EFFICIENT ALGORITHMS ON COCOMPARABILITY GRAPHS VIA VERTEX ORDERINGS

by

Lalla Mouatadid

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2018 by Lalla Mouatadid
Abstract
Efficient Algorithms on Cocomparability Graphs via Vertex Orderings
Lalla Mouatadid
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2018

In this thesis, we give new structural and algorithmic results on cocomparability (ccp) graphs; particularly, by connecting structural properties of graph searching to those of vertex ordering characterizations and linear extensions of posets. To do so, we present new tools that we hope will prove useful on other applications – especially a new graph partition refinement technique, and a new graph parameter that captures the linear structure exhibited by certain graph classes.

The thesis consists of two main parts. In the first half, we give the first linear time algorithm to compute a LexDFS ordering of ccp graphs. Doing so, we overcome a bottleneck to achieving certifying linear time Hamilton path, max independent set and max matching algorithms on this graph class. Building on this result, we give a new simple and elegant algorithm to the classical minimum bump problem on posets. Lastly, we introduce a new graph invariant, LexCycle, which measures how linearly structured certain graph classes are, this linear structure has been a driving factor to a number of efficient algorithms.

In the second half of the thesis, we lift our results to weighted ccp graphs. In particular, we give the first linear time robust algorithm for the weighted max independent set problem on this graph class; we then build on this result to give the fastest (O(mn) time) algorithm for the weighted induced matching problem on ccp graphs. In this latter work, we also give a unifying framework that captures graphs, characterized by forbidden orderings, which are closed under $L^2(\cdot)$, the operation of computing the square of the line graph.

We conclude by presenting structural properties on AT-free graphs, a superclass of ccp graphs, that we hope can be used to lift some of these results to this larger graph class.
In memory of Zi and Mma.
Acknowledgements

I postponed writing this page of my thesis because I knew I would not do it justice, but I have to submit my final thesis, so here we go. Obviously, I would not be sitting here writing the last section of my PhD dissertation if it was not for a number of wonderful, inspiring, and most importantly generous people I have been lucky to meet, work with, and learn from.

First and foremost, I would like to thank my advisors Derek Corneil and Allan Borodin, for giving me the opportunity to be their student; and in particular, for giving me the freedom to choose and work on my own problems, and develop my own research interests. I want to thank Derek for many things, especially for his kindness, his patience, and his wise guidance. I am indeed in debt to him for his patience to teach me his vast knowledge (that seems unattainable at times), his guidance while I was learning, his generous support to go to conferences, research visits, workshops, and for opening the door for me to a solid international network of collaborators and colleagues. There are many things I would like to think I have learnt from Derek; but if there is one I hope I did learn is his ability to not lose his patience over all aspects of research, and life in general. Thank you Derek for your support and guidance throughout all these years. I will forever be happy to have knocked on your door.

I would like to thank Allan for being patient and supportive while I was finding my way around the different areas of TCS. Thank you for exposing me to other topics, for always reminding me to broaden my research scope, to present my work at different venues, and for supporting my choices while allowing me to dive into other areas of TCS.

This thesis would not have been possible without the continuous support of Michel Habib. I want to thank Michel for his unshakable optimism, his passion for research, his choice of problems, his optimistic view on every aspect of research. I particularly want to thank Michel for being so generous with his time, his knowledge, and his guidance. It may not be written officially on documents, but Michel has definitely been a main advisor throughout my PhD, and for that, I will forever be grateful. Thank you Michel, for all the insightful and enjoyable conversations we have had these past years, about research, books, life and politics, for accommodating my Skype calls, for all the research visits, the advice, the support, and the encouragements. Thank you for the continuous flow of new problems and new ideas, and for patiently waiting for me to grasp the big picture of many topics without ever losing your patience. I hope this collaboration will not stop any time soon.

I would also like to thank my committee: Mike Molloy, for your careful readings, your advice, your challenging questions. I will not forget all the advice you gave me while going over one of my talks slide by slide – Thank you Mike. That conversation has since shaped the way I give talks. Thank you Sasho Nikolov, for being on my committee, for allowing
me to storm into your office and discuss my graph problems, but more importantly, thank
you for being the friend that you are. Thank you also to my external committee Sushant
Sachdeva, Nisarg Shah, and Kristina Vuskovic for your careful read of my thesis, and all
your comments and questions.

Research is a social activity, and I feel lucky to have been able to collaborate with many
colleagues around the world. I would like to thank all my co-authors, Ekki, Gara, Michel,
Pierre, Reza, Feodor, for sharing their knowledge with me, and for the fun (and stressful)
times we had staring at the board. A big thank you in particular goes to Ekki Kähler. I
can thank Ekki for many things, but the first one would be for giving me the confidence to
believe in my research abilities. Thank you Ekki for sharing your optimism while solving
problems with me, and for your friendship. I hope we will collaborate again.

I would like to thank Derek (again) and Barb for their friendship, and Steve and Linda
for their friendship as well as their unbelievable support during the infamous adventure I
had :). Thank you Toni for being the great mentor I look up to, for inspiring me, for your
encouragements, and for always having your door open. I hope we will continue to look for
(and find) a problem you and I can both work on. I would also like to thank Dominique
and Gara, both of whom have been a big influence on the Lalla I am today.

A big thank you also goes to the teaching faculty at DCS - especially Francois, Danny,
Karen, and Tom, for giving me the opportunity to dive deeper into my passion of teaching.

Thank you to my awesome group of friends who have made Toronto home: Roro (espe-
cially for all the music conversations), Noah, Jimmy, Atiyeh, Joel, Spooner, Jay, David(s),
Tyrone, Venkatesh, Mika, and Yuval: Thank you for making the lab the fun environment
that it was despite the lack of windows. To my “non-lab” friends, Teresa, Lindsay, Chris,
Aida, Gabby, David, Sarah, Bobby, thank you for reminding me there is more to life than
TCS.

Finally, I would like to thank my family: papa, maman, Meo, Sawsaw, Mimzi. I don’t
know who or what I would be without you. Je vous aime. A special thank you goes to my
sister, Sawsaw. I cannot thank Sawsaw enough. I love you, and look up to you. You inspire
me every day. And the biggest thank you goes to my parents for all the sacrifices they have
made for my siblings and me, for their continuous support, for inspiring us every day, for
their love, and for being the best parents anyone can hope to have. This thesis is dedicated
to them, to Zi and to Mma.
Contents

1 Introduction 1
   1.1 Our Contributions .......................................... 4
   1.2 Overview of the Thesis ...................................... 5

2 Background 8
   2.1 Preliminaries ................................................ 8
   2.2 Cocomparability Graphs ...................................... 11
   2.3 Graph Searching .............................................. 15
      2.3.1 Lexicographic Breadth First Search .................. 15
      2.3.2 Lexicographic Depth First Search ..................... 18
      2.3.3 Some Graph Searching Properties ..................... 20
      2.3.4 Partition Refinement .................................. 21

3 Linear Time LexDFS on Cocomparability Graphs 25
   3.1 Overview of the Algorithm .................................. 25
   3.2 Vertex Labeling .............................................. 27
   3.3 Partition Refinement ........................................ 29
   3.4 The Complete Algorithm .................................... 31
   3.5 Correctness of the Algorithm ............................... 32
   3.6 Implementation and Running Time ........................... 36
   3.7 Variants of the Algorithm: LexDFS^+ & LexDFS(\overline{G}) 39
   3.8 Conclusion ................................................... 41

4 The LexDFS Structure of Posets 42
   4.1 Lexicographic labelings of Posets ........................... 42
      4.1.1 Theorems about Lexicographic Labelings ............. 44
      4.1.2 The Bump Number Algorithm: GreedLex ............... 46
      4.1.3 Implementation and Running Time ...................... 50
# LexCycle: A New Graph Parameter

5.1 Motivation ........................................... 52
5.2 LexCycle: First Properties .......................... 53
5.3 Graph Classes with Small LexCycle ............... 58
  5.3.1 Interval Graphs & Proper Interval Graphs ..... 59
  5.3.2 Co-bipartite Graphs ............................. 63
  5.3.3 Domino-free Cocomparability Graphs .......... 66
  5.3.4 Trees ........................................... 69
5.4 Conclusion ........................................... 71

# Vertex Orderings Based Algorithms on Weighted Graphs

6.1 A Robust Linear Time Algorithm for the Weighted MIS on Cocomparability Graphs .............................. 74
  6.1.1 Overview of The Algorithm ...................... 75
  6.1.2 Correctness and Robustness of the Algorithm ... 77
  6.1.3 Implementation and Running Time ............... 80
6.2 Weighted Induced Matching via Vertex Orderings .................. 81
  6.2.1 Closure Under $L^2(\cdot)$ ....................... 83
  6.2.2 Vertex Orderings of $L^2(G)$ in Linear Time ... 88
  6.2.3 Application: $O(mn)$ Time WMIM on Cocomparability Graphs .............................. 89
6.3 Weighted Graph Searches ................................ 94

# AT-free Graphs

7.1 Structural Properties ................................ 98

# Future Work

8 Future Work ........................................... 112

# Bibliography

Bibliography ............................................ 114

# Appendices

Appendices .............................................. 121

# A Refinement Example

A Refinement Example .................................. 122
Chapter 1

Introduction

The goal of this thesis is to present new structural and algorithmic results on cocomparability graphs. These results come in two folds: First, we explore and use graph searching as a mechanism to reveal structural properties of this graph class. This technique is particularly powerful when combined with the vertex ordering characterization of cocomparability graphs – known as cocomparability orderings. Second, we study these orderings to extract structure and algorithmic results that could not have been seen with graph searching; in particular, when lifting our structural properties to weighted problems, graph searching does not capture the weights constraint. Throughout this work we encounter applications in order theory.

**Definition 1.1.** A graph $G(V, E)$ is a cocomparability graph if $G$ is the complement of a comparability graph, i.e., a graph $\overline{G}(V, \overline{E})$ whose edge $\overline{E}$ can transitively oriented.

Cocomparability graphs (and their complements, comparability graphs) are well known and well studied classes of graphs [35]. Many NP-complete problems have polynomial time solutions on cocomparability graphs. Many problems however are not solved on the given graph but are rather translated into posets or comparability problems. This transformation requires the computation of the complement, thus imposing a $O(n^2)$ term even though a linear time solution might exist. For instance, the solutions to the independent set problem as well as the colouring problem both follow from the polynomial time algorithms on comparability graphs [36]. Problems such as the Hamilton path and cycle and the minimum path cover problem exploit the relationship between cocomparability graphs and the bump number in posets [23, 24]. In order to design faster algorithms, there has been a growing motivation to exploit the structure of cocomparability graphs without computing their complements [12, 13, 58]. These algorithms use the vertex ordering characterization of cocomparability graphs known as a cocomparability ordering, or an umbrella-free ordering, defined as:
Chapter 1. Introduction

**Definition 1.2.** An ordering \( \sigma \) is a cocomparability ordering if for any triple \( a \prec_\sigma b \prec_\sigma c \): if \( ac \in E \) then either \( ab \in E \) or \( bc \in E \).

If neither \( ab \) nor \( bc \) are in \( E \), we say that the umbrella \( ac \) flies over \( b \), and we call \( a, b, c \) a bad triple.

Cocomparability graphs form a large graph class that includes cographs, interval, permutation and trapezoid graphs, all of which arise naturally in different applications such as biology, archaeology, scheduling and transportation to name a few [36]. These graph classes have been studied separately, and extensively, and solving problems on their superclass, cocomparability graphs, not only extends the solutions to all of them, but often sheds light on some underlying structure that is the silent workhorse behind a number of algorithms.

A successful approach that worked on graph classes has been the use of a mechanism known as graph searching. This is just a way to traverse the graph using some specific prescribed rule. A graph search that has proven quite powerful is lexicographic breadth first search, or LexBFS for short. Introduced in 1976 by Rose, Tarjan, and Lueker, LexBFS was first used to recognize chordal graphs [67]. Since then, it has been used as a multisweep algorithm to recognize interval graphs [19], permutation graphs [51], cographs [4], and more recently (2017), cocomparability graphs [29]. LexBFS has also proven useful in coloring algorithms, for instance [68] presents a LexBFS based algorithm to color Meyniel graphs, and in computing dominating pairs on AT-free graphs [18].

LexBFS seems to fail for optimization problems on cocomparability graphs, however. In 2008, another graph search was introduced which later has helped remedy this problem: lexicographic depth first search – LexDFS. Introduced by Corneil and Krueger in [16] in a body of work that unifies various graph searches, LexDFS did not have any applications at first. It was not until 2013, that Corneil, Dalton, and Habib used LexDFS to give the first near-linear time algorithm to compute a minimum path cover on cocomparability graphs [12]. Their algorithm works by first “preprocessing” a cocomparability ordering with LexDFS, then running the algorithm that works on interval graphs on cocomparability graphs. Since this first result, a number of other algorithms have appeared, that follow this same template. In [60], Mertzios and Corneil used LexDFS on cocomparability graphs to give the first polynomial \( O(n^4) \) time algorithm to compute a longest path on this graph class. In [13], Corneil, Dusart, Habib, and Köhler gave a certifying LexDFS based algorithm to compute a maximum cardinality independent set. In 2017, Mertzios, Nichterlein, and Niedermeier gave a LexDFS based algorithm to compute a maximum cardinality matching [61]. All of these algorithms follow a lifting framework from interval to cocomparability graphs, which consists of two main steps:

1. **Preprocess** a cocomparability ordering with LexDFS.
2. **Lift**, by sometimes slightly modifying, the algorithm that works on interval orderings
to cocomparability orderings.

The fact that all these algorithms follows this same technique raises the question of what structure is revealed during this LexDFS preprocessing. One can conjecture that cocomparability graphs have some sort of “interval-like” structure, or more broadly “linear-structure” that allows these (often) greedy algorithms to work optimally on these preprocessed orderings. One way to formalize what is meant by linear-structure is to think of the graph as having a spine: The vertices are either on the spine, or at a close distance to it. Prior to this idea of preprocessing with LexDFS, this spine of the graph was only seen by taking a step back into an even larger graph class: Asteroidal triple free graphs (AT-free). In [18], Corneil, Olariu, and Stewart showed that connected AT-free graphs, a superclass of cocomparability graphs, always have a dominating pair that can be computed in linear time. That is, there exists a path $P$ where every vertex is either on $P$ or has a neighbour on $P$. Other ways to capture this linear structure on cocomparability and AT-free graphs are presented in Chapter 7 of this thesis.

This lifting framework however seems to be a stronger technique to exhibit this linearity, since it allows cocomparability graphs to behave almost like interval graphs. The first step – the preprocessing step – has been the bottleneck to achieving linear running time for a number of these algorithms. One of the contributions of this work is the first linear time algorithm to compute these so-called LexDFS cocomparability orderings. With the structural results we have learnt in this work, we also give a new simple algorithm to compute linear extensions of posets with minimum bump.

Now, although LexDFS helps extract this sort of “interval-like” structure, all of these algorithms first require a cocomparability ordering as an input, and LexDFS is not useful when it comes to this recognition problem – these orderings characterize this graph class. However, a recent LexBFS based recognition algorithm for cocomparability graphs was presented in [29]. The algorithm works by running a number of LexBFS traversals, and shows that the $n^{th}$ ordering is a cocomparability ordering. In fact, it runs a variant of LexBFS known as LexBFS$^+$ $n$ times. This result is reminiscent of the interval graph recognition algorithm given by Corneil and Köhler where they also showed that the $n^{th}$ LexBFS$^+$ ordering is what we call an interval ordering [14]. In fact, in an unpublished manuscript, Ma showed that for every constant $c$, there exists an interval graph $G_c$ such that $c$ LexBFS$^+$ traversals are not enough to produce an interval ordering [57]. Since all the known LexDFS based algorithms on cocomparability require a cocomparability ordering as an input, and LexBFS is the graph search we know how to use to produce such orderings, it is natural to wonder whether LexBFS is the right graph search to also find this “interval-like” structure on cocomparability graphs. After all, the LexDFS based algorithms mentioned above all require a cocomparability ordering first before proceeding to this “lifting” technique. LexDFS
does not produce such orderings even after $O(n)$ sweeps. In an attempt to understand this phenomena better, we introduced a new graph parameter that captures this linear-like structure. This parameter, called LexCycle, is computed from a series of $\text{LexBFS}^+$ traversals that (eventually) cycle since the number of such orderings is finite. The LexCycle of a graph $G$ is the length of this cycle. We show that many well known graph classes have small LexCycle, 2 in particular. It is open whether the class of cocomparability graphs also has LexCycle two, but we show that for many of its subclasses, that is the case. One of the consequences of these algorithms is the simplest algorithm to transitive orient these graphs. Going back to the dominating path structure on cocomparability and AT-free graphs; it is worth noting that the spine algorithm in [18] is based on $\text{LexBFS}$. The algorithm (which we present in Chapter 7) uses $\text{LexBFS}$ to reach extremities\footnote{But not necessarily diametral pairs} of the graph, which form a dominating pair. This is a pair of vertices $u, v$ that dominates the graph. That is, given any $uv$ path $P$ in $G$, every vertex of $G$ either is on $P$ or has a neighbour on $P$.

With this toolbox of graph searching and vertex orderings in hand, it is only natural to ask whether we can use the same techniques if the graphs are weighted. The second half of this thesis focuses on this problem. Unfortunately, and not surprisingly, graph searching techniques do not work on weighted graphs; although very recently (2017), some work has been done in this direction [54]. But using just the structure of cocomparability orderings, we were able to show that these orderings are enough for certain problems; in particular for the maximum weighted independent set and the maximum weighted induced matching problems.

