Linear Time LexDFS on Cocomparability Graphs

Ekkehard Köhler[1] and Lalla Mouatadid[2]

[1] Brandenburg University of Technology, 03044 Cottbus, Germany
`ekoehler@math.tu-cottbus.de`
[2] University of Toronto, Toronto ON M5S 2J7, Canada
`lalla@cs.toronto.edu`

Abstract. Lexicographic depth first search (LexDFS) is a graph search protocol which has already proved to be a powerful tool on cocomparability graphs. Cocomparability graphs have been well studied by investigating their complements (comparability graphs) and their corresponding posets. Recently however LexDFS has led to a number of elegant polynomial and near linear time algorithms on cocomparability graphs when used as a preprocessing step [3, 4, 15]. The nonlinear running time of some of these results is a consequence of the complexity of a LexDFS preprocessing step. We present the first linear time algorithm to compute a LexDFS cocomparability ordering, therefore answering a problem raised in [3] and helping achieve the first linear time algorithms for the minimum path cover problem, and thus the Hamilton path problem, the maximum independent set problem, the minimum clique cover. Our algorithm was also used to compute a maximum cardinality matching on cocomparability. We end this paper with a linear time algorithm to compute a LexDFS cocomparability ordering when the input graph is a comparability graph. This later result can be used to certify, in linear time, a maximum clique on a cocomparability graph by outputting a minimum colouring of the graph, as is done in [4].

Keywords: lexicographic depth first search, cocomparability graphs, graph searching, posets, Hamiltonian path, linear time algorithm.

1 Introduction

Graph searching is a very useful and widely used tool that gave rise to a number of efficient and easily implementable algorithms. Lexicographic breadth first search (LexBFS) for instance, is a well known graph search protocol which has led to elegant algorithms on various graph families, as illustrated in [1, 9]. More recently another graph search protocol, lexicographic depth first search (LexDFS), was introduced and has already proved to be a powerful tool on cocomparability graphs [3, 4, 15]. Indeed, since LexDFS was introduced [5], many problems, such as computing a maximum cardinality independent set, a minimum clique cover or a minimum path cover, now have near-linear time solutions for cocomparability graphs. These successful approaches share a strategy: They start with

creating a so-called cocomparability ordering σ of the graph, and preprocessing σ with a LexDFS sweep and then basically extend, or slightly modify, the linear time algorithms that work for interval graphs (a subfamily of cocomparability graphs). The nonlinear running time of the algorithms on cocomparability graphs is forced by the non-linearity of LexDFS.

In this paper, we present the first linear time algorithm to compute a LexDFS cocomparability ordering. Therefore, as immediate corollaries, we now have the first linear time algorithm to compute a minimum path cover, and thus a Hamilton path if one exists, and the first linear time algorithm to compute a maximum independent set and a minimum clique cover for cocomparability graphs. Our algorithm was also used more recently to give the first linear time algorithm to compute a maximum cardinality matching on cocomparability graphs [16]. We will also show how to specifically compute a LexDFS$^+$ ordering in linear time. LexDFS$^+$ is a variant of LexDFS that is needed in [3, 4, 15, 16].

Cocomparability graphs are a family of perfect graphs whose complements, comparability graphs, admit a transitive orientation of the edges. That is, for every three vertices x, y, z, if the edges xy, yz are oriented $x \to y \to z$ then $xz \in E$ and $x \to z$. Cocomparability graphs and partially ordered sets, or *posets*, are closely related. In Section 2, we explain this relationship and how algorithms on cocomparability graphs immediately translate into algorithms on posets. Cocomparability graphs have been well studied by investigating their complements (comparability graphs) and their corresponding posets [8]. Recently however, there has been a growing motivation to exploit the structure of cocomparability graphs in order to design algorithms that do not require the computation of the complement or the poset [15, 13, 12, 3, 4, 16, 10]. For this approach the use of LexDFS has proven to be quite successful [3, 4, 15].

A key point in our algorithm for computing a LexDFS ordering is a partition refinement approach of the layers of a corresponding poset of the cocomparability graph, *without* computing the poset itself. This refinement is *in situ* and performed *backwards*, as opposed to the well known forward partition refinement used to compute a LexBFS ordering [9]. We discuss the technique of partition refinement in more details in Section 2. The paper is organized as follows: Section 2 gives the necessary background and relevant definitions. In Section 3, we present the LexDFS algorithm, and in 4, we prove its correctness. Implementation details are presented in Section 5. In Section 6, we show how to modify the algorithm to compute a LexDFS$^+$ ordering in linear time, as well as a LexDFS ordering of $\overline{G}$, when G is a *comparability graph*, in time linear in size of $\overline{G}$, the input graph. We conclude in Section 7 with future directions and some implications of this line of work.

2 Background

We assume the reader to be familiar with basic graph notation. All the graphs considered in this paper are finite, simple, and undirected, unless explicitly stated otherwise. For a vertex v, $N(v) = \{u | uv \in E\}$; and we say v is *simplicial* if $N(v)$

is a clique. An *ordering* σ of V is a bijection $\sigma\colon [1...n] \to V$. Given an ordering $\sigma = v_1, v_2, ..., v_n$, we write $v_i \prec_\sigma v_j$ if v_i appears before v_j in σ, i.e., $i < j$; and $N^+(v_i) = \{v_j | v_i v_j \in E \text{ and } i < j\}$; we denote by $\sigma^- = v_n, v_{n-1}, ..., v_2, v_1$ the reverse ordering of σ. Given a cocomparability graph $G(V, E)$, an ordering σ is a *cocomparability ordering* (or an umbrella free ordering) of G if for any triple $a \prec_\sigma b \prec_\sigma c$ where $ac \in E$, we either have $ab \in E$ or $bc \in E$, or both [13]. If neither ab and bc are edges, we say that the umbrella ac *flies* over b. Note that the cocomparability ordering is just the equivalent to transitivity in the complement. In [14], McConnell and Spinrad presented an algorithm that computes a cocomparability ordering in $\mathcal{O}(m + n)$ time, where $m = |E|$ and $n = |V|$.

A poset $P(V, \prec)$ is an irreflexive, antisymmetric and transitive relation on the set V. We say that two elements $a, b \in V$ are comparable if $a \prec b$ or $b \prec a$, otherwise they are incomparable. A linear extension $\mathcal{L}$ of P is a total ordering of V which respects the order imposed by $\prec$. As already mentioned, posets and cocomparability graphs are closely related. In fact, if $G(V, E)$ is a comparability graph, then G together with a transitive orientation of E can equivalently be represented by a poset $P(V, \prec)$ where $uv \in E$ if and only if u and v are comparable in P. This implies that σ, a cocomparability ordering of $\overline{G}$ is a linear extension of P. Notice that for every poset $P(V, \prec)$, there exists a unique comparability graph $G(V, E)$, and thus a unique cocomparability graph. Conversely, every transitive orientation of a comparability graph, and thus every cocomparability ordering in the complement, is a linear extension of *a* poset P.

A graph search is a mechanism for visiting vertices of a given graph in a certain manner. We say that two or more vertices are *tied* if at a given step of the graph search, these vertices are all eligible to be visited next. In 2008, Corneil and Krueger [5] introduced LexDFS, a graph search that extends depth first search by assigning lexicographic labels to the vertices in order to break ties. Algorithm 1 is the generic LexDFS algorithm, as presented in [5], and Fig. 1 is a step by step illustrative example.

