$ respectively and they are adjacent in $\overline{G}$. Thus, $a \in L_2$ and $c \in L_3$. Moreover, since a and c are adjacent in $\overline{G}$, then a and c are not adjacent in G. Hence, $M[c] = 0$ when $x = a$ in Step 2.3. Therefore, the P_3 abc is considered in Step 2.3.2 when $x = a$ and $y = c$. In the second case of Figure 2, the vertices a and c belong to the 2nd level of $T_{\overline{G}}(b)$, they are adjacent in $\overline{G}$, and a is adjacent to $d \in L_1$ in $\overline{G}$ whereas c is not. Thus, $a \in L_2$, $c \in L_2$ and $M[c] = 0$ when $x = a$ in Step 2.3, and the vertices a and c belong to different sets in the partition of the vertices in L_2 depending on the vertices in L_1 to which they are adjacent in $\overline{G}$. Therefore, the P_3 abc is considered in Step 2.3.3 when $x = a$ and $y = c$. ∎

Lemma 4. *The sequence (x, v, y) of vertices considered at Steps 2.3.2 and 2.3.3 of the recognition algorithm is a P_3 in a P_4 of the input graph G.*

Proof. Let us first consider Step 2.3.2; in this case, the vertices x and y are in the 2nd and 3rd level of $T_{\overline{G}}(v)$ respectively. Then, the path vp_xxy is a P_4 in $\overline{G}$, where p_x is the parent of x in $T_{\overline{G}}(v)$. This implies that $xvyp_x$ is a P_4 in G and xvy is a P_3 in a P_4 of G. Let us now consider Step 2.3.3. Then, the vertices x and y are in the 2nd level of the BFS-tree $T_{\overline{G}}(v)$ of $\overline{G}$ rooted at v. Moreover, since $M[y] = 0$, then x and y are not adjacent in G, that is, they are adjacent in $\overline{G}$. Finally, the fact that x and y do not belong to the same partition set of L_2, implies that there is a vertex in the 1st level of $T_{\overline{G}}(v)$ which is adjacent to one of them in $\overline{G}$ and not to the other one. Suppose that this vertex is z and that it is adjacent to x; the case where z is adjacent to y and not to x is similar. Then, the path $vzxy$ is a P_4 in $\overline{G}$, which implies that $xvyz$ is a P_4 in G. Clearly, xvy is a P_3 in a P_4 of G. ∎

Before analyzing the complexity of the recognition algorithm, we explain in more detail how Steps 2.1 and 2.2 are carried out.

3.1 Computing the Vertex Sets L_1, L_2, and L_3

The computation of these sets can be done by means of the algorithms of Dahlhaus et al. [5] and Ito and Yokoyama [13] for computing the BFS-tree of the complement of a graph in time linear in the size of the given graph. Both algorithms require the construction of a special representation of the graph. However, we will be using another algorithm which computes the vertices in each level of the BFS-tree of the complement of a graph (i.e., it effectively implements breadth-first search on the complement) in the above stated time complexity. The algorithm is very simple and uses the standard adjacency list representation of a graph. It works by constructing each level L_{i+1} from the previous one, L_i, based on the following lemma.

Lemma 5. *Let G be an undirected graph and let L_i be the set of vertices in the i-th level of a BFS-tree of $\overline{G}$. Consider a vertex w which does not appear in any of the levels from the 0th up to the k-th. Then w is a vertex of the $(k+1)$-st level if and only if there exists at least one vertex of L_k which is not adjacent to w in G.*

Proof. The vertex w is a vertex of the $(k+1)$-st level if and only if it is adjacent in $\overline{G}$ to at least one vertex in L_k. The lemma follows. ∎

We give below the description of the algorithm.

Algorithm for computing the BFS-tree of a vertex v in the complement of a given graph G.