### 1.1 Our Contributions

This thesis consists of two main parts: In the first half we present new structural and algorithmic results on cocomparability graphs and some of its subclasses by looking at graph searching and multisweep algorithms. In particular:

1. In Chapter 3, we give the first linear time algorithm to compute a $\text{LexDFS}$ ordering on cocomparability graphs. To do so, we introduce a new partition refinement technique, and we show how the underlying poset structure has been guiding these algorithms that fit into the lifting framework mentioned above. We also modify the algorithm to work on the complement in linear time in the size of the input graph, and show how to compute the $^+$ invariant in linear time as well.

2. In Chapter 4, we give a new simple algorithm to compute linear extensions of posets with minimum bump, i.e., minimum number of contiguous pairs that are comparable
in the poset. This algorithm is inspired by the structure we learned in Chapter 3.

3. In Chapter 5, we focus on LexBFS on cocomparability graphs, which was used for recognition in [29] and for computing a dominating pair in [18]. This latter result shows that LexBFS produces some sort of “spine” structure on this graph class. We try to capture this linear structure by introducing and computing a graph invariant, LexCycle, which measures the size of a cycle of orderings when running LexBFS+. It was conjectured by Stacho that the size of this cycle is at most the asteroidal number of the graph [72]; we disprove Stacho’s conjecture, but conjecture that for cocomparability graphs, it is indeed two, i.e., the asteroidal number of this graph class. Towards proving this conjecture, we show that a number of subclasses of cocomparability graphs (interval, cobipartite, cographs, domino-free cocomparability) all have LexCycle two.

The second half of the thesis explores problems on weighted cocomparability graphs. In this setting, we prove the following results:

4. In Chapter 6, we use cocomparability orderings to compute a maximum weighted independent set. The algorithm runs in linear time, i.e., $O(m+n)$ time. This algorithm is robust, meaning, it either outputs a correct solution, or it can be modified to output a bad triple in the given cocomparability ordering.

5. We build on top of this result to give the fastest algorithm to compute a maximum weighted induced matching on this graph class, $O(mn)$ time, which is an improvement over the $O(n^4)$ algorithm for the unweighted case [38]. Furthermore, we show that for graph classes characterized by a set of forbidden vertex orderings, these graph classes and their vertex orderings are closed under the $L^2(\cdot)$ operation – computing the square of the line graph – and give an algorithm to compute characterizing orderings in linear time, without computing their line graphs or their squares.

In Chapter 7, we present some new structural properties on AT-free graphs that we hope can lead to lifting some of these results on cocomparability graphs to this larger graph class.

1.2 Overview of the Thesis

This thesis is organized as follows: We give the necessary terminology and background in Chapter 2. Chapters 3 to 7 present our results outlined above. We conclude in Chapter 8 with future directions and extensions of this line of work.
Many of the results in this thesis are based on (and may have been taken verbatim from) the following publications. New extensions and other results presented in the thesis have not appeared yet.


This thesis does not include the following work:

- Feodor Dragan, Michel Habib, and Lalla Mouatadid. Graph Searches and Geometric Convexities in Graphs. We discuss the relationship between graph searches and graph convexities. Nearly every graph search produces a convex geometry, we thus study graph searches and graph classes for which this convexity is interesting. These results will be presented at the International Colloquium on Graph Theory and Combinatorics (ICGT) this year – 2018.

- Michel Habib, Lalla Mouatadid, and Mengchuan Zou. Approximating Modular Decomposition is Hard. In this work, we relax the notion of modular decomposition by introducing $\epsilon$-modules,
a relaxation of modules that allows us to maintain some algebraic structure. We show that minimal $\epsilon$-modules can be computed in polynomial time, but that is it NP-hard to compute if a graph admits an $\epsilon$-modular decomposition tree.

- Lalla Mouatadid and Robert Robere
  Path Graphs, Clique Trees and Flowers
  In this work, we look at the structure of asteroidal triples to characterize path graphs. This work is inspired by the classical result of Lekkeikerker and Boland that characterizes interval graphs as both chordal and AT-free graphs.

The numbering in this thesis follows the format “Theorem $i.j$” where $i$ is the chapter number and $j$ the numbering within that chapter. Similarly, “Algorithm $i.j$” refers to the $j^{th}$ algorithm in Chapter $i$. 
Chapter 2

Background

2.1 Preliminaries

We begin by giving the necessary definitions used in this thesis.

A graph $G$ is a pair $(V, E)$ where $V$ is a set of vertices of size $n$ and $E$ a set of edges of size $m$. All the graphs considered in this thesis are simple (no loops or multiple edges), finite and undirected. Two vertices $u, v$ are adjacent if there is an edge $e \in E$ whose endpoints are $u$ and $v$; we write $uv$ to denote the edge in $E$ with endpoints $u$ and $v$. We denote by $N(v) = \{u : uv \in E\}$ the open neighbourhood of vertex $v$, and $N[v] = N(v) \cup \{v\}$ the closed neighbourhood of $v$. The degree of a vertex $v$ is $\deg(v) = |N(v)|$. We denote by $\Delta_G$ the maximum degree in $G$.

The complement of a graph $G(V, E)$ is the graph $\overline{G}(V, \overline{E})$ where $uv \in \overline{E}$ if and only $u \neq v$ and $uv \notin E$. A set $S \subseteq V$ is an independent set if for all $a, b \in S, ab \notin E$; and $S$ is a clique if for all $a, b \in S, ab \in E$. A graph $G(V, E)$ is a complete graph if $V$ is a clique of $G$. A complete graph on $k$ vertices is denoted by $K_k$. The independence (resp. clique) number of a graph $G$, denoted $\alpha(G)$ (resp. $\omega(G)$), is the size of the maximum independent set (resp. clique) of $G$. The chromatic number of a graph $G$, denoted $\chi(G)$, is the minimum number of colours required to properly colour $G$, i.e., to assign colours to $V$ such that adjacent vertices receive different colours.

We write $G[V']$ to denote the induced subgraph $H(V', E')$ of $G(V, E)$ on the subset of vertices $V' \subseteq V$, where for every pair $u, v \in V', uv \in E'$ if and only if $uv \in E$.

A path $P$ in $G$ is a sequence $v_1, v_2, \ldots, v_k$ of vertices where $v_iv_{i+1} \in E$ for $1 \leq i \leq k - 1$. We denote by $V(P)$ the set of vertices of $G$ that appear on the path $P$. If there are no other edges in $G[V(P)]$ then $P$ is an induced path. A path is a cycle $C$ if $v_1 = v_k$. Given a cycle $C$, a chord $c$ in $C$ is an edge that connects two vertices of $C$ where $c$ itself is not an edge of $C$. A $k$-cycle (resp. $k$-path) is a cycle (resp. path) on $k$ vertices, we write $C_k$ (resp. $P_k$).
class is a maximal module of modular decomposition $A$ of a single vertex. This is known as the $G$-module of a prime. A module $M$ is a single vertex. A graph is a matching $M$ if every pair of edges in $M$ is either adjacent to all vertices in $G$ or to none of them. A module $M$ is a twin if $N(u) = N(v)$; if in addition $uv \in E$, then $N[u] = N[v]$ and $u$ and $v$ are called true twins, otherwise they are called false twins.

A module of a graph $G(V,E)$ is a subset $M$ of vertices such that any vertex in $V \setminus M$ is either adjacent to all vertices in $M$ or to none of them. A module $M$ is trivial if $M = V$ or $M$ is a single vertex. A graph is prime if all its modules are trivial, i.e., all its modules are prime. A module $M$ is maximal in $G(V,E)$ if for any $x \notin M$, $M \cup \{x\}$ is not a module of $G$. A modular decomposition of $G(V,E)$ is a partitioning of the vertices in which each partition class is a maximal module of $G$. Given $M = \{M_1, M_2, \ldots, M_k\}$ a modular decomposition of $G$, we write $G/M$ to denote the graph constructed by contracting every module $M_i$ into a single vertex. This is known as the quotient graph of $G$.

Given a graph $G(V,E)$, the line graph of $G$, denoted $L(G)(E, L(E))$, is the graph on $m$ vertices, where every vertex in $L(G)$ represents an edge in $G$, and two vertices in $L(G)$ are adjacent if and only if their corresponding edges share an endpoint in $G$. We write $L^2(G)(E, L^2(E))$ to denote the square of the line graph of $G$.

A matching $M \subseteq E$ is a subset of edges no two of which share an endpoint. An induced matching $M^* \subseteq E$ is a matching in $G$ where every pair of edges in $M^*$ is at distance at least two in $G$; alternatively, where every pair of edges in $M^*$ induces a $2K_2$ (the disjoint
union of 2 cliques of size 2).

A triple of independent vertices \(u, v, w\) forms an asteroidal triple (AT) if every pair of the triple remains connected when the closed neighbourhood of the third vertex is removed from the graph. In general, a set \(A \subseteq V\) of \(G\) forms an asteroidal set if for each vertex \(a \in A\), the set \(A \setminus \{a\}\) is contained in one connected component of \(G[V \setminus N[a]]\). The maximum cardinality of an asteroidal set of \(G\), denoted \(\text{an}(G)\), is called the asteroidal number of \(G\).

A domino is the induced graph \(G(V = \{a, b, c, d, e, f\}, E = \{ab, ac, bd, cd, ce, df, ef\})\).

![Figure 2.1: Domino](image)

Given a graph \(G(V, E)\), a total ordering \(\sigma\) of \(V\) is a bijection \(\sigma : V \to [n]\), where \([n]\) denotes the set of natural numbers \(\{1, 2, \ldots, n\}\). We often refer to \(\sigma\) as an ordering – instead of total ordering. For \(v \in V\), \(\sigma(v)\) refers to the position of \(v\) in \(\sigma\). For a pair \(u, v\) of vertices we write \(u <_\sigma v\) if and only if \(\sigma(u) < \sigma(v)\); we also say that \(u\) (resp. \(v\)) is to the left of (resp. right of) \(v\) (resp. \(u\)). We write \(\{\sigma_i\}_{i \geq 1}\) to denote a sequence of orderings \(\sigma_1, \sigma_2, \ldots\). Given such a sequence, and an edge \(ab \in E\), we write \(a <_i b\) if \(a <_{\sigma_i} b\), and \(a <_{i,j} b\) if \(a <_i b\) and \(a <_j b\). Given an ordering \(\sigma = v_1, v_2, \ldots, v_n\) of \(G\), we write \(\sigma^d\) to denote the dual (also called reverse) ordering of \(\sigma\); that is \(\sigma^d = v_n, v_{n-1}, \ldots, v_2, v_1\). For an ordering \(\sigma = v_1, v_2, \ldots, v_n\), the interval \(\sigma[v_s, \ldots, v_t]\) denotes the ordering of \(\sigma\) restricted to the vertices \(\{v_s, v_{s+1}, \ldots, v_t\}\) as numbered by \(\sigma\). Similarly, if \(S \subseteq V\), and \(\sigma\) an ordering of \(V\), we write \(\sigma[S]\) to denote the ordering of \(\sigma\) restricted to the vertices of \(S\).

Given a graph class \(\mathcal{G}\), we say that \(\mathcal{G}\) has a vertex ordering characterization if there exists a set of properties \(\mathcal{P}\), such that for all \(G, G \in \mathcal{G}\) if and only if there exists a total ordering of the vertices of \(G\) that satisfies the set of properties \(\mathcal{P}\).

A partially ordered set (poset for short) \(P(V, <)\) is a set of elements \(V\) and a reflexive, antisymmetric and transitive relation \(<\) on \(V\). The transitive reduction (resp. closure) of a poset \(P\) is another poset \(P'\), on the same set of elements as \(P\), that does not include any (resp. includes all) transitive edges. We take \(<\) to mean the transitive reduction of a poset, and \(<\) to mean the transitive closure. We say that two elements \(a, b \in V\) are comparable if \(a < b\) or \(b < a\), otherwise they are incomparable, and we write \(a \parallel b\). The diagram of a poset \(P\) is a drawing of the elements of \(P\), where the minimal elements are drawn at the bottom, and for any pair of comparable elements \(a < b\), \(a\) is drawn below \(b\) and \(a, b\) are connected by an edge. The Hasse diagram of \(P\) is the diagram of its transitive reduction, i.e., the
transitive edges are not represented. A **filter** \((F \subseteq V, \prec)\) of \(P\) is an induced sub-poset where \(\forall x \in F, \forall y \in V\) if \(y < x\) then \(y \in F\). An **ideal** \((I \subseteq V, \prec)\) of \(P\) is an induced sub-poset where \(\forall x \in I, \forall y \in V\) if \(x < y\) then \(y \in I\). The **upper cover** of an element \(x \in V\) is the set \(S\) of elements where \(\forall y \in V, x \prec y \implies y \in S\). Similarly, a **lower cover** of an element \(x \in V\) is the set \(S\) of elements where \(\forall y \in V, y \prec x \implies y \in S\).

A poset is a **total order** if all its elements are pairwise comparable. A total ordering is represented as a linear ordering \(\sigma = a_1, a_2, \ldots, a_n\) where \(a_1 \prec a_2 \prec \ldots \prec a_n\). A **linear extension** \(L\) of a poset \(P(V, \prec)\) is a total ordering of \(V\) which respects the partial order of \(\prec\). That is, the \(L\) can impose any arbitrary on the incomparable elements of \(P\) as long as \(L\) respects the ordering of \(\prec\). Given \(L\) a linear extension of \(P\), we say that two consecutive elements \(a_i, a_{i+1}\) in \(L\) form a **bump** if \(a_i \prec a_{i+1}\). The number of bumps in a linear extension is denoted by \(b(L, P)\). The **bump number** of a poset is defined as \(b(P) = \min\{b(L, P) | L\) is a linear extension of \(P\}\). A **chain** \(C\) of \(P\) is a sequence of \(a_1 a_2 \ldots a_k\) where \(a_i \prec a_{i+1}\), for \(i \in [k-1]\). The **depth** of an element \(x\) in \(P\), denoted \(\text{depth}(x)\), is the length of the longest chain where \(x\) is as a minimal element. An **antichain** is a subset \(S \subseteq V\) of pairwise incomparable elements in \(P\). The **width** of \(P\), often denoted \(w\), is the size of its largest antichain. A **layer** in a poset \(P(V, \prec)\) is an antichain \(S \subseteq V\), where \(\forall u, v \in S, \text{depth}(v) = \text{depth}(u)\). A **layering** of a poset \(P\) is a partition of \(P\) into layers. The **shelling** of \(P\) is the removal of elements of \(P\) that are minimal. For \(S \subseteq V\), we write \(P \setminus S\) to mean the induced sub-poset \(P'\) of \(P\) on \(V \setminus S\), and \(L \setminus S\) to mean a linear extension of \(P'\).

Throughout the thesis, we write \(A \cdot B\) to denote concatenation, whether \(A, B\) are paths, edges, vertex orderings, linear extensions, or lexicographic labels.

### 2.2 Cocomparability Graphs

The focus of this body of work is on **cocomparability graphs**. A graph is a cocomparability graph if it is the complement of a comparability graph. A comparability graph is a graph whose edge set admits a transitive orientation. That is, \(G(V, E)\) is a comparability graph if one can orient the edges in \(E\) so that for every triple \(a, b, c \in V\), if the edge \(ab\) is oriented \(a \rightarrow b\) and the edge \(bc\) is oriented \(b \rightarrow c\), then the edge \(ac\) must exist and is oriented \(a \rightarrow c\). This orientation is known as a **transitive orientation** of \(G\). A comparability graph can have multiple transitive orientations. We illustrate below an example of two graphs; the graph on the left is a comparability graph with an illustrative transitive orientation, and the graph on the right does not have a transitive orientation.
Chapter 2. Background

Figure 2.2: A comparability graph (left) and a non-comparability graph (right).

Given a poset $P(V, \prec)$, one can construct a unique graph $G(P) = G(V, E)$ on the same set of elements (vertices) where $uv \in E$ if and only if $u$ and $v$ are comparable in $P$. It is easy to see that the graph $G(P)$ is a comparability graph: Orienting the transitive closure of $P$ bottom up in the Hasse diagram gives a transitive orientation of $G(P)$. The poset below for instance has the graph (to the left) in Fig. 2.2 as its $G(P)$.

Figure 2.3: A poset $P$ with $G(P)$ in Fig. 2.2.

For every poset $P$, one can associate a corresponding cocomparability graph as well, the complement of $G(P)$. Conversely, since a comparability graph can have multiple transitive orientations, each of these orientations (up to symmetry) can give rise to a different Hasse diagram, and thus a different poset.

Cocomparability graphs and their complements are well studied. Many NP-complete problems have polynomial time solutions on these graph classes, the main reason being that cocomparability graphs are perfect, and thus the chromatic, independence, and clique number can all be computed in polynomial time for these graphs and their complements. For the fastest known algorithms, many problems however are not solved directly on cocomparability graphs. Indeed, the line of work in this area had for a while focused on solving problems either on comparability graphs or on posets, and transporting the solutions to cocomparability graphs. More recently, there has been a growing interest in solving problems directly on this graph class, and a few factors contributed to this shift. The first and obvious one is to see if the $O(n^2)$ cost of going through the complement and/or the poset can be overcome. The second one is the introduction of a new graph traversal that has proved particularly powerful on this graph class (see Chapter 3).

A third factor has to do with the graph class itself; cocomparability graphs form a large
family that captures a number of real world problems, but also contains various classes that model different applications as well. In particular, cographs, interval, permutation, and trapezoid graphs are some of the well studied graph classes in the literature that are all contained in cocomparability graphs. Thus, efficient algorithms on this large class would lead to efficient algorithms on all its subclasses. Some of the real world problems that these graphs capture include the well known physical mapping problem in biology which tries to reconstruct a sequence of the DNA when given fragments of said DNA [64], problems also in scheduling [65], and VLSI design [70] for instance.