Algorithm 1 LexDFS

H

Input: a graph $G(V, E)$ and a start vertex s
Output: a LexDFS ordering σ of V
1: assign the label ϵ to all vertices
2: $label(s) \leftarrow \{0\}$
3: **for** $i \leftarrow 1$ to n **do**
4:     pick an unnumbered vertex v with lexicographically largest label
5:     $\sigma(i) \leftarrow v$ ▷ v is assigned the number i
6:     **foreach** unnumbered vertex w adjacent to v **do**
7:         prepend i to $label(w)$
8:     **end for**
9: **end for**

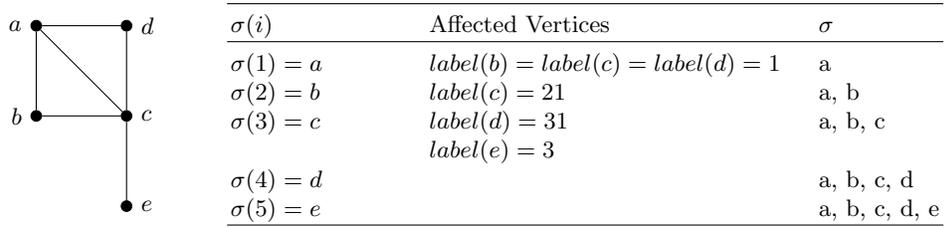| $\sigma(i)$	Affected Vertices	σ
$\sigma(1) = a$	$label(b) = label(c) = label(d) = 1$	a
$\sigma(2) = b$	$label(c) = 21$	a, b
$\sigma(3) = c$	$label(d) = 31$	a, b, c
	$label(e) = 3$	
$\sigma(4) = d$		a, b, c, d
$\sigma(5) = e$		a, b, c, d, e

Fig. 1. $G(V, E)$ a cocomparability graph, and a step by step computation of a LexDFS ordering of G starting at vertex a, $\sigma = a, b, c, d, e$.

LexDFS admits the following vertex ordering characterization, known as the *4 Point Condition*:

Theorem 1 (The 4 Point Condition). *[5] σ is a LexDFS ordering of a graph $G(V, E)$ if and only if for every triple $a \prec_\sigma b \prec_\sigma c$ where $ac \in E, ab \notin E$, there must exists a vertex d such that $a \prec_\sigma d \prec_\sigma b$ and $db \in E, dc \notin E$.*

LexDFS$^+$ is the LexDFS variant with the additional '*rightmost*' tie breaking rule. That is, given a vertex ordering σ of G, the ordering $\tau = $ LexDFS$^+(\sigma)$ is a LexDFS of G where ties between eligible vertices are broken by choosing the rightmost vertex in σ. Therefore, by definition, LexDFS$^+$ always starts by the rightmost vertex in σ. For an example, look at the graph in Fig. 1. If we compute $\tau =$LexDFS$^+(\sigma)$, for this example, we have to start with vertex e, i.e. $\tau(1) = e$; obviously, $\tau(2) = c$. For $\tau(3)$, in a regular LexDFS, a, b and d are tied. However, for LexDFS$^+(\sigma)$, $\tau(3) = d$, since d is rightmost in σ among a, b, and d. It was shown in [3] that if $G(V, E)$ is a cocomparability graph, and σ a cocomparability ordering of G, then the LexDFS ordering $\tau = LexDFS^+(\sigma)$ is also a cocomparability ordering of G. It is easy to see that if σ is a LexDFS cocomparability ordering, then for any triple $a \prec_\sigma b \prec_\sigma c$ where c is a non-simplicial vertex and $ab \notin E, ac, bc \in E$, there exists a vertex d such that $a \prec_\sigma d \prec_\sigma b$ and $ad, db \in E$ and $dc \notin E$ [3]. Indeed the edge ad destroys the umbrella ac over d.

As was already pointed out in the introduction, the key idea of our algorithm to determine a LexDFS of a cocomparability graph is the backward in situ partition refinement of the layers of a corresponding poset, without computing the poset itself. Given a set S, we call $\mathcal{P} = (P_1, P_2, ..., P_k)$ a partition of S if for all $P_i, P_j, i \neq j$, $P_i \cap P_j = \emptyset$ and $\bigcup_{i=1}^{k} P_i = S$. Given a set $T \subseteq S$, we say that T refines $\mathcal{P}$ when every partition class $P_i \in \mathcal{P}$ is replaced with subpartition classes $A_i = P_i \cap T$ and $B_i = P_i \backslash A_i$. This technique, known as *partition refinement* has led to a simple and elegant implementation of LexBFS in linear time [9]. The LexBFS partition refinement algorithm is as follows: Initially $\mathcal{P} = (V)$; select a start vertex s where $N(s)$ refines $\mathcal{P}$ by placing $A = V \cap N(s)$ before $B = V \backslash A$. The vertex whose neighbourhood is used to refine the partition classes is called a *pivot*. Pick the next pivot v amongst vertices in A; and use $N(v)$ for refining A then B and maintaining the order of the partition classes created so far:

$(A \cap N(v), A \backslash N(v), B \cap N(v), B \backslash N(v))$ in this order. This process is repeated until all partition classes have been refined.

This refinement can be seen as a *forward* refinement in the sense that pivots are selected left to right, i.e., from the A's sets then the B's set, and the refinement is *in situ*, meaning the A_i's always precede the B_i's. In other words, pivots do not reorder the already created subpartitions of P. That is , if $N(u)$ was used to refine P to $A = P \cap N(u)$ and $B = P \backslash A$, and v is the next pivot then $N(v)$ is used to refine A first to $A \cap N(v)$ followed by $A \backslash N(v)$, next $N(v)$ refines B to $B \cap N(v)$ followed by $B \backslash N(v)$, and the subpartitions of A always precede the subpartitions of B. This in situ refinement results in a linear time implementation for LexBFS. Also for LexDFS, one can define a partition refinement scheme, but this partition refinement is not in situ and can be seen as a *backward* refinement in the sense that pivots are selected right to left. Consider a pivot v; due to the depth first search character of LexDFS, v has to pull *all* its neighbours to the front, i.e., $A \cap N(v)$ followed by $B \cap N(v)$ both precede $A \backslash N(v)$ followed by $B \backslash N(v)$. This sorting of the partition classes is an obstacle to a linear time implementation for LexDFS. In fact, to the best of our knowledge, there is no linear time implementation of LexDFS. The best known algorithm takes $\mathcal{O}(\min(n^2, n + m \log \log n))$ time and uses the above explained non in situ partition refinement together with van Emde Boas trees [17].

3   The Algorithm

Before presenting our algorithm in detail we first give an overview. Let $G(V, E)$ be a cocomparability graph. We first compute a cocomparability ordering σ using the algorithm in [14]. Then, we assign a label, denoted $\#(v)$, to each vertex v in σ. We use these labels to compute, for every vertex v, the length of a largest chain succeeding v in the corresponding poset of the complement of G. Roughly speaking, we then partition V by iteratively placing vertices with smallest label into the same partition set. In a comparability graph one could finds these sets by iteratively removing the set of maximal elements of the poset. Since we work on the complement, this has to be done using only edges of the cocomparability graph, i.e. non-edges of the comparability graph. Once all the vertices have their initial labeling, we iteratively create a partition $\mathcal{P}$ of V wherein each step i, the partition class P_i consists of the vertices of minimum label value. When a vertex v is added to a partition class P_i, we say that v has been *visited*.