1. Initialize to 0 all the entries of the array $Adj[\,]$ which is of size n;
2. Construct a list L_0 containing a single record associated with the vertex v and a list S containing a record for each of the vertices of G except for v;
3. $i \leftarrow 0$;
   While the list L_i is not empty, do
   3.1 initialize the list L_{i+1} to the empty list;
   3.2 for each vertex u in L_i do
            for each vertex w adjacent to u in G do
                increment $Adj[w]$ by 1;
   3.3 for each vertex s in S do
            if $Adj[s] < |L_i|$
            then remove s from S and add it to the list L_{i+1}
            else $Adj[s] \leftarrow 0$;
   3.4 increment i by 1;

The correctness of the algorithm follows from Lemma 5. Note that the set S contains the vertices which, until the current iteration, have not appeared in any of the computed levels. Moreover, because of Steps 1 and 3.3, the entries of the array $Adj[\,]$ corresponding to the vertices in S are equal to 0 at the beginning of each iteration of the while loop in Step 3. In this way, the test "$Adj[s] < |L_i|$" correctly tests the number of vertices of L_i which are adjacent to the vertex s in G against the size of L_i. Finally, it must be noted that when the while loop of Step 3 terminates, the list S may very well be non-empty; this happens when the graph $\overline{G}$ is disconnected.

Suppose that the input graph G has n vertices and m edges. Clearly, Steps 1 and 2 take $O(n)$ time. In each iteration of the while loop of Step 3, Steps 3.1 and 3.4 take constant time, while Step 3.2 takes $O(\sum_{u \in L_i} d_G(u))$ time, where $d_G(u)$ denotes the degree of the vertex u in G. Step 3.3 takes time linear in the current size of the list S; the elements of S can be partitioned into two sets: (i) the vertices which end up belonging to L_{i+1}, and (ii) the vertices for which the corresponding entries of the array $Adj[\,]$ are equal to $|L_i|$. The number of elements of S in the former set does not exceed $|L_{i+1}|$, while the number of elements in the latter set does not exceed the sum of the degrees (in G) of the vertices in L_i. Thus, Step 3.3 takes $O(|L_{i+1}| + \sum_{u \in L_i} d_G(u))$ time.

Therefore, the time taken by the algorithm is

$$
\begin{aligned}
& O(n) + \sum_i \Big(O(1) + O\Big(|L_{i+1}| + \sum_{u \in L_i} d_G(u) \Big) \Big) \\
&= O(n) + O\Big(\sum_i \Big(1 + |L_{i+1}| \Big) \Big) + O\Big(\sum_i \sum_{u \in L_i} d_G(u) \Big) \\
&= O(n) + O(n) + O(m).
\end{aligned}
$$

The inequalities $\sum_i |L_i| \le n$ and $\sum_i \sum_{u \in L_i} d_G(u) \le \sum_u d_G(u) = 2m$ hold because each vertex belongs to at most one level of the BFS-tree. Moreover, the space needed is $O(n + m)$. Consequently, we have:

Theorem 1. *Let G be an undirected graph on n vertices and m edges, and v be a vertex of G. Then, the above algorithm computes the vertices in the levels of the BFS-tree of the complement $\overline{G}$ of G rooted at v in $O(n + m)$ time and $O(n + m)$ space.*

3.2 Partitioning the Vertices in L_2

It is not difficult to see that the partition of the vertices in L_2 depending on their neighbors in $\overline{G}$ which are in L_1 is identical to the partition of the vertices in L_2 depending on their neighbors in G which are in L_1. This is indeed so, because the subset of vertices in L_1 which are adjacent (in G) to a vertex $x \in L_2$ is $L_1 - N_x$, where N_x is the subset of L_1 containing vertices which are adjacent (in $\overline{G}$) to x. But then, if for two vertices x and y the sets N_x and N_y are equal then so do the sets $L_1 - N_x$ and $L_1 - N_y$, whereas if $N_x \neq N_y$ then $L_1 - N_x \neq L_1 - N_y$. Therefore, in the algorithm we will be working with neighbors in G instead of neighbors in $\overline{G}$.