Since we will often come back to some of these graph classes, and prove results on them as well, let us define them formally.

- A graph \( G(V, E) \) is a **cograph** if \( G \) can be constructed from a single vertex by disjoint union and complementation. Cographs are also characterized as the graphs that do not contain an induced path on four vertices as an induced subgraph, and also the graphs that are totally decomposable with respect to modular decomposition.

- A graph \( G(V, E) \) is an **interval** graph if \( V \) can be represented as a collection of non-empty intervals on the real line. That is, for every \( v \in V \), create an interval \( I_v \) such that \( uv \in E \) if and only if \( I_v \cap I_u \neq \emptyset \). This is known as \( \mathcal{R} \), the interval representation of \( G \).

- A graph \( G(V, E) \) is a **proper interval** graph if \( G \) is an interval graph whose interval representation has no interval fully contained in another interval.

These classes are incomparable. Take cographs and proper interval graphs for instance; a \( P_4 \), the forbidden subgraph of cographs, is a proper interval graph, and a \( C_4 \) is a cograph but not a proper interval graph – not an interval graph either.

In [39], Golumbic, Monma, and Trotter showed that cocomparability graphs are a subclass of **asteroidal-triple free graphs**. Asteroidal-triple free graphs (AT-free for short) are the graphs that do not contain asteroidal triples, thus \( an(G) \leq 2 \) for \( G \) AT-free. AT-free graphs were initially introduced by Lekkerkerker and Boland to characterize interval graphs, in their now classical theorem:

**Theorem 2.1.** [55] A graph \( G \) is an interval graph if and only if \( G \) is both an AT-free graph and a chordal graph.

Then Gilmore and Hoffman tightened Theorem 2.1 to the following:

**Theorem 2.2.** [34] A graph \( G \) is an interval graph if and only if \( G \) is both a cocomparability graph and a chordal graph.
A graph $G$ is chordal if its largest induced cycle is a triangle – alternatively, $G$ is chordal if it does not contain a hole. This graph class has a characterizing vertex ordering, known as a perfect elimination ordering – or PEO for short.

**Definition 2.3.** An ordering $\sigma = v_1, v_2, \ldots, v_n$ is a PEO of a graph $G(V,E)$ if for every $i \in [n], v_i$ is simplicial in $G[\{v_1, \ldots, v_i\}]$; or alternatively, if for every $1 \leq i < j < k \leq n$, if $v_iv_k, v_jv_k \in E$ then $v_iv_j \in E$.

Interval graphs also have a characterizing vertex ordering, known as the I-ordering:

**Definition 2.4.** An ordering $\sigma = v_1, v_2, \ldots, v_n$ is an I-ordering of a graph $G(V,E)$ if for every $1 \leq i < j < k \leq n$, if $v_iv_k \in E$ then $v_iv_j \in E$.

The I-ordering is sometimes referred to as the left endpoint ordering of $G$ because there always exists an interval representation $\mathcal{R}$ of $G$, where the ordering of $\sigma$ is precisely the orderings of the intervals in $\mathcal{R}$ sorted by their left endpoints in increasing order. One can also construct a right endpoint ordering of $G$ as the ordering of the intervals in $\mathcal{R}$ sorted by decreasing right endpoints.

Proper interval graphs also have a vertex ordering characterization – the PI-ordering:

**Definition 2.5.** An ordering $\sigma = v_1, v_2, \ldots, v_n$ is a PI-ordering of a graph $G(V,E)$ if for every $1 \leq i < j < k \leq n$, if $v_iv_k \in E$ then $v_iv_j \in E$ and $v_jv_k \in E$.

As mentioned in the introduction, cocomparability graphs also have cocomparability orderings as characterizing vertex orderings.

**Definition 2.6** (Umbrella-Free Orderings). An ordering $\sigma = v_1, v_2, \ldots, v_n$ of the vertices of a graph $G$ is an umbrella-free ordering, if for every triple $1 \leq i < j < k \leq n$, if $v_iv_k \in E$ then either $v_iv_j \in E$ or $v_jv_k \in E$ or both.

In [52], Kratsch and Stewart showed the following characterization:

**Theorem 2.7.** [52] A graph is a cocomparability graph if and only if it has an umbrella-free ordering (Fig. 2.4).

In [52], Kratsch and Stewart showed the following characterization:

**Theorem 2.7.** [52] A graph is a cocomparability graph if and only if it has an umbrella-free ordering (Fig. 2.4).

![Figure 2.4: An umbrella ac over b](image)

Umbrella-free orderings are also known as cocomparability orderings. Notice that a cocomparability ordering is just a transitive orientation in the complement. Namely, if
both $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Therefore cocomparability orderings are also just linear extensions of some poset. This is a key idea that we will use often throughout this thesis. In [58], McConnell and Spinrad gave a linear time algorithm to compute cocomparability orderings. Let $ccorder(G)$ denote this algorithm.

**Theorem 2.8.** [58] Cocomparability orderings can be computed in linear time.

We abuse notation and write $\sigma = ccorder(G)$ to denote the ordering $\sigma$ produced by the algorithm $ccorder(G)$.

### 2.3 Graph Searching

Graph searching – sometimes called graph traversal, is a mechanism to visit every vertex of a graph, one at a time, in a specific manner, where the next chosen vertex to visit must not have been previously selected. Of course, there could be more than one vertex eligible to visit next. Two of the classical graph searches are *breadth first search* and *depth first search* (BFS and DFS). Roughly speaking, the former asks that all the unvisited neighbours of the last visited vertex be visited first before moving to the unvisited non-neighbours. Whereas the latter requires that we first visit vertices down a path, as long as there are unvisited vertices to select, before backtracking. However, as one can see from the nature of these searches, it is possible that at a given step there are more than one eligible vertex to visit. The “breadth” and the “depth” rules do not specify how to deal with “tied” vertices. One way to do so is to introduce a new rule; a tie breaking rule. This of course would lead to different graph searches, however if we add this tie-breaking rule to an already defined search $S$, we would just be refining $S$. This is precisely what **LexBFS** and **LexDFS** do.

#### 2.3.1 Lexicographic Breadth First Search

Lexicographic breadth first search (**LexBFS**) is a graph search variant of BFS that assigns lexicographic labels to vertices, and breaks ties between them by choosing vertices with lexicographically highest labels. The labels are words over the alphabet $\{0, \ldots, n - 1\}$. By convention $\epsilon$ denotes the empty word. The formal algorithm for **LexBFS** is given below, and a step by step example is shown in Table 2.1, using the graph in Fig. 2.5. Notice that the append operation in Step 7 forces this traversal to be a BFS ordering first.
Algorithm 2.1 LexBFS

**Input:** A graph \( G(V, E) \) and a start vertex \( s \)

**Output:** an ordering \( \sigma \) of \( V \)

1. assign the label \( \epsilon \) to all vertices
2. label\((s) \leftarrow \{n + 1\}\)
3. for \( i \leftarrow 1 \) to \( n \) do
   4. pick an unnumbered vertex \( v \) with lexicographically largest label
   5. \( \sigma(i) \leftarrow v \) \( \triangleright v \) is assigned the number \( i \)
   6. foreach unnumbered vertex \( w \) adjacent to \( v \) do
      7. append \( n - i \) to label\((w)\)
   8. end for
9. end for

As one can see from the example, the algorithm was forced to visit vertex \( c \) before \( d \) because \( b \) assigned its label to \( c \) and not \( d \).

Given a graph \( G(V, E) \), if a specific start vertex \( s \in V \) is given at which to start the LexBFS algorithm, then we write \( \text{LexBFS}(G, s) \), otherwise we write \( \text{LexBFS}(G) \) when any \( v \in V \) can be used as a start vertex. We also write \( \sigma = \text{LexBFS}(G) \) to denote the ordering \( v_1, v_2, \ldots, v_n \) in which the vertices of \( G \) have been visited by the algorithm. LexBFS was originally introduced by Rose et al. in [67] to recognize chordal graphs. They showed in
particular that the vertex ordering produced by LexBFS is always a PEO when $G$ is chordal. Since its introduction, LexBFS has been used successfully for a number of other algorithmic and structural results [67, 18, 68, 11, 19, 29]. One of the main reasons for this success is the relationship between the vertex ordering characterizations of both the graph class and the graph search. Indeed, LexBFS itself has its own vertex ordering characterization, known as the LexBFS 4 Point Condition, initially proven by Dragan, Falk and Brandstädt, and Golumbic:

**Theorem 2.9 (The LexBFS 4 Point Condition [27, 36]).** A vertex ordering $\sigma$ is a LexBFS ordering if and only if for every triple $a \prec_\sigma b \prec_\sigma c$ if $ac \in E, ab \notin E$, there must exist a vertex $d$ such that $d \prec_\sigma a$ and $db \in E, dc \notin E$.

In other words, the theorem says that if we visit a non-neighbour $b$ of $a$ before a neighbour $c$ of $a$, then $b$ must have a private neighbour with respect to $c$ that was visited before $a$. We illustrate this characterization below:

![Figure 2.6: The LexBFS 4 Point Condition.](image)

**Definition 2.10 (Bad Triple).** A bad triple in an ordering $\sigma$ is a triple of vertices $a \prec_\sigma b \prec_\sigma c$ where $ac \in E$ and $ab \notin E$.

Combining graph searching with vertex ordering characterizations has been especially successful on multisweep algorithms. A multisweep algorithm is an algorithm that computes a number of orderings where each ordering $\sigma_i, i > 1$ uses the previous ordering $\sigma_{i-1}$ to break ties using specified tie breaking rules. One rule in particular that has proven successful is the + rule, formally defined as follows: Given a graph $G(V, E)$, an ordering $\sigma$ of $G$, and a graph search $S$ (such as LexBFS), $S^+(G, \sigma)$ is a new ordering $\tau$ of $G$ that uses $\sigma$ to break any remaining ties from the $S$ search. In particular, given a set of tied vertices $T$, the + rule chooses the vertex in $T$ that is rightmost in $\sigma$. We sometimes write $\tau = S^+(\sigma)$ instead of $\tau = S^+(G, \sigma)$ if there is no ambiguity in the context. Multisweep algorithms are indeed a powerful tool to extract structure from a graph. With every traversal, new properties can be exposed. Since their introduction, LexBFS based multisweep algorithms have led to a number of algorithms, from graph recognition [67, 19, 29] to colouring [68], to computing dominating pairs [18]. Surprisingly however, it was not until 2008 that a corresponding extension of DFS was introduced formally as a graph search, similar to how LexBFS was presented [16].
2.3.2 Lexicographic Depth First Search

Lexicographic depth first search, LexDFS for short, is an extension of DFS that, similarly to LexBFS, assigns lexicographic labels to vertices, and breaks ties between eligible vertices using these labels. A formal description of the algorithm is given below, which we illustrate in Table 2.7 using the same graph in Fig. 2.5. In contrast with the LexBFS algorithm, LexDFS prepends labels to force the DFS nature of the search. As one can see from the example, the algorithm was forced to visit vertex e before d because b assigned its label to e and not d.

Algorithm 2.2 LexDFS

**Input:** A graph $G(V, E)$ and a start vertex $s$

**Output:** an ordering $\sigma$ of $V$

1: assign the label $\epsilon$ to all vertices
2: $\text{label}(s) \leftarrow \{0\}$
3: for $i \leftarrow 1$ to $n$ do
4: pick an unnumbered vertex $v$ with lexicographically largest label
5: $\sigma(i) \leftarrow v$ $\triangleright v$ is assigned the number $i$
6: foreach unnumbered vertex $w$ adjacent to $v$ do
7: preprend $i$ to $\text{label}(w)$
8: end for
9: end for

<table>
<thead>
<tr>
<th>vertex</th>
<th>$i=1$</th>
<th>$i=2$</th>
<th>$i=3$</th>
<th>$i=4$</th>
<th>$i=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$\epsilon$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>$\epsilon$</td>
<td>1</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>2</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>$d$</td>
<td>$\epsilon$</td>
<td>1</td>
<td>1</td>
<td>31</td>
<td>431</td>
</tr>
</tbody>
</table>

Figure 2.7: LexDFS$(G, a)$ produces the ordering $\sigma = a, b, c, e, d$.

The notion of a lexicographic depth first search goes back to 1984 where Ghosh and Bhattacharjee gave an NC algorithm\(^1\) for LexDFS in acyclic directed graphs [33]. It was not until 2008 however that LexDFS was reintroduced by Corneil and Krueger in [16] where they captured different graph searches by their corresponding vertex ordering characterizations. Indeed, LexDFS also has its 4 point condition, which says the following:

---

\(^1\)An algorithm that runs in polylogarithmic time using a polynomial number of parallel processors.
Chapter 2. Background

Theorem 2.11 (The LexDFS 4 Point Condition [16]). A vertex ordering $\sigma$ is a LexDFS ordering if and only if for every bad triple \( a \prec_\sigma b \prec_\sigma c \), there must exist a vertex $d$ such that \( a \prec_\sigma d \prec_\sigma b \) and $db \in E, dc \notin E$.

![Figure 2.8: The LexDFS 4 Point Condition.](image)

In contrast with LexBFS, vertex $b$ has a private neighbour $d$ with respect to $c$, except that now $d$ must have been visited later than $a$ due to the “depth first” nature of the search.

So far, LexDFS has found applications in cocomparability graphs only – with an exception in [21] where it was used for clustering algorithms. Since its introduction, many problems on cocomparability graphs, such as computing a maximum cardinality independent set and a minimum clique cover [13], a minimum path cover [12], a maximum matching [61], now have linear time solutions for this graph class, with the use of our linear time LexDFS algorithm (cf. Chapter 3). These successful approaches all fall into the “lifting framework”: They start by computing a cocomparability ordering $\sigma$ of the graph, and preprocessing $\sigma$ with a LexDFS$^+$ sweep and then extending, or slightly modifying, the linear time algorithms that work for interval graphs.

This approach is surprising for various reasons. First, being able to “lift” algorithms from a smaller class of graphs, such as interval graphs, to a class containing it such as cocomparability graphs suggests that maybe this latter family has some sort of linear structure, similar to the one exhibited by interval representations; especially, since there are subclasses of cocomparability graphs (permutation, trapezoid and cographs) that are incomparable to interval graphs, yet benefit from this lifting strategy. Second, the mere fact that all that was needed to lift these algorithms is to pre-process a cocomparability ordering $\sigma$ with a LexDFS$^+$ sweep suggests that this graph search is processing the graph in a specific way. Especially since LexBFS, which so far has been the successful graph search, fails miserably as a preprocessing step for all these algorithms. This raises the question of what kind of structure is LexDFS revealing on cocomparability graphs, that LexBFS fails to see. We answer this question in the chapters to follow. In particular, we will show that LexDFS orders the maximal cliques of a cocomparability graph by computing a layering of a corresponding poset. We make this idea clear in Chapter 3. Lastly, a number of these algorithms are not only simple but also certifying. For instance the maximum cardinality independent set algorithm also produces a clique cover as a certificate [13]. Of course, these results raise the
question of whether other algorithms on interval graphs can be lifted to cocomparability graphs, and whether LexDFS has applications on other graph classes. For the purpose of this thesis, we focus on cocomparability graphs, and in this next section, we show how these graph searching tools help expose more structure on this graph class.

### 2.3.3 Some Graph Searching Properties

We mentioned earlier that combining graph searching with vertex ordering characterizations has been a useful technique; we illustrate this claim with a few structural properties on cocomparability graphs.

**Theorem 2.12.** [13, 59] Let $\sigma$ be a cocomparability ordering. Then $\tau = \text{LexBFS}^+(\sigma)$ and $\phi = \text{LexDFS}^+(\sigma)$ are cocomparability orderings.

Thus these graph searches preserve cocomparability orderings. We will call an ordering a LexBFS (resp. LexDFS) cocomparability ordering if it is the result of a LexBFS$^+$ (resp. LexDFS$^+$) sweep of a cocomparability ordering. The next two properties illustrate the $C_4$ structure of LexBFS and LexDFS cocomparability orderings:

**Property 2.13 (The LexBFS $C_4$ Property [13]).** Let $\sigma$ be a cocomparability ordering and $\tau = \text{LexBFS}^+(\sigma)$. For every bad triple $a \prec \tau b \prec \tau c$, there exists a vertex $d$ (as defined in Theorem 2.9) such that the quadruple $d, a, b, c$ induces a $C_4 = da, ac, cb, bd$.

This follows from the LexBFS 4 Point Condition and the fact that LexBFS preserves cocomparability orderings. In particular, since $\tau$ is a LexBFS ordering, by Theorem 2.9, every bad triple in $\tau$ has a vertex $d$ that satisfies $d \prec \tau a \prec \tau b \prec \tau c$ such that $ac, db \in E$, $ab, dc \notin E$. Furthermore, by Theorem 2.12, $\tau$ is also a cocomparability ordering, and thus to avoid umbrellas in $d \prec \tau a \prec \tau b \prec \tau c$, we must have $da, cb \in E$. Therefore the quadruple $d, a, b, c$ induces a $C_4$. Similarly, LexDFS also has a $C_4$ property:

**Property 2.14 (The LexDFS $C_4$ Property [13]).** Let $\sigma$ be a cocomparability ordering and $\tau = \text{LexDFS}^+(\sigma)$. For every bad triple $a \prec \tau b \prec \tau c$, there exists a vertex $d$ (as defined in Theorem 2.11) such that the quadruple $a, d, b, c$ induces a $C_4 = ad, db, bc, ca$.