Since σ is a cocomparability ordering, and thus a linear extension of a poset, the P_1 vertices are exactly the elements in the linear extension with no upper cover. Therefore they are just the maximal elements of the partial order defined by σ in $\overline{G}$. Similarly, when all the P_1 have been visited, i.e., 'removed', P_2 is just the set of maximal elements in the partial order of $\overline{G} \backslash P_1$, and so on. Creating the partition classes is indeed equivalent to removing the maximal elements of a poset corresponding to $\overline{G}$ one layer at a time.

The final step of the algorithm is the partition refinement where we refine each partition class one at a time in a specific manner. In particular, each partition

class P_i is assigned a set S_i of pivots that will be used to refine P_i only. The set S_i is implemented as a stack, and the order in which the pivots are pushed onto S_i is crucial. When v is taken from S_i to be the next pivot, $N(v)$ performs an in situ refinement on P_i. We use τ_i to denote the final (refined) ordering of P_i. When all partition classes have been refined, we concatenate all the τ_i's in order, i.e., $\tau = \tau_1 \cdot \tau_2 \cdot ... \cdot \tau_p$ where $\cdot$ denotes concatenation, and use τ to denote the final ordering. Our main theorem is the following:

Theorem 2. *Let $G(V, E)$ be a cocomparability graph, τ is a LexDFS cocomparability ordering that can be computed in $\mathcal{O}(m + n)$ time.*

We next discuss each step of the algorithm in more detail.

3.1 Vertex Labelling

Let $G(V, E)$ be an undirected cocomparability graph. Let $\sigma = v_1 \prec_\sigma v_2 \prec_\sigma ... \prec_\sigma v_n$ be a cocomparability ordering of G returned by the algorithm in [14]; we use $\sigma = ccorder(G)$ to denote such an algorithm. For every vertex $v \in V$, we assign a label $\#(v)$ initialized to the number of nonneighbours of v to its right in σ: $\#(v) = |\{u|uv \notin E \text{ and } v \prec_\sigma u\}|$.

Given such a labelling of the vertices, we create a partition of V denoted by $\mathcal{P} = \bigcup_{i=1}^{p} P_i$ in the following manner: Initially all vertices are marked unvisited, P_1 is the set of vertices with the smallest $\#$ label value. Now all vertices in P_1 are marked to be visited. For all unvisited vertices u and for all $v \in P_1$, such that $uv \in E$, $\#(u)$ is incremented by 1. To create P_2, again select the set of unvisited vertices of smallest $\#$ value. These vertices in P_2 are marked to be visited and for each such $v \in P_2$ and unvisited u adjacent to v, $\#(u)$ is incremented by 1. We increment i and repeat this operation of creating a partition class of the vertices with the smallest label until all vertices belong to a partition class.

Algorithm 2 is a formal description of the algorithm which takes a cocomparability graph $G(V, E)$ as input and returns the partition $\mathcal{P} = (P_1, P_2, ..., P_p)$.
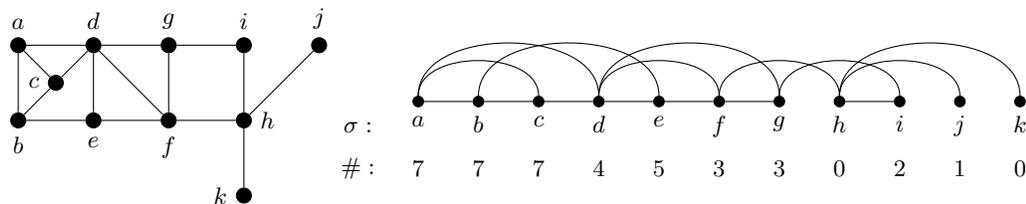

Fig. 2. $G(V, E)$, a cocomparability order σ of V, and the initial labelling $\#$ of V.

Let $\pi = P_1 \cdot P_2 \cdot ... \cdot P_p$ be the order of V resulting from Algorithm 2 such that $\forall x \in P_i, y \in P_{j>i}$, we have $x \prec_\pi y$. The order inside each P_i is

---

Algorithm 2 PartitionClasses

Input: a cocomparability graph $G(V, E)$
Output: partition $\mathcal{P}$ of V with p partition classes, and p

  $\sigma \leftarrow ccorder(G(V, E))$ $\triangleright$ As computed in [14]
  $S \leftarrow \emptyset$
  for $i \leftarrow n$ downto 1 **do**
      $\#(v_i) \leftarrow (n - i) - |S \cap N(v_i)|$ $\triangleright$ Initial labelling $\#(v)$
      $S \leftarrow S \cup \{v_i\}$
  end for
  $U \leftarrow V$ $\triangleright$ U the set of unvisited vertices
  $i \leftarrow 1$
  while U not empty **do**
      $P_i \leftarrow \{v | \#(v) = min(\#(U))\}$ $\triangleright$ Creating Partition Classes
      $U \leftarrow U \backslash P_i$
      for $v \in P_i$ **do**
          for $u \in U$ and $uv \in E$ **do**
             $\#(u) \leftarrow \#(u) + 1$
          end for
      end for
      $i \leftarrow i + 1$
  end while
  $p \leftarrow i - 1$
  return $\mathcal{P} \leftarrow (P_1, P_2, ..., P_p)$ and p

---

arbitrary. Consider the graph in Fig. 2 with a valid cocomparability vertex ordering. The numbers below the vertices are their labels as computed by Algorithm 2. $PartitionClasses(G(V, E))$ would then produce the following partition: $\mathcal{P} = \{P_1 = \{h, k\}, P_2 = \{j\}, P_3 = \{g, i\}, P_4 = \{d, e, f\}, P_5 = \{a, b, c\}\}$, as shown in Table 1.

P_i	smallest label	$v \in P_i$	Labels to increment
P_1	0	{h, k}	$\#(j) = \#(j) + 1 = 2$
			$\#(i) = \#(i) + 1 = 3$
			$\#(f) = \#(f) + 1 = 4$
P_2	2	{j}	
P_3	3	{g, i}	$\#(d) = \#(d) + 1 = 5$
			$\#(f) = \#(f) + 1 = 5$
P_4	5	{d, e, f}	$\#(a) = \#(a) + 1 = 8$
			$\#(b) = \#(b) + 1 = 8$
			$\#(c) = \#(c) + 1 = 8$
P_5	8	{a, b, c}	

Table 1. Creation of the partition classes for the graph in Fig. 2

In the remainder of the paper, we will use $\#^*(v_i)$ to refer to the initial value of v_i's label, i.e., the number of nonneighbours of v to its right in σ; and $\#_k(v_i)$ to denote the label value of v_i when P_k is being created, i.e., at iteration k.

3.2 Partition Refinement

Once all the partition classes are computed, we reorder the adjacency list of each v according to π in $\mathcal{O}(m+n)$ time (Section 4), then construct a new ordering of V by refining $\mathcal{P}$. Our refinement (algorithm *Refine*) is slightly different than the generic partition refinement algorithm presented in [11] and briefly explained in Section 2.