The algorithm initially considers a single set (list) which contains all the vertices of the set L_2. It then processes each vertex, say, u, of the set L_1 as follows: For each set of the current partition, we check if none, all, or only some of its elements are neighbors of u in G; in the first and second case, the set is not modified, in the third case, it is split into the subset of neighbors of u in G and the subset of non-neighbors of u in G. After all the vertices of L_1 have been processed, the resulting partition is the desired partition.

Algorithm for partitioning the set L_2 in terms of adjacency to elements of the set L_1.

1. Initialize to 0 the entries of the arrays $M[\,]$ and $size[\,]$ which are of size n; insert all the vertices in L_2 in the list $LSet[1]$ and set $size[1] \leftarrow |L_2|$;
   $k \leftarrow 1$; $\{$*k holds the number of sets*$\}$
2. for each vertex u in L_1 do
   2.1 for each vertex w adjacent to u in G do
   $M[w] \leftarrow 1$; $\{$*mark in $M[\,]$ the neighbors of u in G*$\}$
   2.2 $k_0 \leftarrow k$;
   for each list $LSet[i]$, $i = 1, 2, \ldots, k_0$, do
   2.2.1 traverse the list $LSet[i]$ and count the number of its vertices which are neighbors of u in G (use the array $M[\,]$); let ℓ be the number of these vertices;
   2.2.2 if $\ell > 0$ and $\ell < size[i]$
   then $\{$*split $LSet[i]$; create a new set*$\}$
   increment k by 1;
   traverse the list $LSet[i]$ and for each of its vertices w which is a neighbor of u in G (use $M[\,]$), delete w from $LSet[i]$ and insert it in $LSet[k]$;
   $size[k] \leftarrow \ell$;
   decrease $size[i]$ by ℓ;
   2.3 for each vertex w adjacent to u in G do
   $M[w] \leftarrow 0$; $\{$*clear $M[\,]$*$\}$
3. for each list $LSet[i]$, $i = 1, 2, \ldots, k$, do
   traverse the list $LSet[i]$ and for each of its vertices set the corresponding entry of the array $Set[\,]$ equal to i;

Note that thanks to the array $Set[\,]$, checking whether two vertices x and y belong to the same partition set of L_2 reduces to testing whether the entries $Set[x]$ and $Set[y]$ are equal.

The correctness of the algorithm follows from induction on the number of the processed vertices in L_1. At the basis step, when no vertices from the set L_1 have been processed, all the elements of the set L_2 belong to the same set, as desired. Suppose that after processing $i \geq 0$ vertices from L_1, the resulting partition of L_2 is correct with respect to the processed vertices. Let us consider the processing of the next vertex, say, u, from L_1: then, only the sets which contain at least one vertex which is adjacent (in G) to u and at least one vertex which is not adjacent (in G) to u should be split, and indeed these are the only ones that are split; the splitting produces a subset of neighbors of u in G and a subset of non-neighbors of u (Step 2.2.2). Note that because of Steps 1, 2.1, and 2.3, the array $M[\,]$ is clear at the beginning of each iteration of the for loop in Step 2, so that in Step 2.2 the marked entries are precisely those corresponding to the neighbors of the current vertex u in G.

Step 1 of the algorithm clearly takes $O(n)$ time. Steps 2.1 and 2.3 take $O(d_G(u))$ time, where $d_G(u)$ is equal to the degree of u in G. Step 2.2.1 takes

$O(|LSet(i)|)$ time and so does Step 2.2.2, since deleting an entry from and inserting an entry in a list can be done in constant time, and the remaining operations take constant time. Therefore, Step 2.2 takes time linear in the total size of the lists $LSet[i]$ which existed when u started being processed; since the lists contained the vertices in L_2 and none of these lists was empty, we conclude that Step 2.2 takes $O(|L_2|)$ time. Then, Step 2 can be executed in $O(\sum_u (d_G(u) + |L_2|)) = O(m + n|L_2|)$ time. Step 3 takes time linear in the total size of the final lists $LSet[i]$, i.e., $O(|L_2|)$ time. Thus the entire partitioning algorithm takes $O(m + n\,|L_2|)$ time. Since all the initialized lists $LSet[i]$ contain at least one vertex from L_2 and since these lists do not share vertices, then the space complexity is $O(n + |L_2|) = O(n)$.