The $C_4$ structure of LexBFS cocomparability ordering is “twisted” whereas it is “open” for LexDFS as shown below. This of course does not mean all $C_4$s are either all twisted or all open; just consider a LexDFS on the simple graph on two $C_4$’s that share two edges.

![Figure 2.9: The LexBFS and LexDFS $C_4$ properties.](image)
A useful lemma to prove results on multisweep orderings is the Flipping Lemma:

**Lemma 2.15 (The Flipping Lemma [12, 13, 59]).** Let $\sigma$ be a cocomparability ordering, and $\tau = \text{LexBFS}^+(\sigma)$. For every pair of non-adjacent vertices $u, v$, $u \prec_\sigma v$ if and only if $v \prec_\tau u$. The same holds if $\tau = \text{LexDFS}^+(\sigma)$.

The proof of the Flipping Lemma is not hard; a short proof appears in Section 3.5.

### 2.3.4 Partition Refinement

There are two main ways to think about LexBFS (and LexDFS), the first one, via lexicographic labels as we have shown above. The second one is via a technique known as partition refinement.

Given a set $S$, we call $P = (P_1, P_2, \ldots, P_k)$ a partition of $S$ if $\bigcup_{i=1}^k P_i = S$ and for all $P_i, P_j$, $i \neq j$, $P_i \cap P_j = \emptyset$. Partition refinement is a technique to order the elements of a set using a specific type of refinement. In general, given a set $S$, and a partition $P = (P_1, P_2, \ldots, P_k)$ of $S$, a subset $T \subseteq S$ refines $P$ if each $P_i$ is replaced with $A_i = P_i \cap T$ and $B_i = P_i \setminus T = P_i \setminus A_i$.

A partition is ordered if the ordering of the partition classes, the $P_i$’s, matters. I.e., for $P = (P_1, P_2, \ldots, P_k)$, the elements of $P_i$ appear in $P$ before the elements of $P_j$ for $i < j$.

If a partition $P = (P_1, \ldots, P_k)$ is ordered, and $T$ is the subset used to refine $P$, then the newly created partition is ordered as well and respects the ordering of $P$. This means every $P_i$ is replaced by $A_i = P_i \cap T$ and $B_i = P_i \setminus T$, but $A_i$ must appear before $B_i$ and for any $i < j$, $A_i, B_i$ appear before $A_j, B_j$. Thus the newly refined and ordered partition is $P = (A_1, B_1, A_2, B_2, \ldots, A_k, B_k)$.

For the purpose of our work, we focus on partition refinement when used on graphs. In particular, we look at the scenario that uses partition refinement to produce an ordering of the vertices. Formally, given a graph $G(V, E)$, we initialize the ordered partition $P$ to be all of $V$, $P = (V)$. The vertex $x$ whose open neighbourhood $N(x)$ refines $P$ is called a pivot, i.e., $N(x)$ plays the role of the set $T$ defined above. Pivots are selected one at a time, from a partition class that 1. is not empty, 2. contains a vertex who was not a pivot before, and 3. is the leftmost such partition in $P$. Ties are broken arbitrarily if there is more than one eligible pivot, i.e., this leftmost non-empty partition class has more than one element. When vertex $x$ is selected as a pivot from partition class $P_i$, $x$ is removed from $P_i$ into a new single-element partition class $\{x\}$ then $N(x)$ refines all partition classes $P_j$ for $j \geq i$ to create the ordered partition $P = (\ldots, \{x\}, P_i \cap N(x), P_i \setminus N(x), P_{i+1} \cap N(x), \ldots, P_k \cap N(x), P_k \setminus N(x))$.

We illustrate the first two iterations of this process. Initially $P = (V)$, all vertices are tied and are eligible to be pivots. We start by selecting a start vertex $v_1$. We use the neighbourhood of $v_1$, $N(v_1)$, to refine $P$ as described above, i.e., $T = N(v_1)$. This first step
of the refinement creates the ordered partition \( \mathcal{P} = (\{v_1\}, A_1 = N(v_1), B_1 = V \setminus N[v_1]) \). If one of the newly created partition classes is empty, it is then removed from \( \mathcal{P} \). The next step consists of choosing a new pivot as the leftmost available vertex in \( \mathcal{P} \) that has not been a pivot yet. Suppose that \( N(v_1) \neq \emptyset \), then the next pivot must be a neighbour of \( v_1 \) chosen arbitrarily from \( A_1 \). Call this neighbour \( v_2 \). The refinement creates the ordered partition \( \mathcal{P} = (\{v_1\}, \{v_2\}, A_1 \cap N(v_2), A_1 \setminus N(v_2), B_1 \cap N(v_2), B_1 \setminus N(v_2)) \). If on the other hand, \( N(v_1) \) is empty, i.e., \( A_1 \) is empty, then \( v_2 \) is selected from the leftmost non-empty partition class that does not consist of a previously chosen pivot. Thus \( v_2 \) would have been selected from \( B_1 \).

In general, suppose after \( k \) iterations of this process, we have the ordered partition \( \mathcal{P} = (\{v_1\}, \{v_2\}, \ldots, \{v_k\}, P_1, P_2, \ldots, P_t) \), where \( v_1, \ldots, v_k \) are the pivots of the previous \( k \) iterations. The next vertex to be a pivot is a vertex in the leftmost non-empty partition class in \( \mathcal{P} \) that does not consist of a pivot. Suppose without loss of generality that \( P_1 \) is non-empty. Then \( v_{k+1} \), the next pivot, is selected from \( P_1 \) and its open neighbourhood \( N(v_{k+1}) \) is used to refine \( P_1, \ldots, P_t \); thereby creating a newly refined ordered partition \( \mathcal{P} = (\{v_1\}, \ldots, \{v_k\}, \{v_{k+1}\}, P_1 \cap N(v_{k+1}), P_1 \setminus N(v_{k+1}), P_2 \cap N(v_{k+1}), P_2 \setminus N(v_{k+1}), \ldots, P_t \cap N(v_{k+1}), P_t \setminus N(v_{k+1})) \).

This process is repeated until we either have used every vertex as a pivot, or all the partition classes are singletons. When a vertex \( v \) is a pivot, we say that \( v \) is “pulling” its neighbours and “pushing” its non-neighbours. The ordering in which the vertices were used as pivots is the vertex ordering we are interested in. That is, at the end of this process, we have created a partition that looks like \( \mathcal{P} = (\{v_1\}, \{v_2\}, \ldots, \{v_n\}) \). The corresponding vertex ordering of this partition is the ordering \( \sigma = v_1, v_2, \ldots, v_n \). Before going further, we illustrate this idea using the graph in Fig. 2.5 redrawn below.

<table>
<thead>
<tr>
<th>Pivot</th>
<th>( \mathcal{P} )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a, b, c, d, e )</td>
<td>Initially ( \mathcal{P} = (V) )</td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td>( a, b, c, d, e )</td>
<td>( a ) is selected as a pivot</td>
</tr>
<tr>
<td>( b )</td>
<td>( a, b, c, d, e )</td>
<td>All classes are singletons now</td>
</tr>
</tbody>
</table>

Figure 2.10: An in situ partition refinement example.

We said that when vertex \( x \) is a pivot, \( x \) pulls its neighbours and pushes its non-
neighbours. However, notice in the example above that when \( b \) was a pivot, \( b \) did not pull all its neighbours first, in particular \( b \) did not pull \( e \) before \( d \) despite having \( be \in E, bd \notin E \). This is because this refinement does not re-order the partition classes. This is known as an **in–situ** refinement: We refine “in place” without moving the partition classes around.

Suppose now that we do allow the re-ordering of the classes. In particular, suppose at iteration \( k \), we have \( \mathcal{P} = (\{v_1\}, \{v_2\}, \ldots, \{v_k\}, P_1, P_2, \ldots, P_t) \), and \( v_{k+1} \) is the next pivot to be selected from \( P_1 \). When \( N(v_{k+1}) \) is used to refine \( \mathcal{P} \), we first place all the \( P_1 \cap N(v_{k+1}), P_2 \cap N(v_{k+1}), \ldots, P_t \cap N(v_{k+1}) \) before all of \( P_1 \setminus N(v_{k+1}), P_2 \setminus N(v_{k+1}), \ldots, P_t \setminus N(v_{k+1}) \), while still maintaining \( i \leq j \). The partition \( \mathcal{P} \) then becomes \( (\{v_1\}, \ldots, \{v_k\}, P_1 \cap N(v_{k+1}), \ldots, P_t \cap N(v_{k+1}), P_1 \setminus N(v_{k+1}), \ldots, P_t \setminus N(v_{k+1})) \). This is also a partition refinement that creates a different ordering of the vertices. This refinement is known as a **non-in–situ** refinement, since the pivot has the power to “pull” all its neighbours first before “pushing” its non-neighbours. The partition \( \mathcal{P} \) is still ordered, in the sense that at the end of the refinement, the single-element classes we end up with define the ordering \( \sigma \) we are interested in. We illustrate this non-in–situ refinement on the same graph below. Notice for example when \( b \) is the pivot, the partition class of \( e \) was moved in front of the one for \( d \).

<table>
<thead>
<tr>
<th>Pivot</th>
<th>( \mathcal{P} )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a, b, c, d, e )</td>
<td>Initially ( \mathcal{P} = (V) )</td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td>( a ) ( b, c, d ) ( e )</td>
<td>( a ) is selected as a pivot</td>
</tr>
<tr>
<td>( b )</td>
<td>( a ) ( b ) ( c ) ( c ) ( d )</td>
<td>( b ) pulls all its neighbours first.</td>
</tr>
<tr>
<td>( a, b, c, e, d )</td>
<td>The corresponding ( \sigma ) we get</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.11: A **non-in–situ** partition refinement example.

We call both refinements described above as **forward**, meaning that the pivots are selected by always choosing an eligible pivot from the leftmost available partition class. We move from left to right choosing pivots from the appropriate partition class.

Partition refinement has led to a number of simple algorithms, two of them in particular are **LexBFS** and **LexDFS**. As shown in [41], the ordering produced by the **in–situ forward** partition refinement on a graph \( G \) is a **LexBFS** ordering, whereas the ordering produced by the **non-in–situ forward** partition refinement is a **LexDFS** ordering. As a matter of fact, the orderings produced in Fig. 2.10 and 2.11 are **LexBFS** and **LexDFS** orderings of the same graph respectively. One of the reasons **LexBFS** is easy to implement in linear time...
is precisely due to its in-situ refinement [41]. The reshuffling of the partition classes in the non-in-situ refinement is a bottleneck to getting a linear time algorithm to compute a LexDFS ordering on arbitrary graphs. In fact, to the best of our knowledge, there is no linear time implementation of LexDFS. The best known algorithm takes $O(\min(n^2, n + m \log \log n))$ time and uses the above explained non-in-situ partition refinement together with van Emde Boas trees [71]. In one of our results, we give a linear time algorithm to compute a LexDFS ordering on cocomparability graphs. To do so, we introduce a backward in-situ partition refinement of the layers of a corresponding poset of $G$, without computing the poset itself.

Suppose at iteration $k$ of the refinement, we have the following partition $P = (\{v_1\}, \{v_2\}, \ldots, \{v_k\}, P_1, P_2, \ldots, P_t)$. Clearly, the ordering in which the pivots were processed is forward: $v_1$ before $v_2$ ... before $v_k$. Without worrying for now about how to get a partition $P$ in this format $(\{v_1\}, \{v_2\}, \ldots, \{v_k\}, P_1, P_2, \ldots, P_t)$, suppose we were given such a partition and were asked to refine $P_1 \ldots P_t$ by first using $v_k$ as a pivot, then $v_{k-1}, \ldots$ then $v_1$. We call this refinement backward. We introduce this backward refinement in the next chapter. We will formalize this idea when we need it, and we will explain how to initially obtain a non-empty partition $P$ with pivots to process in reverse order.
Chapter 3

Linear Time LexDFS on Cocomparability Graphs

Overview. In this chapter, we present the first linear time algorithm to compute a LexDFS cocomparability ordering. I.e., an ordering that is both a LexDFS ordering and a cocomparability ordering. To achieve this running time, we introduce a new partition refinement technique, as well as expose the poset structure of cocomparability graphs as seen through the lens of graph searching. In the next chapter, we tie this result to what LexDFS is actually doing on a poset. The results in this chapter are based on joint work with Ekkehard Köhler [47].

3.1 Overview of the Algorithm

Before presenting our algorithm in detail we first give the intuition behind what the algorithm is doing, and an overview of the step by step execution. One of the main take aways from this algorithm and its structure is that cocomparability graphs have LexDFS orderings that produce clique covers. By looking at the output of the algorithm from the poset perspective, one sees that LexDFS is traversing the poset layer by layer, where the algorithm visits all the maximal elements of the poset first, before “shelling” them and moving the next new set of maximal elements, and so on. One way to produce such specific orderings is by starting with a cocomparability ordering, and extracting these poset layers in a clever way. To do so, we assign a label to each vertex in a cocomparability ordering. Intuitively, this label, say at vertex $x$, just counts the number of upper cover elements of vertex $x$. Clearly, the layer of maximal elements is easy to compute since it has no upper covers. Once the maximal elements are identified, we “remove” them by updating the labels of the remaining elements, and repeat the same process, until we have removed all the elements...
one layer at a time. Since every layer in a poset induces an antichain, this implies that every layer is a clique in our cocomparability graph. And thus, at the end of this process, we have created a clique cover of the graph. The next step of the algorithm is then to reorder the elements within every layer in a way that obeys the lexDFS nature of the search. This is the step where we use our modified partition refinement technique. Throughout the algorithm, we exploit the structure of both cocomparability graphs and their corresponding posets, which allows us to complete all the steps of the algorithm in linear time.

We formalize these ideas next. Let $G(V,E)$ be a cocomparability graph. We first compute a cocomparability ordering $\sigma$ using the algorithm in [58]. Then, we assign a label, denoted $\#(v)$, to each vertex $v$ in $\sigma$. 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 $G$ defined by $\sigma$. Roughly speaking, we then partition $V$ by iteratively placing vertices with smallest label into the same partition set. In a comparability graph one could find 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 $P$ of $V$ where at each step $i$, the partition class $P_i$ consists of the vertices of minimum label value. Note that this is not the refinement step, we first just partition the vertices into buckets according to their label value. When a vertex $v$ is added to a partition class $P_i$, we say that $v$ has been visited.

Since $\sigma$ is a cocomparability ordering of $G$, and thus a linear extension of a poset on $\overline{G}$, 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 $\sigma$ in $\overline{G}$. Similarly, when all the $P_1$ vertices have been visited, i.e., ‘removed from the maximal elements of the poset’, $P_2$ is just the set of maximal elements in the partial order of $\overline{G}\setminus P_1$, and so on. Creating the initial partition classes is indeed equivalent to removing the maximal elements of a poset corresponding to $\overline{G}$ one layer at a time. We will denote this initial partition by $\pi$ in the following.

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, the set $N(v)$ is used to perform an in situ refinement on $P_i$. We use $\tau_i$ to denote the final (refined) ordering of $P_i$. When all partition classes have been refined, we concatenate all the $\tau_i$’s in order, i.e., $\tau = \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_p$, and use $\tau$ to denote the final ordering. We will show that $\tau$ is a lexDFS cocomparability ordering of $G$ that can be computed in linear time. Our main theorem is the following:
Theorem 3.1. Let $G(V, E)$ be a cocomparability graph, a LexDFS cocomparability ordering of $G$ can be computed in $O(m + n)$ time.

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

### 3.2 Vertex Labeling

Let $G(V, E)$ be an undirected cocomparability graph. Let $\sigma = v_1 \prec_\sigma v_2 \prec_\sigma \ldots \prec_\sigma v_n$ be a cocomparability ordering of $G$ returned by the algorithm in [58]; recall we use $\sigma = \text{ccorder}(G)$ to denote such an algorithm. For every vertex $v \in V$, we assign a label $\#(v)$ initialized to the number of non-neighbours of $v$ to its right in $\sigma$: $\#(v) = |\{u|uv \notin E \text{ and } v \prec_\sigma u\}|$.

Given such a labelling of the vertices, we create a partition of $V$ denoted by $P = (P_1, P_2, \ldots, P_p)$ 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 as 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 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 PartitionClasses is a formal description of the algorithm which takes a cocomparability ordering $\sigma$ of a graph $G(V, E)$ as input, and returns the partition $P = (P_1, P_2, \ldots, P_p)$. 

Algorithm 3.1 PartitionClasses

**Input:** a cocomparability ordering $\sigma$

**Output:** partition $\mathcal{P}$ of $V$ with $p$ partition classes, and $p$

1. $S \leftarrow \emptyset$
2. for $i \leftarrow n$ downto 1 do
3. $\#(v_i) \leftarrow (n - i) - |S \cap N(v_i)| \quad \triangleright \text{Initial labelling } \#(v)$
4. $S \leftarrow S \cup \{v_i\}$
5. end for
6. $U \leftarrow V \quad \triangleright U \text{ the set of unvisited vertices}$
7. $i \leftarrow 1$
8. while $U$ not empty do
9. $P_i \leftarrow \{v | \#(v) = \min(\#(U))\} \quad \triangleright \text{Creating Partition Classes}$
10. $U \leftarrow U \setminus P_i$
11. for $v \in P_i$ do
12. for $u \in U$ and $uv \in E$ do
13. $\#(u) \leftarrow \#(u) + 1$
14. end for
15. end for
16. $i \leftarrow i + 1$
17. end while
18. $p \leftarrow i - 1$
19. return $\mathcal{P} \leftarrow (P_1, P_2, ..., P_p)$ and $p$

Let $\pi = P_1 \cdot P_2 \cdot ... \cdot P_p$ be the order of $V$ resulting from Algorithm PartitionClasses such that $\forall x \in P_i, y \in P_j, j > i$, we have $x \prec_\pi y$. The order inside each $P_i$ is arbitrary. As sketched in the beginning of this section, these sets form a partition of our vertex set into anti-chains of the corresponding poset, such that for all vertices in such a set $P_i$, the length of the longest chain succeeding these vertices is $i$.

