ON THE RECOGNITION OF
$P_4$–COMPARABILITY GRAPHS

S. Nikolopoulos and L. Palios

6–2002

Preprint, no 6–02/2002

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece
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
e-mail: {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 Hoang 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 Roschle 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 $O(nm)$-time, $O(n+m)$-space algorithm for the recognition of $P_4$-comparability graphs. The algorithm computes the $P_4$s of the input graph $G$ by means of the BFS-trees of the complement of the graph rooted at each of its vertices. The key to achieve the stated time complexity lies in the observation that for a graph $G$, the number of vertices in all the levels, except for 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 $G$; thus, considering pairs of vertices located in these levels takes $O(\sum_v d_G(v)) = O(nm)$, where $d_G(v)$ denotes the degree of $v$ in $G$. Our algorithm is simple, uses simple data structures, and leads to an $O(nm)$-time 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

Let $G = (V,E)$ be a simple non-trivial undirected graph. An orientation of the graph $G$ 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 $\overrightarrow{vu}$ 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: $\overrightarrow{ab}$, $\overrightarrow{bc}$ and $\overrightarrow{cd}$, or $\overrightarrow{ab}$, $\overrightarrow{bc}$ and $\overrightarrow{cd}$.

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 \cite{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 the $P_4$-comparability graphs was introduced by Hoàng and Reed, along 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 \cite{12}.

In the early 1980s, Chvátal introduced the class of \textit{perfectly orderable} graphs \cite{4}; see also \cite{11, 15, 17}. These are the graphs for which there exists a \textit{perfect order} on the set of their vertices. An order on the vertex set of a graph $G$ is called \textit{perfect} if for each subgraph $H$ of $G$, the greedy algorithm (sometimes called the \textit{first-fit} algorithm) computes an optimal coloring of $H$ by processing the vertices of $G$ in that order. Equivalently, Chvátal \cite{4} showed that a graph is perfectly orderable if and only if there exists an acyclic orientation such that no $P_4$ of the graph has an obstruction. The fact that the $P_4$-comparability graphs are perfectly orderable follows immediately from that since they are defined to admit acyclic orientations that do not contain obstructions.

The class of perfectly orderable graphs is very important since a number of problems which are NP-complete in general can be solved in polynomial time on its members \cite{2, 8, 19}; unfortunately, it is NP-complete to decide whether a graph admits a perfect order or, equivalently, an acyclic obstruction-free orientation \cite{15}. Chvátal showed that the class of perfectly orderable graphs contains the comparability and the triangulated graphs \cite{4}. It also contains a number of other classes of perfect graphs which are characterized by important algorithmic and structural properties; we mention the classes of 2-threshold, brittle, co-chordal, weak bipolarizable, distance hereditary, Meyniel $\cap$ co-Meyniel, $P_4$-sparse, etc. \cite{3, 8}. Finally, since every perfectly orderable graph is strongly perfect \cite{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 \cite{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 \cite{6, 11, 12, 18}.

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 polynomial time algorithms for their solution \cite{11, 12}. The algorithms are based on detecting whether
the input graph $G$ contains a “homogeneous set” or a “good partition” and recursively solve the same problem on the graph that results from the input graph after contraction of one or two vertex sets into a single vertex each. The recognition and the orientation algorithms require $O(n^3)$ and $O(n^2)$ time respectively, where $n$ is the number of vertices of $G$. Improved results on these problems were provided by Raschle and Simon [18]. Their algorithms work along the same lines, but they focus on the $P_4$-components of the graph. In particular, for a non-trivial $P_4$-component $C$ of the input graph $G$, they compute the set $R$ of vertices adjacent to some but not all the vertices of $C$, depending on whether $R$ is empty or not, they contract $C$ into one or two (non-adjacent) vertices and they recursively solve the problem on the resulting graph. The time complexity of their algorithms for either problem is $O(n + m^2)$, where $m$ is the number of edges of $G$, as it is dominated by the time to compute the $P_4$-components of $G$. 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 $O(nm)$-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 $P_4$s of the input graph $G$ by means of the BFS-trees of the complement of the graph rooted at each of its vertices. The key to achieve the stated time complexity lies in the observation 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 $G$; thus, considering pairs of vertices located in these levels takes $O(\sum_v d_G^2(v)) = O(nm)$, where $d_G(v)$ denotes the degree of $v$ in $G$. We believe that this important observation, in combination with algorithms which perform breadth-first and depth-first search on the complement of a graph in time linear in the size of the given graph, will result in improved algorithmic solutions for other problems as well. 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 $O(nm)$-time algorithm for computing an acyclic $P_4$-transitive orientation of a $P_4$-comparability graph.