Algorithm 3 Refine

Input: a partition class P ordered by π and its corresponding ordered list of pivots S
Output: refinement τ of P
1: $Q_1 \leftarrow P$, $k \leftarrow 1$
2: **while** S not empty **do**
3:     $j \leftarrow 1$
4:     $v \leftarrow S.pop$ $\triangleright$ S is implemented as a stack
5:     **for** $i \leftarrow 1$ to k **do**
6:         **if** $|Q_i \cap N(v)| = 0$ or $|Q_i \cap N(v)| = |Q_i|$ **then**
7:             $Q'_j \leftarrow Q_i$
8:             $j \leftarrow j + 1$
9:         **else**
10:            $Q'_j \leftarrow Q_i \cap N(v)$
11:            $Q'_{j+1} \leftarrow Q_i \backslash N(v)$
12:            $j \leftarrow j + 2$
13:         **end if**
14:     **end for**
15:     $k \leftarrow j - 1$
16:     **for** $i \leftarrow 1$ to k **do** $\triangleright$ Rename the new partitions for the next pivot
17:         $Q_i \leftarrow Q'_i$
18:     **end for**
19: **end while**
20: **return** $\tau \leftarrow Q_1 \cdot Q_2 \cdot ... \cdot Q_k$ $\triangleright$
    $x \in Q_i, y \in Q_{j>i} \implies x \prec_\tau y$ and $x, y \in Q_i \implies x \prec_\tau y$ iff $x \prec_\pi y$

Given the partition $\mathcal{P} = (P_1, P_2, ..., P_p)$ returned by Algorithm 2, we associate a set S_i to each P_i, where S_i is a set of pivots that will be used to refine P_i. We say that a partition class P_i is *processed* when it has been refined. We use τ_i to denote the final ordering of P_i after it has been refined, i.e. $\tau_i = Refine(P_i, S_i)$. If a partition class P_i has an empty pivot set S_i, then for τ_i the (arbitrary) ordering of P_i in π is used.

The sets S_i are implemented as stacks and are created as follows: $S_1 = \emptyset$ and P_1 is considered processed. For all $P_{i>1}$, we scan τ_1 from *left to right* and

---

Algorithm 4 UpdatePivots

Input: a newly refined partition class P_j and its index j
Output: updated pivot lists for the upcoming partition classes, i.e. for P_i, $i > j$

1: **for** $v \in P_j$ **do**   ▷ in the τ_j order
2:   **if** v has neighbours in $P_{i>j}$ **then**
3:    $S_i.push(v)$ ▷ Update the pivot list of P_i
4:   **end if**
5: **end for**

---

for each $v \in \tau_1$ and every $u \in P_{i>1}$ where $uv \in E$, we push v in S_i. In general, every time a partition class $P_{j<i}$ is refined, i.e., τ_j has been produced, we scan τ_j from left to right, and for every $v \in \tau_j$ with neighbours in $P_{i>j}$, we push v into S_i. To refine P_i, we pop elements of S_i one at a time, and for each $v \in S_i$, v is the pivot that refines P_i by reordering P_i into the subpartitions $P_i \cap N(v)$ followed by $P_i \backslash N(v)$. The next pivot out of S_i performs an *in situ* refinement of the current subpartitions of P_i.

Let $u_j^1, u_j^2, ..., u_j^k$ be the left to right ordering of the vertices inside τ_j. We mentioned in Section 1 that not only this refinement is in situ, but also *backwards*. Backwards in two ways: First, the pivots of τ_j have a higher priority, i.e. a stronger pull, than the pivots of $\tau_{k<j}$, and second the pivots are pushed down the stack S_i in $\tau_1 \cdot \tau_2 \cdot ... \cdot \tau_{i-1}$ order (left to right) and thus are popped in reverse order. Therefore we maintain the priority of the pivots in the backward order: $(\tau_1 \cdot \tau_2 \cdot ... \cdot \tau_{i-1})^-$, but also the priority of the pivots inside each $\tau_{j<i}$ in the backward order τ_j. That is for any two vertices $u_j^a, u_j^b \in \tau_j$ where $a < b$, if u_j^a and u_j^b are both pivots for P_i, then u_j^b refines P_i first before $u_j^{a<b}$. Note that this is very similar to standard partition refinement with the difference that in standard partition refinement, P_i is first refined by τ_1 then τ_2, τ_3 and so on. Here we start refining with the last vertex in τ_{i-1}, then τ_{i-2}, and so on up to τ_1. This opposite refinement shows the key difference between LexDFS and LexBFS. Whereas in a LexBFS order the earliest neighbours have the strongest pull and the latest neighbours the weakest, in a LexDFS the last vertices are more influential then the earlier visited ones.

P_i ordered by π	S_i	Pivots	Refinement	τ_i
P_1: {h, k}	$\emptyset$	-	(h, k)	h, k
P_2: {j}	h	h	(j)	j
P_3: {g, i}	h	h	((i)(g))	i, g
P_4: {d, e, f}	g, h	g	((d, f)(e))	f, d, e
		h	((f)(d)(e))	
P_5: {c, a, b}	e, d	e	((b)(c, a))	b, c, a
		d	((b)(c, a))	

Table 2. The refinement of the P_i's constructed in Table 1

Algorithm 3, *Refine*, takes P_i and S_i as input, and returns the new ordering τ_i of P_i. Algorithm 4, *UpdatePivots*, takes τ_j as input, the refined ordering of P_j, and updates the stacks S_i for all unprocessed partition classes $P_{i>j}$.

Let τ denote the final ordering of all the refined partition classes, i.e., $\tau - \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_p$. Using the graph again in Fig. 2, we show in Table 2 the in situ refinement of each P_i. The final ordering is $\tau = h, k, j, i, g, f, d, e, b, c, a$.

3.3 The Complete Algorithm

We are now ready to present the complete algorithm *CCLexDFS*.

Algorithm 5 CCLexDFS

Input: a cocomparability graph $G(V, E)$
Output: a LexDFS order τ of G that is also a cocomparability order of G
1: $\tau \leftarrow \emptyset$
2: $(\mathcal{P}, p) \leftarrow PartitionClasses(G)$ $\triangleright$ Compute the partition classes
3: $S_1, ..., S_p \leftarrow \emptyset$
4: **for** $i \leftarrow 1$ to p **do**
5:      $\tau_i \leftarrow Refine(P_i, S_i)$ $\triangleright$ Refine the partition classes
6:      $UpdatePivots(\tau_i, i)$ $\triangleright$ Update the pivot sets
7:      $\tau \leftarrow \tau \cdot \tau_i$
8: **end for**
9: **return** τ

---

4 Correctness of the Algorithm

We denote the set of partition classes P_1 to P_{i-1} by $\mathcal{P}_i = (P_1, P_2, ..., P_{i-1})$.

Lemma 1. *For any u, v such that $u \prec_\sigma v$ and $uv \notin E$: $\#^*(u) > \#^*(v)$ and at any step i, $\#_i(u) > \#_i(v)$.*

Proof. Since $u \prec_\sigma v$ and $uv \notin E$, any vertex w with $v \prec_\sigma w$, if $wv \notin E$ then $wu \notin E$ otherwise uw flies over v - a contradiction to σ being a cocomparability ordeirng. Thus w contributes equally to $\#^*(u)$ and $\#^*(v)$. Moreover, since $u \prec_\sigma v$ and $uv \notin E$, v also contributes to $\#^*(u)$. Therefore $\#^*(u) > \#^*(v)$.