Consider the graph $G$ in Fig. 3.1 with a valid cocomparability vertex ordering. The numbers below the vertices are their labels as computed by Algorithm PartitionClasses on $G$, which would 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 the table in Fig. 3.2.
In the remainder of this chapter, we will use \( \#^*(v_i) \) to refer to the initial value of \( v_i \)'s label, i.e., the number of non-neighbours of \( v \) to its right in \( \sigma \); and \( \#_k(v_i) \) to denote the label value of \( v_i \) when \( P_k \) is being created, i.e., at iteration \( k \).

### 3.3 Partition Refinement

Once all the initial partition classes are computed, we reorder the adjacency list of each \( v \) according to \( \pi \) in \( O(m + n) \) time (cf. Section 3.6), then construct a new ordering of \( V \) by refining \( P \). Our refinement (algorithm \texttt{Refine}) is slightly different than the generic partition refinement algorithm presented in [41] and explained in Chapter 2.

Given the partition \( P = (P_1, P_2, ..., P_p) \) returned by Algorithm \texttt{PartitionClasses} 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 \textit{processed} when it has been refined. We use \( \tau_i \) to denote the final ordering of \( P_i \) after it has been refined, i.e., \( \tau_i = \texttt{Refine}(P_i, S_i) \). If a partition class \( P_i \) has an empty pivot set \( S_i \), then for \( \tau_i \) the (arbitrary) ordering of \( P_i \) in \( \pi \) is used.
Algorithm 3.2 Refine

**Input:** a partition class $P$ ordered by $\pi$ and its corresponding ordered list of pivots $S$

**Output:** refinement $\tau$ 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 \textcolor{Gray}{\textgreater 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.        $Q_{j+1}' \leftarrow Q_i \setminus N(v)$
12.        $j \leftarrow j + 2$
13.     end if
14.   end for
15.   $k \leftarrow j - 1$
16.   for $i \leftarrow 1$ to $k$ do \textcolor{Gray}{\textgreater Rename the new partitions for the next pivot}
17.      $Q_i \leftarrow Q_i'$
18.   end for
19. end while
20. return $\tau \leftarrow Q_1 \cdot Q_2 \cdot \ldots \cdot Q_k$ \textcolor{Gray}{\textgreater}

$x \in Q_i, y \in Q_j, j > i \implies x \prec_\tau y$ and $x, y \in Q_i \implies x \prec_\tau y$ iff $x \prec_\pi y$

The sets $S_i$ are implemented as stacks and are created as follows: $S_1 = \emptyset$ and $P_1$ is considered processed. For all $P_i, i > 1$, we scan $\tau_1$ from left to right and for each $v \in \tau_1$ and every $u \in P_i, i > 1$ where $uv \in E$, we push $v$ in $S_i$. In general, every time a partition class $P_j$, for $j < i$, is refined, i.e., $\tau_j$ has been produced, we scan $\tau_j$ from left to right, and for every $v \in \tau_j$ with neighbours in $P_i, i > j$, 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 $P_i \setminus N(v)$. The next pivot out of $S_i$ performs an in situ refinement of the current subpartitions of $P_i$.

Let $u_j^1, u_j^2, \ldots, u_j^k$ be the left to right ordering of the vertices inside $\tau_j$. We previously mentioned that not only is this refinement in situ, but also backwards. Backwards in two ways: First, the pivots of $\tau_j$ have a higher priority, i.e., a stronger pull, than the pivots of $\tau_k, k < j$, and second the pivots are pushed down the stack $S_i$ in $\tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_{i-1}$ order (left to right) and thus are popped in reverse order. Therefore we maintain the priority of the pivots in the backward order: $(\tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_{i-1})^d$, but also the priority of the pivots inside each $\tau_j, j < i$ in the backward order $\tau_j$. 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 $u_j^a$, for $a < b$. Note that
Chapter 3. Linear Time LexDFS on Cocomparability Graphs

Algorithm 3.3 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
   \( ♦ \) processed in the ordering of \( τ_j \)
2: if \( v \) has neighbours in \( P_i, i > j \) then
3: \( S_i.push(v) \)
   \( ♦ \) Update the pivot list of \( P_i \)
4: end if
5: end for

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. In particular, one can view the concatenation of \( τ_1 \cdot τ_2 \cdot τ_3 \) as the ordering \( φ \) described at the end of Chapter 2, that was processed left to right. Whereas 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.

Algorithm 3.2, Refine, takes \( P_i \) and \( S_i \) as input, and returns the new ordering \( τ_i \) of \( P_i \). Algorithm 3.3, UpdatePivots, takes \( τ_j \), the refined ordering of \( P_j \), as input and updates the stacks \( S_i \) for all unprocessed partition classes \( P_i, i > j \).

Let \( τ \) denote the final ordering of all the refined partition classes, i.e., \( τ = τ_1 \cdot τ_2 \cdot \ldots \cdot τ_p \). Using the graph again in Fig. 3.1, we show in Table 3.1 the in situ refinement of each \( P_i \). The final ordering is \( τ = h, k, j, i, g, f, d, e, b, c, a \).

<table>
<thead>
<tr>
<th>( P_i ) ordered by ( π )</th>
<th>( S_i )</th>
<th>Pivots</th>
<th>Refinement</th>
<th>( τ_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 ): {h, k}</td>
<td>0</td>
<td>-</td>
<td>(h, k)</td>
<td>h, k</td>
</tr>
<tr>
<td>( P_2 ): {j}</td>
<td>h</td>
<td>h</td>
<td>(j)</td>
<td>j</td>
</tr>
<tr>
<td>( P_3 ): {g, i}</td>
<td>h</td>
<td>h</td>
<td>((i)(g))</td>
<td>i, g</td>
</tr>
<tr>
<td>( P_4 ): {d, e, f}</td>
<td>g, h</td>
<td>g</td>
<td>((d, f)(e))</td>
<td>f, d, e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h</td>
<td>((f)(d)(e))</td>
<td></td>
</tr>
<tr>
<td>( P_5 ): {c, a, b}</td>
<td>e, d</td>
<td>e</td>
<td>((b)(c, a))</td>
<td>b, c, a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
<td>((b)(c, a))</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: The refinement of the \( P_i \)'s constructed in the table in Fig. 3.2

3.4 The Complete Algorithm

We are now ready to present the complete algorithm CCLexDFS.
### Algorithm 3.4 CCLexDFS

**Input:** a cocomparability graph \( G(V,E) \)  
**Output:** a LexDFS order \( \tau \) of \( G \) that is also a cocomparability order of \( G \)  

1. \( \sigma \to \text{ccorder}(G) \)  
2. \( \tau \leftarrow \emptyset \)  
3. \( (\mathcal{P},p) \leftarrow \text{PartitionClasses}(\sigma) \) ↩ Compute the partition classes corresponding to \( \sigma \)  
4. \( S_1,\ldots,S_p \leftarrow \emptyset,\emptyset,\ldots,\emptyset \)  
5. for \( i \leftarrow 1 \) to \( p \) do  
6. \( \tau_i \leftarrow \text{Refine}(P_i,S_i) \) ↩ Refine the partition classes  
7. \( \text{UpdatePivots}(\tau_i,i) \) ↩ Update the pivot sets  
8. \( \tau \leftarrow \tau \cdot \tau_i \)  
9. end for  
10. return \( \tau \)

## 3.5 Correctness of the Algorithm

Throughout this section, let \( \sigma \) be the cocomparability ordering of \( V \), and \( \mathcal{P} = (P_1,P_2,\ldots,P_p) \) the partition of \( V \) returned by Algorithm \text{PartitionClasses}. We denote the set of partition classes \( P_1 \) to \( P_{i-1} \) by \( \mathcal{P}_i = (P_1,P_2,\ldots,P_{i-1}) \). It is important to keep in mind that \( \sigma \) the cocomparability ordering of \( G(V,E) \) is just a linear extension of a poset \( P(\prec,V) \) defined on \( G \).

**Lemma 3.2.** 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.** Algorithm \text{PartitionClasses} takes as input a cocomparability ordering \( \sigma \). Notice first that for any vertex \( w \) with \( v \prec_{\sigma} w \), if \( uw \notin E \) then \( wu \notin E \). This is because \( u \prec_{\sigma} v \) and \( uv \notin E \), and if \( uw \in E \), then the edge \( uw \) would fly over vertex \( v \), which would contradict \( \sigma \) being a cocomparability ordering. 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 an earlier partition class \( P_j \in \mathcal{P}_i \) where \( j < i \) must have closed the gap in \( \#^*(u) > \#^*(v) \) by contributing to \( \#_j(v) \) but not to \( \#_j(u) \). Let \( z \in P_j, j < i \) 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 \( \#_j(v) \), thus reducing its difference from \( \#_j(u) \). Notice that \( u \prec_{\sigma} z \), otherwise \( zv \) flies over \( u \) which contradicts \( \sigma \) being a cocomparability ordering. Therefore \( z \) contributes one to \( \#^*(u) \) since it is a non-neighbour of \( u \) to its right in \( \sigma \), but \( z \) does not contribute to \( \#^*(v) \) since \( zv \in E \). Thus, when placing \( z \) in \( P_j \), \( z \) could not have closed the gap between \( u \) and \( v \)’s labels since its...
contribution to $\#_j(v)$ is cancelled out by $z$’s contribution to $\#^*(u)$. And so $\#_i(u)$ remains greater than $\#_i(v)$. \qed

**Lemma 3.3.** For $1 \leq i \leq p$, $P_i$ is the set of maximal elements in the poset $P \setminus \bigcup_{j=1}^{i-1} P_j$.

*Proof.* The proof is by induction on the partition classes $P_i$. $P_1$ is the set of vertices that are adjacent in $G$ to all the vertices to their right in $\sigma$. Since $\sigma$ is a cocomparability ordering, it is a linear extension of a poset $P(V, \prec)$ defined on $\overline{G}$. Thus $P_1$ is the set of elements in $P$ that are incomparable to all the elements to their right in the linear extension $\sigma$ 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 $P \setminus \bigcup_{j=1}^{i-1} P_j$. In particular, let $v$ denote an element in $P_i$ that is not maximal in $P \setminus \bigcup_{j=1}^{i-1} P_j$. This means there exists an element $u$ where $v \prec u$ and $u$ is an element of $P \setminus \bigcup_{j=1}^{i-1} P_j$. Since $\sigma$ 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 3.2, $\#_i(v) > \#_i(u)$ and thus $v \notin P_i$. \qed

**Lemma 3.4.** Every partition class $P_i \in \mathcal{P}$ returned by Algorithm *PartitionClasses* is a clique.

*Proof.* By Lemma 3.3, every $P_i$ is the set of the maximal elements in the poset $P \setminus \bigcup_{j=1}^{i-1} P_j$. 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 $G$. \qed

**Lemma 3.5** (The Flipping Lemma). [12, 13, 59] Let $\sigma$ be a cocomparability order, and $\tau$ the corresponding ordering created from $\sigma$ and returned by Algorithm *CCLexDFS*. Then for every $uv \notin E$, $u \prec_\sigma v$ if and only if $v \prec_\tau u$.

*Proof.* As we are assigning vertices to their partition classes, let $u$ and $v$ be the leftmost pair of vertices in $\tau$ to satisfy $uv \notin E$ and $u \prec_\sigma v$ and $u \prec_\tau v$: this means let $u$ be the leftmost vertex in $\tau$ for which there exists a non-neighbour $w$ such that $u \prec_\sigma w$ and $u \prec_\tau w$. Of all these non-neighbours $w$, let $v$ be the leftmost one of them in $\tau$.

By Lemma 3.2, $\#^*(u) > \#^*(v)$. And by Lemma 3.4, $u$ and $v$ belong to two different partition classes; $P_i$ and $P_j$, $j > i$ respectively since $u \prec_\tau v$. Therefore when $P_i$ was created $\#_i(u) < \#_i(v)$, which contradicts Lemma 3.2. Therefore $v \prec_\tau u$.

For sufficiency, suppose $v \prec_\tau u$. Since $uv \notin E$, by Lemma 3.4, $u, v$ belong to different partition classes. Since $v \prec_\tau u$, it follows that $v \in P_i$ and $u \in P_j$ for $i < j$. In particular, this means at iteration $i$:

$$\#_i(v) < \#_i(u) \quad (\dagger)$$
For sake of contradiction, suppose $v \prec_{\sigma} u$, then Lemma 3.2 implies $\#_i(v) > \#_i(u)$; thereby contradicting (†).

\begin{proof}
Corollary 3.6. $\tau$ is a cocomparability order of $G$.

Proof. As shown in [12], if $\tau$ 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 (Lemma 3.5) we have $y \prec_{\sigma} x$ and $z \prec_{\sigma} y$, which implies that the umbrella $zx$ flies over $y$ in $\sigma$, contradicting the fact that $\sigma$ is a cocomparability order. \hfill \Box

We are now ready to prove the first half of our main theorem, Theorem 3.1, which we restate as a lemma below:

\begin{lemma}
The ordering $\tau$ produced by Algorithm CCLexDFS is a LexDFS cocomparability ordering of $G$.
\end{lemma}

\begin{proof}
By Corollary 3.6, we know that $\tau$ 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 does not exist a vertex $d$ between $a$ and $b$ as required by the LexDFS 4 Point Condition (Theorem 2.11). Since $\tau$ 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 $a \in P_j, j < i$. 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 $c \in P_j, j > i$, 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 $\tau$ that does not satisfy the LexDFS 4 Point Condition (Theorem 2.11). This means $a$ is the leftmost vertex in $\tau$ for which there exists a pair $b, c$ of vertices such that $ac \in E, ab \notin E$ and there does not exists a vertex $d$ that satisfies the condition of Theorem 2.11. Of all eligible pairs $b, c$ that satisfy this construction, choose vertex $b$ to be the leftmost such vertex in $\sigma$, and then choose vertex $c$ to also be the leftmost such vertex, if there is more than one vertex that plays the role of $c$. Now consider the vertices that have contributed to $\#_i(b)$ and $\#_i(c)$. Let $u$ be one of these vertices. Thus either $u$ increased $\#_i(b), \#_i(c)$ by:

...
Case (i): being a non-adjacent right neighbour of $b$ or $c$ in the initial ordering $\sigma$ (Algorithm \texttt{PartitionClasses} Step 3), or

Case (ii) $u$ changed $\#(b), \#(c)$ when $u$ was assigned to a partition class (Algorithm \texttt{PartitionClasses} Steps 11-15).

Case (i): 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 the \texttt{LexDFS} 4 Point Condition (Theorem 2.11); or $u \prec_{\tau} a$ in which case $u$ increments $\#_j(b)$ at iteration $j$ when $u$ was assigned to $P_j, j < i$, 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 $u$ is in a set in $\mathcal{P}_i$ and $uc \in E$; again not reducing the gap. Therefore $u$ must be in one of the remaining partitions we have not processed, i.e., $(P_i, P_{i+1}, \ldots, P_p)$.

Case (ii): 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 $\tau$ and contradicting Corollary 3.6. 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 \texttt{LexDFS}. Moreover, $au \in E$ since $\tau$ is a cocomparability order; and the triple $a, b, u$ must satisfy the \texttt{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 $a \prec_{\tau} w \prec_{\tau} b, wb \in E, au \notin E$; 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 \texttt{LexDFS} for the triple $a, b, c$, and if $wc \in E$, then the umbrella $wc$ flies over $u$, contradicting $\tau$ being a cocomparability order. Therefore there must always exists a vertex that satisfies the \texttt{LexDFS} ordering for $\#_i(b) < \#_i(c)$ to hold; and thus $\tau$ is a \texttt{LexDFS} cocomparability order of $G$. \hfill QED
3.6 Implementation and Running Time

Before delving into the details, we refer the reader to Appendix A for a fully developed example that illustrates the implementation. Recall that \( \#^*(v_i) \) refers to the initial value of \( v_i \)'s label, i.e., the number of non-neighbours of \( v \) to its right in \( \sigma \); and \( \#(v_i) \) is the label value of \( v_i \) as computed in Step 13 of Algorithm PartitionClasses.

**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 \( \sigma \) is also implemented as a doubly linked list. Every vertex \( v_i \) is represented with a node structure containing a variable \( \text{pos} \) to store \( v_i \)'s position \( i \) in \( \sigma \), a variable \( \#(v_i) \) indicating \( v_i \)'s label value, a pointer \( Bptr \) initialized to NULL to indicate \( v_i \)'s corresponding bin, a pointer \( Bpos \) to indicate \( v_i \)'s position in the bin, a pointer \( pc \) initialized to NULL to indicate \( v_i \)'s partition class, and a pointer \( pt \) to \( v_i \)'s adjacency list.

**Lemma 3.8.** Let \( G(V, E) \) be a cocomparability graph and \( \sigma \) a cocomparability ordering of \( G \). The algorithm PartitionClasses takes \( O(m + n) \) time to compute the partition classes of \( G \).