The paper is structured as follows. In Section 2 we review the terminology and we prove a key lemma for our algorithm. We describe and analyze the recognition algorithm in Section 3, and we conclude with Section 4 which summarizes our results and presents some open questions.

2. Theoretical Framework

Let $G = (V, E)$ be a simple non-trivial connected graph on $n$ vertices and $m$ edges. 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 undirected or directed depending on whether $G$ is an undirected or a directed graph. 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 $P_4$.

Let abcd be a $P_4$ of a graph $G$. The vertices $b$ and $c$ are called midpoints and the vertices $a$ and $d$ endpoints of the $P_4$ abcd. The edge connecting the midpoints of a $P_4$ is called the rib.
the other two edges (which are incident to the endpoints) are called the wings. For example, the edge \( bc \) is the rib and the edges \( ab \) and \( cd \) are the wings of the \( P_4 \) \( abcd \). Two \( P_4 \)s 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 \( P_4 \)s of a graph \( G \); the subgraphs of \( G \) spanned by the edges of the \( P_4 \)s 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 \( P_4 \). If the set of midpoints and the set of endpoints of the \( P_4 \)s of a non-trivial \( P_4 \)-component \( C \) define a partition of the vertex set \( V(C) \), then the \( P_4 \)-component \( C \) is called separable.

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. In particular, what they proved ([12], Theorem 3.1) can be paraphrased in terms of the \( P_4 \)-components as follows:

**Lemma 2.1.** ([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.

It must be noted that although determining that each of the \( P_4 \)-components of a graph admits an acyclic \( P_4 \)-transitive orientation suffices to establish that the graph is \( P_4 \)-comparability, the directed graph produced by placing the oriented \( P_4 \)-components together may contain cycles.

Given a non-trivial \( P_4 \)-component \( C \) of a graph \( G = (V, E) \), the set of vertices \( V - V(C) \) can be partitioned into three sets:

(i) \( R \) contains the vertices of \( V - V(C) \) which are adjacent to some (but not all) of the vertices in \( V(C) \).

(ii) \( P \) contains the vertices of \( V - V(C) \) which are adjacent to all the vertices in \( V(C) \), and

(iii) \( Q \) contains the vertices of \( V - V(C) \) which are not adjacent to any of the vertices in \( V(C) \).

The adjacency relation is considered in terms of the given graph \( G \).

In [18], Raschle and Simon showed that, given a non-trivial \( P_4 \)-component \( C \) and a vertex \( v \notin V(C) \), if \( v \) is adjacent to the midpoints of a \( P_4 \) of \( C \) and is not adjacent to its endpoints, then \( v \) does so with respect to every \( P_4 \) in \( C \) (that is, \( v \) is adjacent to the midpoints and not adjacent to the endpoints of every \( P_4 \) in \( C \)). This implies that any vertex of \( G \), which does not belong to \( C \) and is adjacent to at least one but not all the vertices in \( V(C) \), is adjacent to the midpoints of all the \( P_4 \)s in \( C \). Based on that, Raschle and Simon showed that in terms of the \( P_4 \)-component \( C \), the sets \( R \), \( P \), and \( Q \) exhibit the adjacencies shown in Figure 2. The set \( V_1 \) is the set of the midpoints of all the \( P_4 \)s in \( C \), whereas the set \( V_2 \) is the set of endpoints; the dashed segments between \( R \) and \( P \) and between \( P \) and \( Q \) indicate that there may be edges between pairs of vertices in the corresponding sets.

Our algorithm relies on the following crucial lemma in order to achieve its stated time complexity.
Lemma 2.2. Let \( G \) be a simple undirected graph and let \( T_{\overline{G}}(v) \) be the BFS-tree of the complement \( \overline{G} \) 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 \( \overline{G} \).

We believe that this is a very important observation and we expect that it will help establish improved complexity bounds in other problems as well.