Suppose at a step $i, \#_i(u) < \#_i(v)$ then a vertex $z \in P_{j<i} \in \mathcal{P}_i$ must have closed the gap in $\#^*(u) > \#^*(v)$ by contributing to $\#_j(v)$ but not to $\#_j(u)$. Let $z \in P_{j<i}$ be such a vertex, we are only interested in the case when $zv \in E$ and $zu \notin E$, since adding z to $P_j \in \mathcal{P}_i$ would have incremented $\#_j(v)$, making it closer to $\#_j(u)$. Notice that $u \prec_\sigma z$, otherwise zv flies over u which contradicts σ being a cocomparability ordering. Therefore z contributes one to $\#^*(u)$, thus not reducing the gap between u and v's labels and $\#_i(u) > \#_i(v)$. $\qquad\square$

Lemma 2. *For $1 \leq i \leq p$, P_i is the set of maximal elements in the poset $P \backslash \bigcup_{j=1}^{i-1} P_j$.*

Proof. The proof is by induction on the partition classes P_i. P_1 is the set of vertices that are adjacent to all the vertices to their right in σ. Since σ is a cocomparability ordering, it is a linear extension of a poset $P(V, \prec)$. Thus P_1 is the set of elements in P that are incomparable to all the elements to their right in the linear extension σ of P. Hence the elements of P_1 are maximal elements in P.

Assume for sake of contradiction that P_i is the first partition class where the vertices of P_i are not the maximal elements of the poset $P \backslash \bigcup_{j=1}^{i-1} P_j$. In particular, let v denote an element in P_i that is not maximal in $P \backslash \bigcup_{j=1}^{i-1} P_j$. This means there exists an element u where $v \prec u$ and u is an element of $P \backslash \bigcup_{j=1}^{i-1} P_j$. Since σ is a linear extension of P and $v \prec u$, we have $v \prec_\sigma u$. And since u and v are comparable, they are non adjacent in G. Therefore, $v \prec_\sigma u$ and $uv \notin E$. By Lemma 1, $\#_i(v) > \#_i(u)$ and thus $v \notin P_i$. $\qquad\square$

Lemma 3. *Every partition class $P_i \in \mathcal{P}$ returned by Algorithm 2 is a clique.*

Proof. By Lemma 2, every P_i is the set of the maximal elements in the poset $P \backslash \bigcup_{j=1}^{i-1} P_j$. And maximal elements in a poset form an antichain, i.e., an independent set in the comparability graph, $\overline{G}$. Thus every P_i is a clique in G. $\qquad\square$

Lemma 4 (The Flipping Lemma). *Let σ be a cocomparability order, and τ the corresponding ordering created from σ and returned by Algorithm 5. Then for every $uv \notin E$, $u \prec_\sigma v \iff v \prec_\tau u$.*

Proof. As we are assigning vertices to their partition classes, let u and v be the left most pair of vertices in τ to satisfy $uv \notin E$ and $u \prec_\sigma v$ and $u \prec_\tau v$. By Lemma 1, $\#^*(u) > \#^*(v)$. And by Lemma 3, u and v belong to two different partition classes; P_i and $P_{j>i}$ respectively since $u \prec_\tau v$. Therefore when P_i was created $\#_i(u) < \#_i(v)$, which contradicts Lemma 1. Therefore $v \prec_\tau u$.

For sufficiency, using the contrapositive we know that $(u \prec_\tau v \Rightarrow v \prec_\sigma u)$ if and only if $(u \prec_\sigma v \Rightarrow v \prec_\tau u)$. Thereby completing the proof. $\qquad\square$

Corollary 1. *τ is a cocomparability order of G.*

Proof. As in [3], if τ is not a cocomparability order as witnessed by $x \prec_\tau y \prec_\tau z$ and $xz \in E, xy, yz \notin E$, then by the Flipping Lemma we have $y \prec_\sigma x$ and $z \prec_\sigma y$, which implies that the umbrella zx flies over y in σ, contradicting the fact that σ is a cocomparability order. $\qquad\square$

We are now ready to prove our main theorem, Theorem 2. Namely, that the ordering τ produced by Algorithm 5 is a LexDFS cocomparability ordering of G. Implementation details follow in the next section.

Proof (of Theorem 2). By Corollary 1, we know that τ is a cocomparability order of G. Suppose it is not a LexDFS order. Therefore for some triple a, b, c with $a \prec_\tau b \prec_\tau c$, $ac \in E$ and $ab \notin E$, there doesn't exist a vertex d as required by the 4 Point Condition (Theorem 1). Since τ is a cocomparability order, $bc \in E$ to destroy the umbrella ac over b.

Suppose b and c belong to the same partition class P_i. Since $ab \notin E$ and $a \prec_\tau b$, we have $a \in P_{j<i}$. Since $ac \in E$, a is a pivot with respect to P_i and thus b and c could not have been in the same subclass since a would have pulled c before b. Since $b \prec_\tau c$, a pivot u that pulled b in front of c must exist, i.e. $ub \in E, uc \notin E$. Pick u to be the rightmost pivot to b to satisfy this configuration. Notice that $a \prec_\tau u$. Otherwise, since the refinement is backwards, a would have refined P_i before u thus pulling c in front of b. Therefore $a \prec_\tau u$. This means that u plays the role of d with respect to LexDFS, a contradiction to our assumption; therefore b and c must be in different partition classes.

Since $b \prec_\tau c$, $b \in P_i$ and $c \in P_{j>i}$, when P_i was created $\#_i(b) < \#_i(c)$. We investigate how this gap could have occurred given $ac, bc \in E$ and $ab \notin E$. Without loss of generality, let a, b, c be the left most triple in τ that does not satisfy the 4 Point Condition, and consider the vertices that have contributed to $\#_i(b)$ and $\#_i(c)$. Let u be one of these vertices. Thus u increased $\#_i(b), \#_i(c)$ either by being a non adjacent right neighbour of b or c in the initial ordering σ (Algorithm 2, line 6) or u changed $\#(b), \#(c)$ when u was assigned to a partition class (Algorithm 2, lines 12-16).

If u is contained in a set of $\mathcal{P}_i$, then $u \prec_\tau b \prec_\tau c$. Consider all the possible adjacencies between u, b and c. If $ub \in E$ and $uc \notin E$, then either $a \prec_\tau u$ in which case u plays the role of d as required by LexDFS; or $u \prec_\tau a$ in which case u increments $\#_j(b)$ at iteration j when u was assigned to $P_{j<i}$, a set in $\mathcal{P}_i$, and u also contributes to $\#^*(c)$ by the Flipping Lemma. Thus u contributes equally to the labels of b and c. If $ub, uc \in E$ then u increments both b's and c's labels when it is assigned to a partition class; and if $ub, uc \notin E$, then by the Flipping Lemma, u contributes to both $\#^*(b)$ and $\#^*(c)$. But in all three cases u does not reduce the gap between b and c's labels. Therefore $ub \notin E, uc \in E$. However by the Flipping Lemma, $b \prec_\sigma u$, and thus u contributes to $\#^*(b)$ since $ub \notin E$, but also u increments c's label since $u \in \mathcal{P}_i$ and $uc \in E$; again not reducing the gap. Therefore u must be in $V \backslash \mathcal{P}_i$.