**Proof.** Consider Step 2: Computing the initial labels of the vertices 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

\[
\#^*(v_i) = n - i - \left| \bigcup_{v_j \in E \atop j>i} v_j \right|
\]

For each vertex \( v_i \), we scan its adjacency list in \( O(d_{v_i}) \) steps where \( d_{v_i} \) is the degree of \( v_i \). Therefore Steps 2 to 5 take \( O(m + n) \) time.

For the while loop in Steps 8-17, consider a set of bins \( B_0, \ldots, B_{n-1} \) together with corresponding counter variables \( c_0, \ldots, c_{n-1} \), 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. Every node in the doubly linked list has a pointer to the corresponding vertex it is referring to. The bins and their counters are stored in an array \( A \) of size \( n \), where the value stored at entry \( A[i] \) contains \( c_i \) for bin \( B_i \) and the head pointer to \( B_i \)'s doubly linked list. Once a vertex \( v \) is placed in a bin \( B_i \), its corresponding \( Bptr \) pointer points to \( A[i] \), and a new node, pointing to \( v \), is created and inserted to the front of \( B_i \)'s doubly linked list. Vertex \( v \)'s pointer \( Bpos \) then points to this newly created node.

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. In particular, the corresponding pointer \( Bptr \) of vertex \( v \) points
to bin $B_i$, i.e., to $A[i]$. Once all vertices are assigned to a bin, we use these bins to create the $p$ partition classes $P_1, \ldots, P_p$ 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 leftmost 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. To move vertex $u$ from bin $B_k$ to bin $B_{k+1}$, we use $Bpos$ to locate $u$ in $B_k$. Since $B_k$ is a doubly linked list, removing $u$ from $B_k$ takes constant time. We remove $u$ from $B_k$ and insert it to the front of $B_{k+1}$ – thus taking constant time. We update the $Bptr$ pointer of $u$ to point at $B_{k+1}$, and $Bpos$ to point to $u$’s new position in $B_{k+1}$. Therefore when processing a bin $B_i$, for each one of its vertices $v$, we do at most $O(d_v)$ increments and decrements for the neighbours of $v$, and thus creating the partition classes takes at most $O(m + n)$ time over all vertices.

Once all the partition classes are created, we reorder the adjacency list of each $v$ according to $\pi$ in $O(m + n)$ time. To do so, we create new adjacency lists as follows: We go through the vertices according to $\pi$’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 $\pi$. Fig. 3.3 illustrates this construction for vertex $g$ of the graph in Fig. 3.1 with neighbours restricted to partition class $P_4$.

![Figure 3.3](image-url)

**Figure 3.3**: The new adjacency list of $g$ restricted to $P_4$ as constructed in the table in Fig. 3.2.

**The Partition Refinement**: We implement $\text{Refine}$ using a similar data structure as the one presented in [41], where $\pi$ is a doubly linked list and each $P$ is implemented as a doubly linked list with three pointers: one to its first (f) element in $\pi$, one to its last (l), and one denoted c (for current), initialized to NULL. Every vertex $v$ in $\pi$ 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 \( \pi \), as it was just illustrated in Fig. 3.3.

**Lemma 3.9.** Algorithms *Refine* and *UpdatePivots* take linear time in the size of their inputs.

*Proof.* 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 3.3, *UpdatePivots* as follows: Every \( v \) has a pointer to the partition class it belongs to, and the adjacency lists are sorted according to \( \pi \), thus every time a partition class \( P_j \) is refined, we sweep through \( \tau_j \) from left to right, and for each \( v \in \tau_j \), with neighbours in \( P_i, i > j \), \( v \) is pushed onto \( S_i \). This operation takes at most \( O(d_v) \) 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 \( P_i \) for \( i > j \).

Now consider Algorithm *Refine*, Steps 5 to 15: 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 \( \pi \). For each \( w \) adjacent to \( v \) where \( w \) has its image in a subpartition \( Q_i \), we check if the pointer \( c \) of \( Q_i \) is NULL. If \( c \) is NULL, \( w \) is moved to the start of \( Q_i \) where the \( f \) pointer is updated accordingly, and \( c \) points to \( w \) as well. If \( c \) 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 subpartition has its \( c \) reset to NULL, and its first and last pointers updated. It is important to note that if a subpartition \( Q_i \) has \( f = l = c \), then we know that it has a singleton element and thus cannot be refined more; therefore its \( c \) pointer is not reset to NULL.

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 \( O(|N(v) \cap Q_i|) \) steps for every \( Q_i \) with \( c \neq \text{NULL} \). For each \( Q_i \), vertices between \( f \) and \( c \) are moved into a new subpartition \( Q_j \), and \( Q_i \) becomes \( Q_i \setminus Q_j \), and thus, creating \( Q_j \) and \( Q_i = Q_i \setminus Q_j \) takes at most \( O(|N(v) \cap Q_i|) \) steps. Since this operation is performed on every \( w \in N(v) \cap Q_i \), for every \( v \in S \), we perform at most \( O(d_v) \) steps of refinement, thus a total of the number of edges going from vertices in \( S \) to \( P \).

To conclude, algorithm *UpdatePivots* is linear in the number of edges between \( P_j \) and any class \( P_i, i > j \); and algorithm *Refine* is linear in the number of edges between \( S_i \) and \( P_i \). \( \square \)
We are now ready to prove the second half of our main theorem, Theorem 3.1, stated as a lemma below:

**Lemma 3.10.** Algorithm CCLexDFS takes $O(m + n)$ time.

*Proof.* Step 1 can be computer in linear time using the algorithm in [58]. By Lemma 3.8, Step 3 is computed in linear time. By Lemma 3.9, algorithm 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 5 to 9 in algorithm CCLexDFS, we get $\Sigma |S_i| \in O(m)$. The refinement is thus computed in $O(m + n)$ time. Similarly, we just showed that UpdatePivots is linear in the number of edges between $P_j$ and any class $P_i$, $i > j$; consequently looping over the $p$ partitions computed by algorithm PartitionClasses, UpdatePivots takes a total of $O(m + n)$ time. All the remaining steps in algorithm CCLexDFS are clearly linear. \[ \square \]

*Proof of Theorem 3.1.* The proof follows from Lemma 3.7 and Lemma 3.10. \[ \square \]

### 3.7 Variants of the Algorithm: LexDFS$^+$ & LexDFS($\overline{G}$)

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 in the size of the given graph.

The algorithms in [12, 60, 13, 61] all require LexDFS cocomparability orderings as input. One way to get such orderings is to compute a variant known as a LexDFS$^+$ cocomparability ordering. We show how to do so in linear time. Recall the $^+$ rule is defined as follows:

**Definition 3.11.** A LexDFS$^+$ is a variant of LexDFS that takes as extra input an ordering $\sigma$ of $V$, where ties amongst eligible vertices are broken by choosing the rightmost vertex with respect to $\sigma$, where $\tau = \text{LexDFS}^+(\sigma)$.

Therefore given a cocomparability order $\sigma$, LexDFS$^+(\sigma)$ always starts with $\sigma$’s rightmost vertex and breaks ties accordingly. Recall that when the partition classes were created in Algorithm PartitionClasses the vertices in each $P_i$ were arbitrarily ordered. Given an ordering $\sigma = v_1, v_2, ..., v_n$, recall $\sigma^d = v_n, v_{n-1}, ..., v_2, v_1$ is the reverse ordering of $\sigma$.

**Corollary 3.12** (of Theorem 3.1). Prior to the partition refinement step (Algorithm CCLexDFS Step 6), if the vertices inside each partition class were ordered according to $\sigma^d$, then the resulting $\tau$ is a LexDFS$^+(\sigma)$ ordering.

*Proof.* Let $\sigma^d = v_n, v_{n-1}, ..., v_2, v_1$ be the reverse ordering of $\sigma$ and $P_i = P_1, \ldots, P_{i-1}$ the partition created by Algorithm PartitionClasses at point $i$, i.e., the $i^{th}$ loop iteration of lines 8 to 17). Furthermore, suppose that the vertices inside each $P_i$ are ordered according to
Clearly the resulting ordering $\tau$ is a $\text{LexDFS}$ cocomparability ordering, since Theorem 3.1 holds for any arbitrary ordering inside the partition classes.

Suppose $\tau$ is not a $\text{LexDFS}^+$ ordering. That is, at some iteration $i$, there exist two tied vertices $u$ and $v$ where $u \prec_\sigma v$ but $u \prec_\tau v$. Nodes $u, v$ are tied if for all $P_j \in P$, $N(u) \cap P_j = N(v) \cap P_j$ and $u, v \in P_i$, i.e., they are ready to be processed. Since $u \prec_\sigma v$, and vertices inside $P_i$ are ordered according to $\sigma^d$, $v$ is left of $u$ inside $P_i$. Since $N(u) \cap P_i = N(v) \cap P_i$, there does not exist a pivot $w \in P_i$ to pull $u$ in front of $v$ when processing $P_i$. Therefore $v \prec_\tau u$. Consequently, $\tau$ is a $\text{LexDFS}^+$ ordering.

**Lemma 3.13.** $\text{LexDFS}^+$ can be implemented in $O(m + n)$ time.

**Proof.** To obtain a $\text{LexDFS}^+$ ordering, we need to reorder the partition classes to maintain $\sigma^d$’s order. Since every vertex has a pointer to the partition class it belongs to, it suffices to sweep through $\sigma$ from right to left and create the partition classes, which takes $O(n)$ time. Once this is done, the vertices in each partition are ordered according to their $\sigma^d$ ordering. This can be done in $O(n)$ time by sweeping through $\sigma^d$ and adding each vertex to the end of its partition class. The rest of the analysis remains the same.

Suppose now that the input graph $G$ is a comparability graph. One can compute in linear time a transitive orientation of $G$, and an ordering $\sigma$ that is a linear extension of the corresponding partial order using the algorithm in [58]. It is fairly straightforward to see that a $\text{LexDFS}(\overline{G})$ 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 correct neighbours (i.e., the non-neighbours in $\overline{G}$). More precisely, we first modify Algorithm $\text{PartitionClasses}$ to decrement the labels in Step 13, instead of incrementing them. Then in Algorithm 3.2, $\text{Refine}$, we pull the appropriate neighbours first, by swapping Steps 10-11:

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

**Theorem 3.14.** Let $G$ be a comparability graph. One can compute a $\text{LexDFS}$ cocomparability ordering of $\overline{G}$ in linear time in the size of $G$.

In [13], the authors gave a certifying $\text{LexDFS}$ based algorithm that computes a maximum independent set on a cocomparability graph as well as a minimum clique cover as a certificate; with Algorithm $\text{CCLexDFS}$ and Theorem 3.14, this certifying algorithm is a linear time certifying algorithm to compute a minimum colouring of a comparability graph together with a maximum clique as a certificate.
3.8 Conclusion

As one can see through the algorithm and the correctness proof presented in this chapter, LexDFS on cocomparability graphs can easily be seen through the corresponding poset lens. But what does LexDFS do on the poset, if anything? Can we use the cocomparability structure of posets, to come up with fast algorithms on posets? We answer this latter question affirmatively in the next chapter, where we present a new algorithm to compute linear extensions with minimum bump of a poset using the underlying LexDFS structure.
Chapter 4

The LexDFS Structure of Posets

Overview. In this chapter, we seek algorithms on the poset representation that utilize the methods and insights of what was discussed previously. We present here an elegant algorithm, GreedLex, to find an optimal minimum bump linear extension of a poset. Our algorithm is simple, easy to implement, and has short proofs of correctness and complexity analysis. It runs in $O(m + n \log w)$ time, where $w$ is the width of the poset. In contrast, the algorithms that achieve linear time are extremely complex to implement, and we are unaware of any attempts made to do so. The work in this chapter has not appeared yet, and is joint with Derek G. Corneil and Gara Pruesse.

4.1 Lexicographic labelings of Posets

Let $P$ be a poset that has a linear extension $L(P)$ with bump zero. This means every pair of contiguous elements in $L(P)$ are incomparable in $P$. From a cocomparability graph perspective $G(P)$, this means there is a total ordering of the vertices of $G$, where every pair of contiguous elements is adjacent. Thus $G(P)$ has a Hamilton path. In general, a linear extension that minimizes the number of bumps can be seen a path cover of a cocomparability graph that minimizes the number of path needed to cover $G$. With every new bump introduced, the current path ends, and a new one begins. Indeed, a minimum-bump ordering of the poset is equivalent to a minimum path cover of the associated cocomparability graph. Every cocomparability graph that has a Hamilton path has a zero-bump ordering in its associated posets, and vice versa. Hence algorithms for minimum bump orderings of posets can be used to compute minimum path covers on cocomparability graphs.

The Hamilton path problem on cocomparability graphs can be solved in linear time using the lifting framework, as shown by Corneil, Dalton, and Habib in [12]. The algorithm works by computing a LexDFS cocomparability ordering and greedily constructing a path
cover of $G$.

On the other hand, polynomial time algorithms to compute minimum bump orderings were first developed by Habib, Möhring and Steiner [42] in 1988. Around the same time, Schaffer and Simons showed how to modify an algorithm by Gabow [30] for the two-processor scheduling problem to compute the bump number in linear time [69]. The two-processor problem seeks to schedule unit length jobs with precedence on two identical machines. Thus, the fastest bump number algorithm builds off techniques from scheduling theory, and this running time is a result of a special Union-Find algorithm that runs in linear time when the operations are known in advance [31]. We know of no actual implementation of this linear time algorithm. The main result in this chapter is an easily-stated theorem for the bump-number that leads to an algorithm that is trivial to implement. Our algorithm runs in $O(m + n \log w)$ time, where $w$ is the width of the poset.

We begin with some (additional) notation used in this chapter. A multiset $S_1$ is said to be lexicographically larger than a multiset $S_2$ if $\max(S_1 \setminus S_2) > \max(S_2 \setminus S_1)$. We will write $S_1 \succ_{\text{lexico}} S_2$ to denote this relation.

**Definition 4.1.** For a poset $P(V, R)$, a lexicographic labeling $\text{lex} : V \to \mathbb{Z}$ is any labeling of the elements such that for any elements $u$ and $v$ in $V$ if the multiset of $\text{lex}$ numbers of the upper covers of $v$ is lexicographically greater than the multiset of $\text{lex}$ numbers of the upper covers of $u$ then $\text{lex}(u) < \text{lex}(v)$; that is

$$\{\text{lex}(v') : v' \succ v\} \succ_{\text{lexico}} \{\text{lex}(u') : u' \succ u\} \implies \text{lex}(v) > \text{lex}(u)$$

Note that a lexicographic labeling of a poset induces a lexicographic labeling of any filter or ideal of the poset. There are many lexicographic labelings of a poset. Two important types of lexicographic labelings are the following: The parsimonious labeling, which gives all maxima the label 1 and then ascribes the minimum possible label value to each element while adhering to the lexicographic rule; and the sequential labeling, which labels the $t$ maxima with numbers 1 to $t$, and ascribes labels lexicographically to the remaining elements without label duplication.

The sequential labelings impose an arbitrary ordering on the maximal elements; the remainder of the labeling is then essentially fixed by that order, except that ties must be broken when distinct elements have the same upper cover sets. Note that any sequential lex-labeling of a poset yields a total ordering of the elements – indeed, it is the reverse of a linear extension of the poset. In the remainder of the chapter, we will be using the sequential labeling. This labeling makes it particularly easy to tie LexDFS orderings on cocomparability graphs to lex orderings on posets.

Recall the difference between $u \prec v$ and $u < v$. The former says that $v$ is an immediate
cover of \( u \) in the poset (the transitive reduction), whereas the latter says that \( v \) is in the ideal of \( u \) (the transitive closure). Since we will be jumping between representations of the transitive closure and the transitive reduction, we need to confirm that lex labelings operate identically on the two representations. In other words, lex labeling using the set of lex labels of upper covers of each vertex \( v \) yields exactly the same labelings as using the set of lex labels of all \( w > v \). The following lemma makes this clear.

In the lemma and its proof, we call the set of all \( w > u \) the up-set of \( u \). Call a labeling of the elements of a poset a \( \text{lex}^+ \) labeling if it arises from lexicographically labeling the vertices using up-sets rather than cover sets to determine the ordering of the labels. I.e., a function \( \text{lex}^+ : V \to \mathbb{Z} \) is a \( \text{lex}^+ \) labeling if: for any elements \( u \) and \( v \), if the multiset of \( \text{lex}^+ \) numbers of the up-set of \( u \) is lexicographically greater than the multiset of \( \text{lex}^+ \) numbers of the up-set of \( v \) then \( \text{lex}^+(u) > \text{lex}^+(v) \).

**Lemma 4.2.** A labeling is a \( \text{lex}^+ \) labeling if and only if it is a lexicographic labeling.

**Proof.** Given any two elements \( u \) and \( v \) of the poset, if we consider the sequence, in non-decreasing order, of either the lex labels of their cover sets, or the \( \text{lex}^+ \) labels of their up-sets, either way the first label in which \( u \)'s sequence differs from \( v \)'s sequence must occur at the label of a cover.

\[ \square \]

### 4.1.1 Theorems about Lexicographic Labelings

It is easily seen that if \( \text{depth}(u) < \text{depth}(v) \) then \( \text{lex}(u) < \text{lex}(v) \) for any poset and any lexicographic labeling \( \text{lex} \); we state this in the following lemma, and provide a short proof.

**Lemma 4.3.** Let \( P \) be a poset with a lexicographic labeling \( \text{lex} \). For any two elements \( u \) and \( v \) in \( P \), if \( \text{depth}(u) < \text{depth}(v) \) then \( \text{lex}(u) < \text{lex}(v) \).