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 2.1). 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 \( P_4 \)s of all the \( P_4 \)s 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.

The key idea of the algorithm in order to achieve its stated time complexity is the computation of the \( P_4 \)s by means of processing the BFS-trees of the complement \( \overline{G} \) of the input graph rooted at each of its vertices.

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.

Recognition Algorithm.

Input: a simple 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 \( G \)) 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

\[ M[w] \leftarrow 1; \quad \{ \text{mark in } M[\cdot] \text{ the neighbors of } x \text{ in } \overline{G} \} \]

2.3.2 for each vertex \( y \) in \( L_3 \) do

if \( M[y] = 0 \)

then \( \{ xy \text{ is a } P_3 \text{ of a } P_4 \text{ in } G \} \)

If the edges \( xy \) 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 \( xy \) 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 \( L_2 \) (Step 2.2)

then \( \{ xy \text{ is a } P_3 \text{ of a } P_4 \text{ in } G \} \)

process the edges \( xy \) and \( vy \) as in Step 2.3.2;

2.3.4 for each vertex \( w \) adjacent to \( x \) in \( G \), do

\[ M[w] \leftarrow 0; \quad \{ \text{clear } M[\cdot] \} \]

3. After all the vertices have been processed, we check whether the resulting non-trivial \( P_4 \)-components contain directed cycles. This is done by applying topological sorting on the directed graph spanned by the directed edges associated with each of the \( P_4 \)-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 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 from (i) the fact that in Steps 2.3.2 and 2.3.3 it processes the $P_3$s of all the $P_4$s of the input graph $G$ (Lemmas 3.1 and 3.2) and that it assigns correct orientations on the edges of these $P_3$s, (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 2.1 in conjunction with Step 2 of the algorithm.

Note that the initial assignment of 0 to all the entries of the array $M[\cdot]$ 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 3.1. Every $P_3$ of 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. Because $abcd$ is a $P_4$ then its complement is the $P_4$ $bdac$ and it belongs to the complement $G\overline{\cdot}$ of $G$. Let us consider the BFS-tree $T_{\overline{G}}(b)$ of $G\overline{\cdot}$ rooted at $b$. Since $bdac$ is a $P_4$ of $G\overline{\cdot}$, 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 level since $c$ is not adjacent to $b$ in $G$. These two cases are shown in Figure 3.

Since the algorithm processes each vertex $v$ of $G$ in Step 2 and considers the BFS-tree of $G\overline{\cdot}$ rooted at $v$, it will process $b$, it will consider the BFS-tree $T_{\overline{G}}(b)$ of $G\overline{\cdot}$ 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. In the first case of Figure 3, the vertices $a$ and $c$ belong to the 2nd and 3rd level of $T_{\overline{G}}(b)$ respectively and they are adjacent in $G$. Thus, $a \in L_2$ and $c \in L_3$. Moreover, since $a$ and $c$ are adjacent in $G\overline{\cdot}$, 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 3, the vertices $a$ and $c$ belong to the 2nd level of $T_{\overline{G}}(b)$, they are adjacent in $G\overline{\cdot}$, and $a$ is adjacent to $d \in L_1$ in $G\overline{\cdot}$ 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 $G\overline{\cdot}$. Therefore, the $P_3$ $abc$ is considered in Step 2.3.3 when $x = a$ and $y = c$.

Lemma 3.2. The sequence $(x, y, y)$ of vertices considered at Steps 2.3.2 and 2.3.3 of the recognition algorithm is a $P_3$ of a $P_4$ of the input graph $G$.

Proof: Let us first consider Step 2.3.2. Then, the vertices $x$ and $y$ are in the 2nd and 3rd level of $T_{\overline{G}}(v)$ respectively. Then, the path $v_{n_{x}}xy$ is a $P_3$ in $G\overline{\cdot}$, where $p_{x}$ is the parent of $x$ in $T_{\overline{G}}(v)$. This implies that $xv_{n_{x}}y$ is a $P_4$ in $G$ and $xv_{n_{x}}y$ is a $P_3$ of a $P_4$ in $G$.