If u is not in a set of $\mathcal{P}_i$, then u has not been assigned to a partition class yet. Since u is responsible for the gap $\#_i(b) < \#_i(c)$, u created this gap when b and c were assigned their initial labels $\#^*(b)$ and $\#^*(c)$. For u to create such a gap, u must contribute to c's initial label $\#^*(c)$ and not contribute to $\#^*(b)$. In other words, $uc \notin E$ and $c \prec_\sigma u$. Therefore by the Flipping Lemma, $u \prec_\tau c$. Moreover, for u to not contribute to $\#^*(b)$, we either have $ub \in E$ or $ub \notin E$ but $u \prec_\sigma b$. Notice that this latter case is impossible, since $u \prec_\sigma b \Rightarrow b \prec_\tau u$ (by the Flipping Lemma); but also $u \prec_\tau c$ causing bc to fly over u in τ and contradicting Corollary 1. Thus $ub \in E, uc \notin E$ and $c \prec_\sigma u$. Since u is not in a set of $\mathcal{P}_i$, $b \prec_\tau u$, otherwise u plays the role of d with respect to LexDFS. Moreover, $au \in E$ since τ is a cocomparability order; and the triple a, b, u must satisfy the LexDFS ordering otherwise we contradict the choice of a, b, c as $u \prec_\tau c$. Therefore there must exist a vertex w such that $a \prec_\tau w \prec_\tau b, wb \in E, wu \notin E$; this forces the edge aw in order to avoid the umbrella au over w. If $wc \notin E$, then w plays the role of d as required by LexDFS for the triple a, b, c, and if $wc \in E$, then the umbrella wc flies over u, contradicting τ being a cocomparability order.

Therefore there must always exists a vertex that satisfies the LexDFS ordering for $\#_i(b) < \#_i(c)$ to hold; and thus τ is a LexDFS cocomparability order of G. □

5 Implementation Details

Recall that $\#^*(v_i)$ refers to the initial value of v_i's label, i.e., the number of nonneighbours of v to its right in σ; and $\#(v_i)$ is the label value of v_i if it was incremented at some previous iteration.

The Partition Classes: We assume we are given an adjacency list representation of G, where each adjacency list is a doubly linked list. The ordering σ is also implemented as a doubly linked list. Every vertex v_i is represented with a node structure containing a variable *pos* to store v_i's position i in σ, a variable indicating v_i's label value ($\#(v_i)$), a pointer *pc* initialized to NULL to indicate v_i's partition class, and a pointer *pt* to v_i's adjacency list.

Lemma 5. *Let* $G(V, E)$ *be a cocomparability graph. The algorithm* Partition-Classes *takes* $\mathcal{O}(m+n)$ *time to compute the partition classes of* G.

Proof. Step 1 of *PartitionClasses* is computed in $\mathcal{O}(m+n)$ time [14]. Consider step 3: Computing the vertices' labels can be done in linear time by scanning the adjacency list of each v_i and keeping track of the number of vertices v_j where $j > i$. Then the initial label value is just $\#^*(v_i) = n - i - |\bigcup_{v_i v_{j>i} \in E} v_j|$. For each vertex v_i, we scan its adjacency list in $\mathcal{O}(d_{v_i})$ steps where d_{v_i} is the degree of v_i. Therefore steps 3 to 6 take $\mathcal{O}(m+n)$ time.

For the while loop in line 9 to 18, consider a set of bins $B_0, ..., B_{n-1}$ together with corresponding counter variables $c_0, ..., c_{n-1}$, where c_i's value gives the number of elements in B_i. Initially all bins are empty and all $c_i = 0$. Each bin uses a doubly linked list to store its vertices. When vertex v's initial label is computed, v is placed in bin B_i where $i = \#^*(v)$, and c_i is incremented by one. Once all vertices are assigned to a bin, we use these bins to create the p partition classes $P_1, ..., P_p$ in linear time as follows: We process the nonempty bins one at a time in increasing order of their indices. For every bin B_i where B_i is the left most unprocessed bin with at least one vertex (i.e., $c_i \neq 0$), we place the vertices of B_i in P_j where $j \geq 1$ is the next unused index. For every vertex v placed in P_j we update v's *pc* variable to point to P_j, and examine each vertex u adjacent to v. If $u \in B_k$, $k > i$, then u is moved to B_{k+1} and c_k is decremented by one and c_{k+1} is incremented by one; since the bins are doubly linked lists, the insertion and deletion of vertices take constant time. Therefore when processing a bin, for each one of its vertices v, we do at most $\mathcal{O}(d_v)$ increments and decrements for the neighbours of v, and thus creating the partition classes takes at most $\mathcal{O}(m+n)$ time over all vertices.

Once all the partition classes are created, we reorder the adjacency list of each v according to π in $\mathcal{O}(m+n)$ time. To do so, we create new adjacency lists

as follows: We go through the vertices according to π's ordering and for every vertex $v \in \pi$ and every u adjacent to v, we add v to u's new adjacency list, and we update v's pointer *pos* to point to v in π. Fig. 3 below illustrates this construction for vertex g of the graph in Fig. 2 with neighbours restricted to partition class P_4.
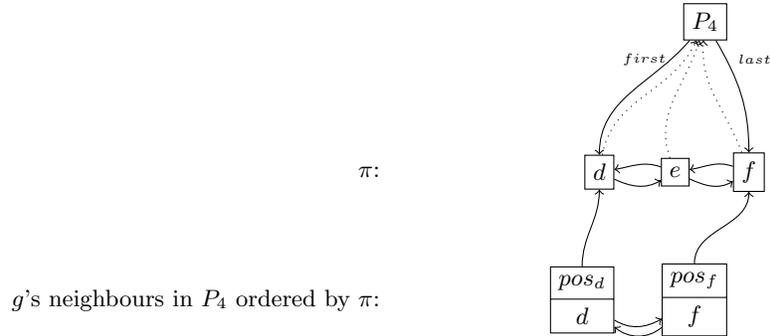

Fig. 3. The new adjacency list of vertex g restricted to the partition class P_4 as constructed in Table 1.

The Partition Refinement: We implement *Refine* using a similar data structure as the one presented in [11], where π is a doubly linked list and each P is implemented as a doubly linked list with three pointers: one to its first ($\mathbf{f}$) element in π, one to its last ($\mathbf{l}$), and one denoted $\mathbf{c}$ (for current), initialized to NULL. Every vertex v in π points back to the partition class it belongs to. It is important to remember when the new adjacency lists were created, vertices in the adjacency lists point back to their image in π, as it was just illustrated in Fig. 3.

When analyzing *Refine*, we are under the assumption that the set of pivots S is given such that every $v \in S$ must have at least one neighbour in P. This is accomplished by Algorithm 4, *UpdatePivots* as follows: Every v knows the partition class to which it belongs, and the adjacency lists are sorted according to π, thus every time a partition class P_j is refined, we sweep through τ_j from left to right, and for each $v \in \tau_j$, with neighbours in $P_{i>j}$, v is pushed onto S_i. This operation takes at most $\mathcal{O}(d_v)$ steps per $v \in P_j$. Summing over all $v \in P_j$, it is easy to see that *UpdatePivots* is linear in the number of edges between P_j and any class $P_{i>j}$.

Now consider Algorithm 3, steps 5 to 15 in *Refine*: Every time we pop an element v out of S, we know that there must exist at least one (sub)partition of P (denoted Q_i in *Refine*) which contains a neighbour of v. When v is the pivot, we scan its adjacency list (but only the neighbours in P), in the order imposed by π. For each w adjacent to v where w has its image in a subpartition Q_i, we check if the pointer $\mathbf{c}$ of Q_i is NULL. If $\mathbf{c}$ is NULL, w is moved to the start of Q_i where the $\mathbf{f}$ pointer is updated accordingly, and $\mathbf{c}$ points to w as well. If $\mathbf{c}$ is not

NULL, w is placed right after the vertex that **c** points to, then **c** is updated to point to w. Notice if w is the last element in Q_i and has been moved, then the **l** pointer of Q_i is also updated accordingly. When all the neighbours of the pivot have been processed, we split the reordered (sub)partitions at every location where **c** is not NULL to reflect the new reordered subpartitions. Each newly created subpartitions has its **c** reset to NULL, and its first and last pointers updated. It's important to note that if a subpartition Q_i has $\mathbf{f} = \mathbf{l} = \mathbf{c}$, then we know that it has a singleton element and thus cannot be refined more, therefore its **c** pointer is not reset to NULL.