**Proof.** We show that any label at a given depth \( d \) is greater than any label at depth \( d - 1 \), by induction on depth \( d \). When \( d = 0 \), the claim is trivially true. For \( d > 0 \), let \( v \) be any element at depth \( d \), and \( u \) an element at depth \( d - 1 \). Note that \( v \) has an upper cover of depth \( d - 1 \), which by the inductive assumption has a larger lex label than any element covering \( u \) and therefore the multiset of lexlabels of upper covers of \( v \) is lexicographically larger than the multiset of lex labels of the upper covers of \( u \), and hence \( \text{lex}(v) > \text{lex}(u) \) by the definition of a lexicographic labeling.

\[ \square \]

The following lemma captures the significance of lex labelings for the algorithms that follow.

**Lemma 4.4.** [Lex Lemma] Let \( P \) be a poset with a lexicographic labeling \( \text{lex} \), and suppose \( \text{lex}(a) \geq \text{lex}(b) \), and there is an element \( c_1 \) such that \( a \parallel c_1 \) and \( b < c_1 \). Then there exists an element \( c_2 \) such that \( b \parallel c_2 \) and \( a < c_2 \), and where \( \text{lex}(c_2) \geq \text{lex}(c_1) \).
Proof. This follows from the definition of lexicographic labeling. \qed

Notice that the Lex Lemma is precisely the \texttt{LexDFS} $C_4$ Property (Property 2.14) on posets. One of the main useful properties of \texttt{LexDFS}, when seen on a poset, is that it computes a layering of a corresponding poset of $P(G)$ (Lemma 3.3). The next lemma ties \texttt{LexDFS} cocomparability orderings to lex labelings of the poset.

**Lemma 4.5.** Let $\sigma$ be a cocomparability ordering of a graph $G$ and let $P(G)$ be the poset constructed from $\sigma$. Let $\pi$ be the $\texttt{LexDFS}^+(\sigma)$, and $\tau$ the lex labeling of $P(G)$. Then $\pi$ is a lex labeling of $P(G)$ and $\tau$ is a \texttt{LexDFS} cocomparability ordering.

**Proof.** 1. Suppose first that $\pi$ is not a lex labeling, and choose a pair $a, b \in V$ such that $a \prec_{\pi} b$ but $\text{lex}(a) > \text{lex}(b)$. In particular choose $a$ and $b$ as the leftmost such pair in $\pi$ – that is $a$ is the leftmost element in $\pi$ for which such an element $b$ exists, and of all the eligible elements “$b$”, choose the one that is also leftmost in $\pi$. Let $A$ and $B$ be the up-sets of $a$ and $b$ respectively, and let $c = \max(A\setminus B)$ and $d = \max(B\setminus A)$. Then by the definition of lex labelings, if $\text{lex}(b) < \text{lex}(a)$ then $\text{lex}(d) < \text{lex}(c)$. And by the choice of $a, b$ as the leftmost “bad” pair in $\pi$, it follows that $d \prec_{\pi} c$. We thus have:

$$d \prec_{\pi} c \prec_{\pi} a \prec_{\pi} b$$

$$ad, bc \in E \text{ and } bd, ac \notin E \text{ (since we are in the complement).}$$

Therefore, the triple $c, a, b$ is a bad triple in $\pi$. Since $\pi$ is a \texttt{LexDFS} ordering, by the 4 Point Condition (Theorem 2.11), there must exist a vertex $e$ that is a private neighbour of $a$ with respect to $b$ (in $G$) such that $c \prec_{\pi} e \prec_{\pi} a$. Therefore $ea \in E, eb \notin E$ which implies $e \in B\setminus A$. However $d \prec_{\pi} c \prec_{\pi} e$, thus $\text{lex}(d) < \text{lex}(e)$, thereby contradicting $d$ as $\max(B\setminus A)$. Thus $\pi$ is a lex ordering.

2. Suppose $\tau$ is not a \texttt{LexDFS} ordering. Since $\tau$ is a lex ordering, it is a linear extension of $P(G)$, and thus a cocomparability ordering. If $\tau$ is not a \texttt{LexDFS} ordering, then there exists a bad triple $a \prec_{\tau} b \prec_{\tau} c$ that does not satisfy the \texttt{LexDFS}4 Point Condition: i.e., there does not exist a vertex $d$ such that $a \prec_{\tau} d \prec_{\tau} b$ such that $db \in E, dc \notin E$. Choose such a triple as the leftmost in $\pi$ (i.e., $a$ is the leftmost vertex for which there exists a $b$, the leftmost non-neighbour of $a$ such that there is a $c$ – leftmost – that is adjacent to $a$ and not $b$). Let $B$ and $C$ be the up-sets of $b$ and $c$ respectively. Thus $a \in B\setminus C$.

Since $b \prec_{\tau} c$, $\text{lex}(b) \leq \text{lex}(c)$, and thus $\max(B\setminus C) < \max(C\setminus B)$. Since $a \in B\setminus C$, it follows that $a \prec_{\tau} \max(C\setminus B)$. Let $e = \max(C\setminus B)$, then $a \prec_{\tau} e$. If $e \prec_{\tau} b$ then $e$ plays the role of vertex $d$ in the \texttt{LexDFS} 4 Point Condition above. Thus $b \prec_{\tau} e \prec_{\tau} c$, which implies $a, b, e$ is a bad \texttt{LexDFS} triple that is to the left of $a, b, c$. Thus by the choice of $a, b, c$, the triple $a, b, e$ satisfies the 4 Point Condition, and thus there exists a vertex $f$ such that $a \prec_{\tau} f \prec_{\tau} b$, \texttt{LexDFS}.
\( fb \in E, fe /\notin E \); if \( fc /\notin E \), then \( f \) plays the role of vertex \( d \) above. Thus \( fc \in E \), but then we have \( f \prec \tau, e \prec \tau, c \) and \( fe \in E, fe, ec /\notin E \); an umbrella in the complement, which contradicts \( \tau \) being a linear extension. Thus \( \tau \) is a LexDFS cocomparability ordering as well.

**Corollary 4.6.** Lex labelings can be computed in linear time.

**Proof.** We use the backward in-situ refinement in the previous chapter to do so.

### 4.1.2 The Bump Number Algorithm: GreedLex

The GreedLex Algorithm for finding a linear extension with the minimum bump number proceeds by first lexicographic labeling the poset, and then shelling minimal elements from the poset “greedily” (if possible, avoiding creating a bump involving the element currently being shelled) and then within the set of choices allowed, selecting one that is lexicographically greatest. The algorithm takes as input a poset \( P \) and a lex labelling of the elements of \( P \).

**Algorithm 4.1 GreedLex**

**Input:** A poset \( P(V, \prec) \) and \( \text{lex} : V \to \mathbb{Z} \) a lex labeling of \( V \)

**Output:** A linear extension \( \pi \) of \( P \) with minimum bumps, and \( \text{bump} \) the bump number for \( P \)

1: \( \pi \) is an initially empty shelling sequence.
2: \( \text{bump} \leftarrow 0 \)
3: while \( P \) is not empty do
4: \hspace{1em} if There are any minima of \( P \) that are not upper covers of the last-shelled node then
5: \hspace{2em} Select such a minimum that has maximum lex label
6: \hspace{2em} Remove it from \( P \) and add it to the end of \( \pi \)
7: \hspace{1em} else \hspace{1em} + The last-shelled node was a unique minimum of \( P \)
8: \hspace{2em} Select a minimum with maximum lex label
9: \hspace{2em} Remove it from \( P \) and add it to the end of \( \pi \)
10: \hspace{2em} \( \text{bump} \leftarrow \text{bump} + 1 \)
11: end if
12: end while
13: return \( \pi \) and \( \text{bump} \)

The engine of the proof of correctness of GreedLex for computing the bump number is the Lex-Yanking Lemma, given below.

**Proposition 4.7.** [The Lex-Yanking Property for Bump Number] Let \( P(V, R) \) be a poset with lexicographic labeling \( \text{lex} \). \( P \) has the Lex-Yanking Property for bump number if the following holds: If \( P \) has a linear extension that starts with \( b \) and has \( k \) bumps, and
there is a minimum element \( a \) of \( P \) such that \( \text{lex}(a) \geq \text{lex}(b) \), then \( P \) has a linear extension that starts with \( a \) and has \( k \) or fewer bumps.

**Lemma 4.8. [The Lex-Yanking Lemma for Bump Number]** All posets have the Lex-Yanking Property for bump number.

The implications of the Lex-Yanking Lemma are that, when constructing a minimum-bump linear extension by shelling the poset (that is, repeatedly removing minimal elements from \( P \), the shelling order being a linear extension of \( P \)), it is safe to select an element that has the greatest lex label if it is not a cover of the last-shelled element (i.e., if it does not introduce a bump).

It remains, then, to determine the minimal element to shell when the ones with maximum lexlabel are all upper covers of the most recently shelled element. The \texttt{GreedLex} algorithm, outlined above, selects an element of maximum lexlabel from among elements that do not cover the last-shelled element; if there are no such elements (i.e., the last-shelled element was a unique minimum of the remaining poset), then it selects the element with maximum lexlabel from among the minima, and a bump is introduced.

We call a linear extension constructed in this way a \texttt{GreedLex} ordering of the poset.

We will prove, as Lemma 4.8 states, that all posets have the Lex-Yanking Property. But first, we will show that if a poset \( P \) and all of its subposets have the Lex-Yanking Property, then any \texttt{GreedLex} ordering of the poset is a minimum-bump linear extension. By proving this first as Lemma 4.11, we will be allowed, in our inductive proof of the Lex-Yanking Lemma, to assume that when the inductive hypothesis can be applied to a smaller poset, then \texttt{GreedLex} orderings of the smaller poset have minimum bump number for that subposet. This simplifies the proof.

First, a few lemmas that identify how we can alter linear extensions and what the consequences are for the bump number.

**Lemma 4.9. [Yanking Lemma]** If \( P \) is a poset and \( w \) is a minimum element of \( P \), and there is a linear extension \( L \) with \( k \) bumps, then \( w \cdot L \setminus \{w\} \) is also a linear extension and has \( k + 1 \) or fewer bumps.

**Proof.** The lemma states that a minimal element can always be yanked to the front of the linear extension, with an increase of at most one to the bump number. Let \( L = x_1x_2 \ldots x_{i-1}wx_{i+1} \ldots x_n \). Then \( wx_1x_2 \ldots x_{i-1}x_{i+1} \ldots x_n \) is also a linear extension with the same bumps as \( L \) except possibly with one \((x_{i-1}, x_{i+1})\) created and possibly with one \((w, x_{i+1})\) destroyed. \( \square \)

**Lemma 4.10. [Swapping Lemma]** If \( P \) is a poset with \( a \) and \( b \) as minimal elements of \( P \), and there is a linear extension \( L \) with \( k \) bumps, then if \( a \) and \( b \) can be swapped without violating a precedence constraint then the resulting linear extension has \( k + 1 \) or fewer bumps.
Proof. If the swap does not violate precedence constraints, then both \( a \) and \( b \) are incomparable to all elements between them, the fact that they are minimal means they are incomparable to elements to the left of them. Hence swapping them can create or destroy only a bump involving the rightmost of \( a \) and \( b \) and its successor in the linear extension.

We now prove that the Lex-Yanking Property is sufficient to ensure that a linear extension produced by the \texttt{GreedLex} algorithm has the minimum number of bumps.

**Lemma 4.11.** If \( P \) is a poset such that the Lex-Yanking Property holds for it and all its induced subposets, then any \texttt{GreedLex} ordering of \( P \) has minimum bump number.

Proof. Let \( P \) be a poset that has the Lex-Yanking Property, and all its induced subposets have the Lex-Yanking property. Let \( L = x_1x_2...x_n \) be any \texttt{GreedLex} ordering of \( P \), and let \( k \) be the number of bumps in \( L \).

Let us call an ordering \( u_1u_2...u_i \) extensible if it is the prefix of some minimum-bump linear extension of \( P \). We are claiming that the prefixes \( x_1...x_i \) of \( L \) are all extensible. We proceed by induction on \( i \). The empty prefix is clearly extensible.

Suppose that \( x_1...x_i \) is extensible. If \( x_{i+1} \) does not exists, we are done, so hereafter in the proof we assume \( i < n \) and \( x_{i+1} \) exists.

**Case 1.** \( x_{i+1} \) has the maximum lexlabel of all remaining elements. There are two subcases. First, if \( x_i \parallel x_{i+1} \) (i.e., no bump is introduced directly after \( x_i \)): given that \( x_1...x_i \) is extensible, then \( x_1...x_{i+1} \) is extensible, by the Lex-Yanking Property. Second, we consider the case where \( x_i \prec x_{i+1} \). The fact that \texttt{GreedLex} selected \( x_{i+1} \), an upper cover of the last-shelled elements \( x_i \), implies that all remaining elements are comparable to \( x_i \), so any extension of \( x_1...x_i \), including the least-bump linear extension, will have a bump after \( x_i \), and by the Lex-Yanking Property of the induced subposet, \( x_{i+1} \) can be selected to extend the prefix.

**Case 2.** \( x_{i+1} \) does not have the maximum lexlabel of all remaining elements. Then those unshelled elements with higher lexlabel than \( x_{i+1} \) are comparable to \( x_i \) whereas \( x_{i+1} \) is not - this is a consequence of the \texttt{GreedLex} algorithm’s selection criteria.

Since \( x_1...x_i \) is extensible, there is a linear extension \( x_1...x_iy_{i+1}...y_n \) that realizes the minimum bump number of \( P \).

**Case 2(a).** If \( x_i \neq y_{i+1} \), then \( \text{lex}(y_{i+1}) \leq \text{lex}(x_{i+1}) \), and therefore by the Lex-Yanking Property, there is a linear extension of \( P \setminus \{x_1,...,x_i\} \) that starts with \( x_{i+1} \) and has no more bumps than \( y_{i+1}...y_n \); attaching this suffix to the prefix \( x_1...x_i \) yields a linear extension of \( P \) with the same or fewer bumps as \( x_1...x_iy_{i+1}...y_n \), which has the minimum number. Hence \( x_1...x_{i+1} \) is extensible.

**Case 2(b).** If \( x_i < y_{i+1} \), then there is a bump \( x_iy_{i+1} \). Note that \( x_{i+1} \) is a minimum of the poset \( P \setminus \{x_1...x_i\} \); therefore we can yank \( x_{i+1} \) out of \( y_{i+1}...y_n \) and place it in front
of \(y_{i+1}\); re-attaching to the prefix \(x_1 \ldots x_i\) results in the destruction of one bump and, by the Yanking Lemma, creating at most one bump. Hence the resulting linear extension has no more bumps than \(x_1x_2 \ldots x_iy_{i+1} \ldots y_n\), which has the minimum number of bumps of \(P\). Therefore \(x_1 \ldots x_i x_{i+1}\) is extensible.

Hence by induction, the \textbf{GreedLex} ordering \(x_1x_2 \ldots x_n\) is a minimum-bump linear extension of \(P\).

\[
\]

We now present the proof of the Lex-Yanking Lemma for Bump Number (Lemma 4.8), that all posets have the Lex-Yanking Property.

\textit{Proof of the Lex-Yanking Lemma:} The proof is by induction on the size of the poset. The property is clearly held by posets on zero or one elements. Let \(P\) be a poset on \(n > 1\) elements. Assume the Lex-Yanking Property is held by all posets on fewer elements; therefore, as a consequence of Lemma 4.11, we may assume that \textbf{GreedLex} orderings are minimum-bump for smaller posets.

If \(P\) is a poset with only one minimal element, then the selection of the first element of any linear extension is fixed, and the Lemma is trivially true.

So let us turn to the remaining case: let \(a\) and \(b\) be any minimal elements of \(P\) with \(\text{lex}(a) \geq \text{lex}(b)\), and let \(L\) be a linear extension of \(P\) that starts with \(b\) and has \(k\) bumps; we will use \(L = by_2y_3 \ldots y_{t-1}ay_{y+1} \ldots y_n\) to denote the various constituents of \(L\).

Let \(L'\) be the linear order \(L\) induced on the set \(V \setminus \{b\}\); hence \(L = bL'\).

The poset \(P \setminus \{b\}\) is smaller than \(P\), so by the inductive hypothesis any \textbf{GreedLex} ordering of \(P \setminus \{b\}\) has the minimum number of bumps for that poset. Let \(L'_G = x_2x_3 \ldots x_i-1ax_{i+1} \ldots x_n\) be a \textbf{GreedLex} ordering of \(P \setminus \{b\}\). The fact that \textbf{GreedLex} did not pick \(a\) until after \(x_2, \ldots, x_{i-1}\) means that \(\text{lex}(x_j) \geq \text{lex}(a) \geq \text{lex}(b), \forall j, 2 \leq j \leq i - 1\). It follows that \(b \not\prec x_j, \forall j, 2 \leq j \leq i - 1\), so \(a\) and \(b\) can be swapped in \(bL'_G\) without violating precedence. If no new bump is introduced, then swapping \(a\) and \(b\) in \(bL'_G\) gives a linear extension of the same (or smaller) bump number as \(L\), as required by the Lemma.

If bumps are introduced by the swap, then by the Swapping Lemma, it is at most one bump, and occurs only if \(b \prec x_{i+1}\) and \(a \not\prec x_{i+1}\).