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 $G\overline{\cdot}$ rooted at $v$. Moreover, since $M[y] = 0$, then $x$ and $y$ are not adjacent in $G$, that is, they are adjacent in $G\overline{\cdot}$. 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 $T_G(v)$ 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 $xyz$ is a $P_4$ in $\overline{G}$, which implies that $xvzy$ is a $P_4$ in $G$. Clearly, $xvy$ is a $P_3$ of a $P_4$ in $G$.

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

**Proof:** The vertex $w$ is a vertex of the $(i+1)$-st level if and only if it is adjacent in $\overline{G}$ to at least one vertex in $L_i$. 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. Set $i$ to 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 $Adj[s] ← 0$;

   3.4 increment $i$ by 1;
The correctness of the algorithm follows from Lemma 3.3. 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 $\text{Adj}[\cdot]$ 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 "$\text{Adj}[s] < |L_i|$" 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 $\text{Adj}[\cdot]$ 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

$$O(n) + \sum_i \left( O(1) + O\left( |L_{i+1}| + \sum_{u \in L_i} d_G(u) \right) \right)$$

$$= O(n) + O\left( \sum_i (1 + |L_{i+1}|) \right) + O\left( \sum_i \sum_{u \in L_i} d_G(u) \right)$$

$$= O(n) + O(n) + O(m).$$

The inequalities $\sum_i |L_i| \leq n$ and $\sum_i \sum_{u \in L_i} d_G(u) \leq \sum_u d_G(u) = 2m$ 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 3.1.** Let $G$ be a simple 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, instead of working with neighbors of the vertices in $\overline{G}$, we will be working with neighbors in $G$, and this will lead to a partitioning algorithm with time complexity linear in the size of the graph $G$.

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 $M[\cdot]$ and $size[\cdot]$ which are of size $n$; insert all the vertices in $L_2$ in the list $LSet[1]$ and set $size[1] \leftarrow |L_2|$; set $k \leftarrow 1$; \{ $k$ 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 $M[\cdot]$ the neighbors of $u$ in $G$ \}

   2.2 set $k_0 \leftarrow k$;
     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 $M[\cdot]$); let $\ell$ 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 $M[\cdot]$), delete $w$ from $LSet[i]$ and insert it in $LSet[k]$;
       $size[k] \leftarrow \ell$;
       decrease $size[i]$ by $\ell$;

   2.3 for each vertex $w$ adjacent to $u$ in $G$ do
       
       $M[w] \leftarrow 0$; \{ clear $M[\cdot]$ \}

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 $Set[\cdot]$ equal to $i$;

Note that thanks to the array $Set[\cdot]$, 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 $M[\cdot]$ 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 $G$. 

10
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 takes constant time (for the deletion, we need to keep a pointer to the previous record during the traversal), and the remaining operations take constant time. Therefore, Step 2.2 takes time linear in the total size of 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 3.2.** Let $G$ be a simple 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 3.1 and 3.2, 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 $L_1$ 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 2.2, we have that $|L_2| \leq d_G(v)$ and $|L_3| \leq d_G(v)$ 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

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

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

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

$$= O(nm) + \sum_v \sum_x d_G(v) + O\left( \sum_v \sum_x d_G(x) \right)$$

$$= O(nm) + \sum_v d_G^2(v) + O\left( \sum_v \sum_x d_G(x) \right)$$

$$= O(nm)$$

since $\sum_v d_G^2(v) \leq n \sum_v d_G(v) = O(nm)$ and $\sum_v \sum_x d_G(x) \leq \sum_v 2m = 2nm$. 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 \(O(\sum_i (n + m_i)) = O(nm)\), 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 \(\log m \leq 2 \log n = O(n)\).

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 3.1 and 3.2), 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 proved the following result:

**Theorem 3.3.** It can be decided whether a simple 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. In this case, we compute the connected components of the graph and work on each one of them as indicated above. In light of Theorem 3.3 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 \(O(n + m) + \sum_i O(n_i m_i) = O(n \sum_i m_i) = O(nm)\) 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 3.4.** It can be decided whether a simple undirected graph on \(n\) vertices and \(m\) edges is a \(P_4\)-comparability graph in \(O(nm)\) time and \(O(n + m)\) space.

4. Concluding Remarks

In this paper, we presented an \(O(nm)\)-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 \(O(nm)\)-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 \(o(nm)\) time.
References