For further clarity on how the partition refinement is performed, we refer the reader to Appendix A for a detailed example.

Given the data structure we used, this reordering (i.e., refinement) of P takes linear time. Splitting a (sub)partition Q_i takes at most $\mathcal{O}(|N(v) \cap Q_i|)$ steps for every Q_i with $\mathbf{c} \neq$ NULL. For each Q_i, vertices between $\mathbf{f}$ and $\mathbf{c}$ are moved into a new subpartition Q_j, and Q_i becomes $Q_i \backslash Q_j$, and thus, creating Q_j and $Q_i = Q_i \backslash Q_j$ takes at most $\mathcal{O}(|N(v) \cap Q_i|)$ steps. Since this operation is performed on every $w \in N(v) \cap Q_i$, for every $v \in S$, we perform at most $\mathcal{O}(d_v)$ steps of refinement, thus a total of the number of edges going from vertices in S to P.

Algorithm 5: By Lemma 5, step 2 is computed in linear time. As was just shown, *Refine* is linear in the size of the edges between S_i and P_i. Therefore, summing over all p partition classes, i.e., steps 4 to 8 in *CCLexDFS*, we get $\Sigma |S_i| \in \mathcal{O}(m)$. The refinement is thus computed in $\mathcal{O}(m+n)$ time. Similarly, we just showed that *UpdatePivots* is linear in the number of edges between P_j and any class $P_{i>j}$; consequently looping over the p partitions, *UpdatePivots* takes a total of $\mathcal{O}(m+n)$ time. All the remaining steps in Algorithm 5 are clearly linear. This, together with the results of Section 4, complete the proof of Theorem 2.

6   Other Variants of The Algorithm

In this section, we show how to compute a LexDFS$^+$ cocomparability ordering, as well as a LexDFS cocomparability ordering when the input graph is a comparability graph. Both modifications are done in linear time.

Recall that computing a LexDFS cocomparability ordering was the bottleneck in achieving linear time algorithms to a number of problems on cocomparability graphs [4, 3]. These algorithms all require LexDFS cocomparability orderings as input. One way to get such orderings is to compute a variant known as a LexDFS$^+$ cocomparability ordering. In this section, we show how to modify our algorithm to compute a LexDFS$^+$ cocomparability ordering in linear time.

Definition 1. *As defined in [3], a LexDFS$^+$ is a variant of LexDFS that takes as extra input an ordering σ, where ties amongst eligible vertices are broken by choosing the right most vertex with respect to σ, when $\tau = LexDFS^+(\sigma)$.*

Therefore given σ a cocomparability order, LexDFS$^+(\sigma)$ always starts with σ's right most vertex and breaks ties accordingly. Recall that when the partition classes were created in Algorithm 2, the vertices in each P_i were arbitrarily ordered. Given an ordering $\sigma = v_1, v_2, ..., v_n$, we denote by $\sigma^- = v_n, v_{n-1}, ..., v_2, v_1$ the reverse ordering of σ.

Corollary 2 (of Theorem 2). *Prior to the partition refinement step (Algorithm 5, step 5), if the vertices inside each partition class were ordered according to σ^-, then the resulting τ is a LexDFS$^+(\sigma)$ ordering.*

Proof. Suppose vertices inside each P_i are ordered according to σ^-. Clearly the resulting ordering τ is a LexDFS cocomparability ordering, since Theorem 2 holds for any arbitrary ordering inside the partition classes. Suppose τ is not a LexDFS$^+$.

That is, at some iteration i, there exists two tied vertices u and v where $u \prec_\sigma v$ but $u \prec_\tau v$. u, v are tied if $N(u) \cap \mathcal{P}_i = N(v) \cap \mathcal{P}_i$ and $u, v \in P_i$, i.e., they are ready to be processed. S ince $u \prec_\sigma v$, and vertices inside P_i are ordered according to σ^-, v is left of u inside P_i. Since $N(u) \cap \mathcal{P}_i = N(v) \cap \mathcal{P}_i$, there doesn't exist a pivot $w \in \mathcal{P}_i$ to pull u in front of v when processing P_i. Therefore $v \prec_\tau u$. τ is a LexDFS$^+$. □

Claim. *LexDFS$^+$ can be implemented in $\mathcal{O}(m + n)$ time.*

Proof. To obtain an *LexDFS$^+$* ordering, we need to reorder the partition classes to maintain σ^-'s order. Since every vertex knows the partition class it belongs to, it suffices to sweep through σ from right to left and recreate the partition classes, which takes $\mathcal{O}(n)$ time. Once this is done, the vertices in each partition are ordered according to their σ^- order. The rest of the analysis remains the same. □

Suppose now that the input graph G is a comparability graph. One can compute an ordering σ, a transitive orientation of G in linear time using the algorithm in [14]. It is fairly straightforward to see that a LexDFS($\overline{G}$) cocomparability ordering can also be computed in time linear in the size of G. To do so, it suffices to modify two steps in the algorithm: 1. Collecting the maximal elements in the corresponding posets, and 2. reversing the refinement to pull the right neighbours (i.e. the non-neighbours in $\overline{G}$). More precisely, we first modify Algorithm 2 to decrement the labels in steps 14, instead of increasing them. Then in Algorithm 3, we pull the appropriate neighbours first, by swapping steps 10-11:

$$Q'_j \leftarrow Q_i \backslash N(v)$$
$$Q'_{j+1} \leftarrow Q_i \cap N(v)$$

Theorem 3. *Let G be a comparability graph. One can compute a LexDFS cocomparability ordering of $\overline{G}$ in linear time in the size of G.*

In [4], the authors gave a certifying LexDFS based algorithm that computes a maximum independent set on a cocomparability graph as well as a minimum clique cover as a certificate. With Algorithm 5, this certifying algorithm is a linear time algorithm. With Theorem 3, this certifying algorithm is a linear time certifying algorithm to compute a minimum colouring of a comparability graph and a maximum clique as a certificate.

7 Conclusion and Open Problems

We have presented the first linear time algorithm to compute a LexDFS cocomparability order, as well as a LexDFS$^+$ cocomparability ordering and a LexDFS cocomparability when given the complement graph, therefore answering a question raised in [3], and also overcoming the bottleneck in the near linear time algorithms in [3, 4]. Our algorithm was also used more recently in [16] to give the first linear time algorithm to compute a maximum cardinality matching on cocomparability graphs. It is still an open question whether there exists a linear time implementation for LexDFS on arbitrary graphs. Our implementation exploits the poset structure of the cocomparability graph. In fact, computing the partition classes is equivalent to computing the layers of the corresponding poset.

Clearly this leads to the obvious question of whether this algorithm can be modified to compute a LexDFS ordering of a comparability graph. In other words, what is the structure of LexDFS on comparability graphs, and thus on posets.