The results of the paragraph are summarized in the following theorem.

Theorem 2. *Let G be an undirected graph on n vertices and m edges, and let L_1 and L_2 be two disjoint sets of vertices. Then, the above algorithm partitions the vertices in L_2 depending on their neighbors in $\overline{G}$ which belong to L_1 in $O(m + n\,|L_2|)$ time and $O(n)$ space.*

Time and Space Complexity of the Recognition algorithm. Clearly, Step 1 of the algorithm takes $O(n)$ time. In accordance with Theorems 1 and 2, Steps 2.1 and 2.2 take $O(n+m) = O(m)$ and $O(m + n\,|L_2|)$ time respectively in the processing of each one of the vertices of G, while Steps 2.3.1 and 2.3.4 take $O(n)$ time. If we ignore the cost of unioning P_4-components, then Steps 2.3.2 and 2.3.3 require $O(1)$ time per vertex in L_3 and L_2 respectively; recall that testing whether two vertices belong to the same partition set of L_2 takes constant time. If we take into account Lemma 2, we have that $|L_2| \le d_G(v)$ and $|L_3| \le d_G(v)$ where $d_G(u)$ denotes the degree of vertex u in G. Therefore, if we ignore P_4-component unioning, the time complexity of Step 2 of the algorithm is

$$T_2 \;=\; \sum_v \left(O\Big(m + n\, d_G(v)\Big) + \sum_x O\Big(d_G(v) + d_G(x)\Big) \right).$$

If we observe that x belongs to L_2, we conclude that x assumes at most $d_G(v)$ different values. Thus,

$$\begin{aligned}
T_2 &= O\Big(\sum_v m + n \sum_v d_G(v)\Big) + O\Big(\sum_v \sum_x \Big(d_G(v) + d_G(x)\Big)\Big)\\
&= O(nm) + O(nm) + O\Big(\sum_v \Big(d_G^2(v) \;+\; \sum_x d_G(x)\Big)\Big)\\
&= O(nm) + O\Big(\sum_v \; d_G^2(v)\Big) + O\Big(\sum_v \sum_x d_G(x)\Big)\\
&= O(nm)
\end{aligned}$$

since $\sum_v d_G^2(v) \le n \sum_v d_G(v) = O(nm)$ and $\sum_v \sum_x d_G(x) \le \sum_v 2m = 2nm$. Now, the time required for all the P_4-component union operations during the processing of all the vertices is $O(m \log m)$ [1]; there cannot be more than $m-1$ such operations (we start with m P_4-components and we may end up with only one), and each one of them takes time linear in the size of the smallest of the two components that are unioned.

Finally, constructing the directed graph from the edges associated with a non-trivial P_4-component and checking whether it is acyclic takes $O(n + m_i)$, where m_i is the number of edges of the component. Thus, the total time taken by Step 2 is $O\big(\sum_i (n+m_i)\big) = O(nm)$, since there are at most m P_4-components and $\sum_i m_i = m$. Thus, the overall time complexity is $O(n + nm + m \log m + nm) = O(nm)$; note that $\log m \le 2 \log n = O(n)$.

The space complexity is linear in the size of the graph G: the array $M[\,]$ takes linear space, both Steps 2.1 and 2.2 require linear space (Theorems 1 and 2), the set L_1 is represented as a list of $O(n)$ size, the sets L_2 and L_3 are represented as lists having $O(d_G(v))$ size each, and the handling of the P_4-components requires one record per edge and one record per component. Thus, the space required is $O(n + m)$.

Therefore, we have proven the following result:

Theorem 3. *It can be decided whether a connected undirected graph on n vertices and m edges is a P_4-comparability graph in $O(nm)$ time and $O(n + m)$ space.*

3.3 The Case of Disconnected Input Graphs

If the input graph is disconnected, we compute its connected components and work on each one of them as indicated above. In light of Theorem 3 and since the connected components of a graph can be computed in time and space linear in the size of the graph by means of depth-first search [1], we conclude that the overall time complexity is $O(n + m) + \sum_i O(n_i\, m_i) = O(n \sum m_i) = O(nm)$ and the space is $O(n + m) + \sum_i O(n_i + m_i) = O(n + m)$ since $\sum_i n_i = n$ and $\sum_i m_i = m$.

Theorem 4. *It can be decided whether an undirected graph on n vertices and m edges is a P_4-comparability graph in $O(n\,m)$ time and $O(n + m)$ space.*

4 Concluding Remarks

In this paper, we presented an $O(n\,m)$-time and linear space algorithm to recognize whether a graph on n vertices and m edges is a P_4-comparability graph. The algorithm exhibits the currently best time and space complexity to the best of our knowledge, and is simple enough to be easily used in practice. Along with the work of [16], it leads to an $O(n\,m)$-time algorithm for computing an acyclic P_4-transitive orientation of a P_4-comparability graph, thus improving the upper bound on the time complexity for this problem as well. We also described a simple algorithm to compute the levels of the BFS-tree of the complement $\overline{G}$ of a graph G in time and space linear in the size of G.

The obvious open question is whether the P_4-comparability graphs can be recognized and/or oriented in $o(n\,m)$ time. Moreover, it is worth investigating whether taking advantage of properties of the complement of the input graph can help establish improved algorithmic solutions for other problems as well; note that breadth-first and depth-first search on the complement of a graph can be executed in time linear in the size of the graph.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. S.R. Arikati and U.N. Peled, A polynomial algorithm for the parity path problem on perfectly orderable graphs, *Discrete Appl. Math.* **65**, 5–20, 1996.
3. A. Brandstädt, V.B. Lê, and J.P. Spinrad, *Graph Classes: A Survey*, Monographs on Discrete Mathematics and Applications 3, SIAM, 1999.
4. V. Chvátal, Perfectly ordered graphs, *Annals of Discrete Math.* **21**, 63–65, 1984.
5. E. Dahlhaus, J. Gustedt, and R.M. McConnell, Efficient and practical modular decomposition, *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)*, 26–35, 1997.
6. C.M.H. de Figueiredo, J. Gimbel, C.P. Mello, and J.L. Szwarcfiter, Even and odd pairs in comparability and in P_4-comparability graphs, *Discrete Appl. Math.* **91**, 293–297, 1999.
7. P.C. Gilmore and A.J. Hoffman, A characterization of comparability graphs and of interval graphs, *Canad. J. Math.* **16**, 539–548, 1964.
8. M.C. Golumbic, *Algorithmic graph theory and perfect graphs*, Academic Press, Inc., New York, 1980.
9. M.C. Golumbic, D. Rotem, and J. Urrutia, Comparability graphs and intersection graphs, *Discrete Math.* **43**, 37–46, 1983.
10. C.T. Hoàng, Efficient algorithms for minimum weighted colouring of some classes of perfect graphs, *Discrete Appl. Math.* **55**, 133–143, 1994.
11. C.T. Hoàng and B.A. Reed, Some classes of perfectly orderable graphs, *J. Graph Theory* **13**, 445–463, 1989.
12. C.T. Hoàng and B.A. Reed, P_4-comparability graphs, *Discrete Math.* **74**, 173–200, 1989.
13. H. Ito and M. Yokoyama, Linear time algorithms for graph search and connectivity determination on complement graphs, *Inform. Process. Letters* **66**, 209–213, 1998.
14. R.M. McConnell and J. Spinrad, Linear-time transitive orientation, *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)*, 19–25, 1997.
15. M. Middendorf and F. Pfeiffer, On the complexity of recognizing perfectly orderable graphs, *Discrete Math.* **80**, 327–333, 1990.
16. S.D. Nikolopoulos and L. Palios, Recognition and orientation algorithms for P_4-comparability graphs, *Proc. 12th Symp. on Algorithms and Computation (ISAAC'01)*, 320–331, 2001.
17. T. Raschle and K. Simon, On the P_4-components of graphs, *Discrete Appl. Math.* **100**, 215–235, 2000.