Looking at the power of LexDFS on cocomparability, and how it has led to simple and elegant algorithms on this graph family when LexBFS has failed, simply by extending the existing algorithms on interval graphs; it is natural to ask whether there are other problems that can be solved using a similar approach: First a LexDFS preprocessing, then extending the algorithm for interval graphs.

Moreover with this algorithm in hand now, preprocessing is 'easy', which raises the question of possible multisweep LexDFS algorithms. Multisweeps algorithms perform a number of sweeps (i.e., graph searches) where each sweep generally reveals more structural properties about the graph. LexBFS multisweep algorithms have led to a number of simple and elegant algorithms, see for instance [6, 7, 2].

Stepping away from cocomparability graphs but still looking at structured graph families, it is natural to ask whether LexDFS can be implemented in linear time for other restricted graph families, such as asteroidal triple free graphs, a graph family that contains cocomparability graphs. But also whether there are other applications to LexDFS in other graph classes. Graph searches have been exploited on various graph families, it is necessary to explore the possible insights LexDFS has to offer, in contrast with these other graph searches.

References

1. Andreas Brandstädt, Feodor F Dragan, and Falk Nicolai. Lexbfs-orderings and powers of chordal graphs. *Discrete Mathematics*, 171(1-3):27–42, 1997.
2. Pierre Charbit, Michel Habib, Lalla Mouatadid, and Reza Naserasr. A new graph parameter to measure linearity. *arXiv preprint arXiv:1702.02133*, 2017.
3. Derek G Corneil, Barnaby Dalton, and Michel Habib. Ldfs-based certifying algorithm for the minimum path cover problem on cocomparability graphs. *SIAM Journal on Computing*, 42(3):792–807, 2013.
4. Derek G Corneil, Jérémie Dusart, Michel Habib, and Ekkehard Kohler. On the power of graph searching for cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 30(1):569–591, 2016.
5. Derek G Corneil and Richard M Krueger. A unified view of graph searching. *SIAM Journal on Discrete Mathematics*, 22(4):1259–1276, 2008.
6. Derek G Corneil, Stephan Olariu, and Lorna Stewart. The lbfs structure and recognition of interval graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1905–1953, 2009.
7. Jérémie Dusart and Michel Habib. A new lbfs-based algorithm for cocomparability graph recognition. *Discrete Applied Mathematics*, 216:149–161, 2017.
8. Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004.
9. Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234(1):59–84, 2000.
10. Michel Habib and Lalla Mouatadid. Maximum induced matching algorithms via vertex ordering characterizations. *arXiv preprint arXiv:1707.01245*, 2017.
11. Michel Habib, Christophe Paul, and Laurent Viennot. Partition refinement techniques: An interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(02):147–170, 1999.
12. Ekkehard Köhler and Lalla Mouatadid. A linear time algorithm to compute a maximum weighted independent set on cocomparability graphs. *Information Processing Letters*, 116(6):391–395, 2016.
13. Dieter Kratsch and Lorna Stewart. Domination on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 6(3):400–417, 1993.
14. Ross M McConnell and Jeremy P Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.
15. George B Mertzios and Derek G Corneil. A simple polynomial algorithm for the longest path problem on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 26(3):940–963, 2012.
16. George B Mertzios, André Nichterlein, and Rolf Niedermeier. Linear-time algorithm for maximum-cardinality matching on cocomparability graphs. *arXiv preprint arXiv:1703.05598*, 2017.
17. J Spinrad. Efficient implementation of lexicographic depth first search. *Submitted for publication*, 2012.

A   A Refinement Example

we consider the following example. Suppose *Refine* takes a partition class P and a set of pivots S such that:

$$P = \{b, a, f, e, d\}$$
$$S = \{x, y, z\}$$
$$N(x) \cap P = \{b, a, e, d\}$$
$$N(y) \cap P = \{a, f, e\}$$
$$N(z) \cap P = \{b, a\}$$

Suppose the ordering in each set is fixed (as would be the case when π imposes its ordering), and pivots are popped out of S in the order of x then y then z. Fig. 4, 5, and 6 illustrate how each pivot refines P (i.e., by reordering the vertices, and splitting at the pointer **c**).
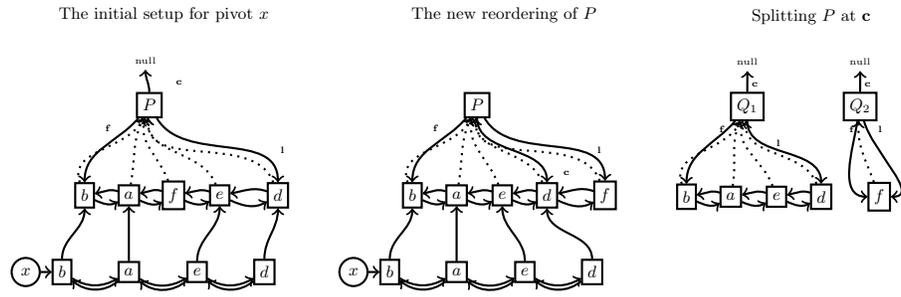
The initial setup for pivot x ⟶ The new reordering of P ⟶ Splitting P at **c**

Fig. 4. Processing pivot x

The next pivot y will refine according to the newly created subpartitions as follows:

The initial setup for pivot y ⟶ The new reordering of the subpartitions ⟶ Splitting the subpartitions at **c**
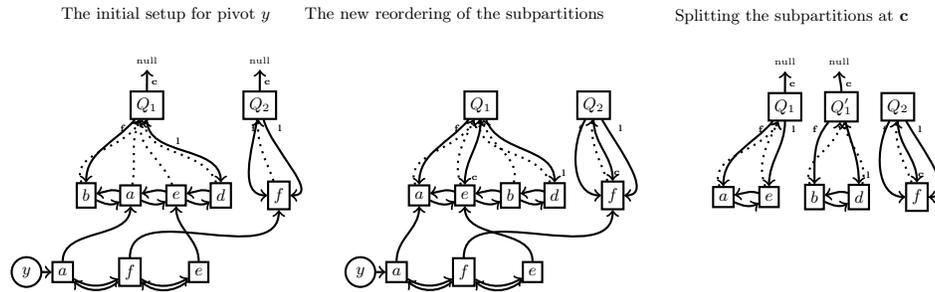

Fig. 5. Processing pivot y

Notice how Q_2 has all its pointers pointing at the same element f, therefore the current pointer **c** is not reset to NULL. The next pivot is z.
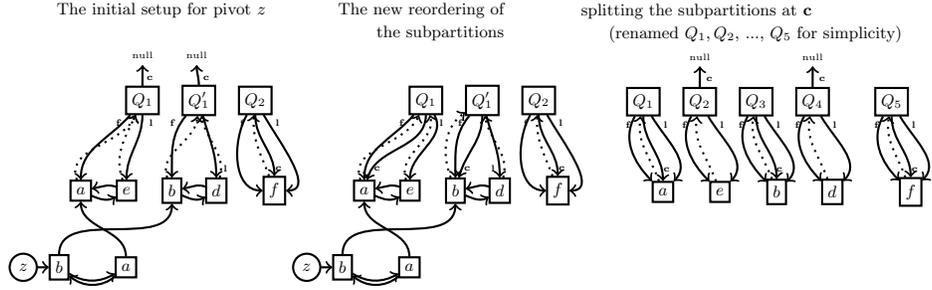

Fig. 6. Processing pivot z

The refinement of $P = \{b, a, f, e, d\}$ is therefore $\tau = \{a, e, b, d, f\}$.