If that is so, and yet \(\text{lex}(a) \geq \text{lex}(b)\), then by the Lex Lemma (Lemma 4.4), there is some \(c'\) where \(a \prec c', b \parallel c'\) and \(\text{lex}(c') \geq \text{lex}(x_{i+1})\). Note that it is possible that \(c'\) is not a minimum in \(P \setminus \{a, b, x_2, \ldots, x_{i-1}\}\). Let \(c\) be a descendent of \(c'\) that is a minimum in that poset, and observe that \(c \parallel b\) and \(\text{lex}(c) \geq \text{lex}(c') \geq \text{lex}(x_{i+1})\). Then by the inductive hypothesis, there is a linear extension of \(P \setminus \{a, b, x_2, \ldots, x_{i-1}\}\), call it \(cL''\), that starts with \(c\) and has no more bumps than \(x_{i+1} \ldots x_n\). Hence the linear extension \(ax_1 \ldots x_{i-1}bcL''\) has no more bumps than \(L\).
Theorem 4.12. Any linear extension constructed by the GreedLex Algorithm has minimum bump number.

Proof. By Lemma 4.11, any GreedLex ordering is minimum-bump if the poset (and all its subposets) has the Lex-Yanking Property. By Lemma 4.8, every poset has the Lex-Yanking Property. By definition, the GreedLex algorithm produces a GreedLex ordering of the input poset.

4.1.3 Implementation and Running Time

To analyze the GreedLex algorithm we present it here in more detail.

Algorithm 4.2 GreedLex

Input: A non-empty poset $P(V, \prec)$, methods to iterate through the upper and lower covers of $u$, for every $u \in V$; and lex : $V \rightarrow \mathbb{Z}$ the lexicographic labeling of $V$.

Output: A linear extension $\pi$ of $P$ with minimum bumps, and $bump$ the bump number for $P$

1: $i \leftarrow 0$ and $bumps \leftarrow 0$
2: for $u \in \text{minima}(P)$ do \hspace{1em} \triangleright Insert minima($P$) into a priority queue $Q$ using lex($\cdot$) as key
3: \hspace{1em} $Q$.add($\{\text{lex}(u), u\}$)
4: end for
5: for $i \leftarrow 1$ to $n$ do
6: \hspace{1em} if $Q$ is empty then
7: \hspace{2em} $bumps \leftarrow bumps + 1$
8: \hspace{2em} for each $v \in \text{uppercovers}(\pi[i]) \cap \text{minima}(P)$ do
9: \hspace{3em} $Q$.add($\{\text{lex}(v), v\}$) \hspace{1em} \triangleright Insert $v$ into $Q$ using $\text{lex}(v)$ as key
10: \hspace{2em} end for
11: \hspace{1em} else
12: \hspace{2em} $i \leftarrow i + 1$
13: \hspace{2em} $\pi[i] \leftarrow Q.pop().value$ \hspace{1em} \triangleright Get max element from $Q$ and remove it from $Q$
14: \hspace{2em} Remove $\pi[i]$ from $P$
15: \hspace{2em} for each $v \in \text{uppercovers}(\pi[i - 1]) \cap \text{minima}(P)$ do
16: \hspace{3em} $Q$.add($\{\text{lex}(v), v\}$)
17: \hspace{2em} end for
18: \hspace{1em} end if
19: end for
20: return $\pi$ and $bumps$

Let $[n]$ be the $v_1, v_2, \ldots, v_n$ elements of the poset, and $m$ the number of cover relations. Let the up-degree and down-degree of $v$ be the number of upper and lower covers of $v$ respectively. We assume $P$ is given as linked lists of upper and lower covers for each element. If only the upper cover lists are given, we can construct the lower covers in linear time. We begin by lex labeling the elements in linear time (Corollary 4.6).

Each element is inserted and removed from the priority queue exactly once. The priority queue always contains a subset of some antichain in the poset (the priority queue elements
are all minimal in the current poset $P$). Therefore the “pop max” operations on the priority queue (Step 13), as well as the insert operations, each run in time $O(\log w)$, where $w$ is the width of the input poset.

The test for inserting elements into the priority queue, i.e., Steps 15-17, requires the ability to test whether an element $v$ is minimal in the current poset. We accomplish this as follows: maintain an array of size $n$ numLowerCovers; numLowerCovers[$v$] is initially set to be the number of lower covers of $v$ in the input poset $P$. When we shell an element $u$ from the poset (Step 14), we traverse its list of upper covers; for each element $v$ that is an upper cover of $u$, we decrement numLowerCovers[$v$]. When for a given $v$, numLowerCovers[$v$] is zero, we insert $v$ into the priority queue. In that way, the two for-loops nested within the if-clause and the else-clause can be implemented to take a total time of $O(m + n \log w)$ over the entire run of the program, including the priority queue inserts. hence the total running time of the algorithm is $O(m + n \log w)$ time.

As previously mentioned, the fastest known algorithm for the bump number \cite{69} takes $O(m + n)$ time; the GreedLex algorithm however is simple to implement as shown above.
Chapter 5

LexCycle: A New Graph Parameter

Overview. In this chapter we introduce a new graph parameter, LexCycle, which captures linear structure seen on a number of graph classes. LexCycle is a graph invariant that measures the size of the maximum cycle of vertex orderings a LexBFS$^+$ multisweep algorithm produces. This chapter is based on [10] with cleaner and tighter proofs.

5.1 Motivation

As previously explained (cf. Chapter 2), multisweep based algorithms have proven successful on graph classes with particular structure. Among such results is a recent one of Dusart and Habib which states:

**Theorem 5.1.** [29] Given a cocomparability graph $G$ and a sequence $\{\sigma_i\}_{i \geq 1}$ of LexBFS orderings of $G$ where LexBFS$^+(G, \sigma_i) = \sigma_{i+1}$, the ordering $\sigma_n$, where $n = |V|$, is a cocomparability ordering of $G$, independent of the choice of $\sigma_1$.

Evidently, as the number of distinct orderings of vertices of a finite graph is finite ($n!$ to be exact), no matter which ordering $\sigma_1$ we start with, this sequence $\{\sigma_i\}_{i \geq 1}$ of LexBFS$^+$ orderings will cycle eventually. That is, for some $i$ and $k$, $\sigma_{i+k} = \sigma_i$. Thus if we keep running LexBFS$^+$ traversals, we will eventually cycle.

For general graphs there are two questions of high interest on the subject:

(i). Among all possible choices of $\sigma_1$ as a start ordering, how long does it take to cycle?

(ii). How large can this cycle be?

Regarding the second question, and restricted to the class of cocomparability graphs, Dusart and Habib have conjectured that the length of this largest cycle of vertex orderings is as small as possible:
Conjecture 5.2. [29] Given a cocomparability graph $G$ and a sequence $\{\sigma_i\}_{i \geq 1}$ of LexBFS orderings of $G$ where $\sigma_i$ is used to break ties for $\sigma_{i+1}$, for $i$ sufficiently large, and independent of the choice of $\sigma_1$, we have $\sigma_{i+2} = \sigma_i$.

This conjecture is still open. At the end of this chapter, we present an example that requires $n \text{ LexBFS}^+$ traversals to reach a cycle; hence $i$ sufficiently large means $i \geq n$ in general. Experimentally, we noticed that we achieve a cycle much earlier than $n \text{ LexBFS}^+$ traversals.

Observing that cocomparability graphs are asteroidal triple free, and thus have asteroidal number two, Stacho, [72] asked if the length all such cycles is bounded by the asteroidal number of the graph.

In this chapter, we first answer Stacho’s question negatively. Then, we provide strong support for the conjecture of Dusart and Habib by proving it for the subclass of domino-free cocomparability graphs. While this subclass of cocomparability graphs contains proper interval graphs, interval graphs, cographs and cobipartite graphs, for each of these cases, we give an independent proof which provides stronger results, and sheds light into structural properties of these graph classes. Furthermore, we prove that the same bound of 2 holds for trees, which are not necessarily asteroidal triple-free graphs, using an old result of Jordan (1869) – Two BFS’s are enough to compute a diametral pair of a tree. Before formally defining LexCycle, the parameter that captures the size of this cycle, we first try to understand why Conjecture 5.2 might hold.

5.2 LexCycle: First Properties

Since cocomparability graphs encapsulate all the classes we will discuss – and since we are ultimately interested in proving Conjecture 5.2 – we will focus on the LexBFS properties of cocomparability graphs. Recall that cocomparability orderings are preserved under LexBFS$^+$ (Theorem 2.12), and that LexBFS cocomparability orderings give rise to the LexBFS $C_4$ Property (Property 2.13).

In this chapter, when choosing vertex $d$ as described in Property 2.13, we always choose it as the leftmost private neighbour of $b$ with respect to $c$. We write $d = \text{LMPN}(b|\sigma c)$ and read $d$ is the leftmost private neighbour of $b$ with respect to $c$ in $\sigma$. This is to say that prior to visiting vertex $d$ in $\sigma$, vertices $b$ and $c$ were tied and label$(b) = \text{label}(c)$ as assigned by Algorithm LexBFS$^+$, and vertex $d$ caused $b \prec_\sigma c$.

An intuition as to why Conjecture 5.2 could be true comes from the Flipping Lemma (Lemma 2.15), which states that when applied to a cocomparability ordering, LexBFS$^+$ will reverse all the non edges. Therefore, in a sequence $\{\sigma_i\}_{i \geq 1}$, with $\sigma_1$ being a cocomparability

---

$^1$The label assigned at step 7 of the LexBFS algorithm – Algorithm 2.1.
ordering, all pairs of non adjacent vertices are exactly in the same order in \( \sigma_i \) and \( \sigma_{i+2} \). A direct consequence of the Flipping Lemma is the following corollary:

**Corollary 5.3.** For a cocomparability graph \( G \), \( \text{LexCycle}(G) \) is necessarily of even length \( \geq 2 \).

**Proof.** If \( G \) contains a pair of nonadjacent vertices, then the claim is a trivial consequence of the Flipping Lemma. Otherwise \( G \) is a complete graph and \( \sigma_2 = \sigma_1^d \) is the cycle of length 2.

If Conjecture 5.2 is true, then the following simple algorithm will always return cocomparability orderings that cycle, and thus a transitive orientation of a comparability graph.

**Algorithm 5.1 SimpleTO:** A Potential Simple Transitive Orientation Algorithm

**Input:** A cocomparability graph \( G(V,E) \)

**Output:** An ordering \( \sigma_i \) of \( G \) whose \( \text{LexBFS}^+(\sigma_i) \) is \( \sigma_{i-1} \)

1: \( \sigma_1 \leftarrow \text{LexBFS}(G) \), \( \sigma_2 \leftarrow \text{LexBFS}^+(\sigma_1) \), \( \sigma_3 \leftarrow \text{LexBFS}^+(\sigma_2) \)
2: \( i \leftarrow 3 \)
3: while \( \sigma_i \neq \sigma_{i-2} \) do
4: \( i \leftarrow i + 1 \)
5: \( \sigma_i \leftarrow \text{LexBFS}^+(\sigma_{i-1}) \)
6: end while
7: return \( \sigma_i \)

A simple consequence of Theorems 2.12 and 5.1 is:

**Proposition 5.4.** Let \( G \) be a cocomparability graph. If Algorithm SimpleTO (Algorithm 5.1) ends when applied on \( G \), then the last two computed \( \text{LexBFS}^+ \) orderings are cocomparability orderings.

**Definition 5.5.** A sequence of orderings \( \sigma_1, \sigma_2, \ldots, \sigma_k \) is a cycle of orderings of length \( k \) if \( \sigma_i = \text{LexBFS}^+(\sigma_{i-1}) \) for \( i \in \{2, \ldots, k\} \) and \( \sigma_1 = \text{LexBFS}^+(\sigma_k) \).

With this idea in mind, we now define LexCycle. Starting from an ordering \( \sigma_0 \) of \( G \), we compute the following sequence: \( \sigma_{i+1} = \text{LexBFS}^+(G, \sigma_i) \).

**Definition 5.6** (LexCycle). For a graph \( G(V,E) \), \( \text{LexCycle}(G) \) is the maximum length of a cycle of vertex-orderings obtained via a sequence of \( \text{LexBFS}^+ \) sweeps.

Another viewpoint on LexCycle can be seen as a directed graph, \( G_{lex} \), constructed as follows: Vertices of \( G_{lex} \) are the \( \text{LexBFS} \) orderings of \( G \), \( V(G_{lex}) = \{\sigma_0, \sigma_1, \ldots, \sigma_k\} \) and we add an arc from \( \sigma_i \) to \( \sigma_j \) if \( \text{LexBFS}^+(G, \sigma_i) = \sigma_j \). It is easy to see that \( G_{lex} \) is a functional
graph since every vertex has an out-degree of exactly one; and therefore every connected component of $G_{lex}$ is a circuit on which are planted some directed trees. For instance, if $K$ is a clique, $K_{lex}$ is just the union of directed circuits of size two joining one permutation to its reverse. LexCycle($G$) is then just the maximum size of a directed circuit in $G_{lex}$, but we do not know of any example of a graph with two distinct cycle lengths.

We study the first properties of this new graph invariant, LexCycle. Due to the nature of the $^+$ rule, LexCycle($G$) $\geq 2$ as soon as $G$ contains more than one vertex. At first glance we know that LexCycle($G$) $\leq n!$, and more precisely LexCycle($G$) is bounded by the number of LexBFS orderings of $G$.

**The Starjoin Construction.** We introduce a construction below, which we call *Starjoin*, to first disprove a conjecture of Stacho [72], which claims that LexCycle($G$) $\leq an(G)$. We then use the Starjoin construction to show that we cannot bound LexCycle($G$) by a polynomial on $n$.

Let us first consider some interesting examples with high values of LexCycle, i.e., $\geq 3$.

Figure 5.1 illustrates a graph with LexCycle($G_3$) $\geq 3 = an(G_3)$, starting with $\sigma_1 = $ LexBFS($G$) = $x, b, a, c, e, f, d, z, y$.

![Figure 5.1: Example of a graph with LexCycle($G_3$) $\geq 3$ where the 3-cycle consists of $C_3 = [\sigma_1, \sigma_2, \sigma_3]$.](image)

LexBFS($G$) = $\sigma_1 = x, b, a, c, e, f, d, z, y$
LexBFS$^+$(\(\sigma_1\)) = $\sigma_2 = y, f, e, a, c, d, b, x, z$
LexBFS$^+$(\(\sigma_2\)) = $\sigma_3 = z, d, c, e, a, b, f, y, x$
LexBFS$^+$(\(\sigma_3\)) = $\sigma_1 = x, b, a, c, e, f, d, z, y$

Figure 5.2 illustrates a graph with LexCycle($G_4$) $\geq 4 = an(G_4)$, starting with $\mu_1 = $ LexBFS($G$) = $x_4, z_4, y_1, y_3, y_4, y_2, z_2, z_1, z_3, x_2, x_3, x_1$.

![Figure 5.2: Example of a graph with LexCycle($G_4$) $\geq 4$ where the 3-cycle consists of $C_3 = [\sigma_1, \sigma_2, \sigma_3]$.](image)
We now show how one can construct graphs with \( \text{LexCycle}(G) > an(G) \). Consider the following graph operation that we call Starjoin.

**Definition 5.7 (Starjoin).** For a family of graphs \( \{G_i\}_{1 \leq i \leq k} \), we define \( H = \text{Starjoin}(G_1, \ldots, G_k) \) as follows: For \( i \in [k] \), add a universal vertex \( g_i \) to \( G_i \), then add a root vertex \( r \) adjacent to all \( g_i \)'s.

**Property 5.8.** Let \( G_i \) be a graph with a cycle \( C_i \) in a sequence of \( \text{LexBFS}^+ \) orderings of \( G_i \) and let \( H = \text{Starjoin}(G_1, \ldots, G_k) \). We have

- \( an(H) = \max\{k, an(G_1), an(G_2), \ldots, an(G_k)\} \)
- \( \text{LexCycle}(H) \geq \text{lcm}_{1 \leq i \leq k}\{|C_i|\} \), where lcm stands for the least common multiple.

**Proof.** Notice first that selecting one vertex per \( G_i \) would create a \( k \)-asteroidal set. Since every \( g_i \) vertex is universal to \( G_i \), we can easily see that every asteroidal set of \( H \) is either restricted to one \( G_i \), or it contains at most one vertex per \( G_i \). This yields the first formula.

For the second property, we notice first that a cycle of \( \text{LexBFS}^+ \) orderings is completely determined by its initial \( \text{LexBFS} \) ordering, since all ties are resolved using the + rule. For \( 1 \leq i \leq k \), let \( \sigma_i^1 \) denote the first \( \text{LexBFS}^+ \) ordering on \( C_i \), the cycle in a sequence of \( \text{LexBFS}^+ \) orderings of \( G_i \).

Consider the following \( \text{LexBFS} \) ordering of \( H \): \( \sigma_i^H = r, g_1, \ldots, g_k\sigma_i^1, \ldots, \sigma_i^k \). Notice that in any \( \text{LexBFS}^+ \) ordering of the cycle generated by \( \sigma_i^H \) the vertices of \( G_i \) are consecutive. Furthermore \( \sigma_j^H|_{G_i} = \text{LexBFS}^+(G_i, \sigma_j^1) \). Therefore if we take \( \sigma_i^1 \) as the first \( \text{LexBFS}^+ \) ordering of \( C_i \), then the length of the cycle generated by \( \sigma_i^H \) is necessarily a multiple of \( |C_i| \).

This property shows, intuitively, that while \( \text{LexCycle} \) of a starjoin of graphs can grow as a multiplicative function of the \( \text{LexCycles} \) of the components it is built from, the total
number of vertices grows linearly and the asteroidal number remains almost fixed. To turn this intuition into facts, it remains to construct a sequence of graphs $G_1, \ldots, G_k$ with corresponding $\text{LexCycle}(G_i) = p_i$, where the number of vertices is bounded by a polynomial on $p_i$. While the general case remains to be constructed, we use the two constructions, we gave before to answer Stacho’s question in the negative:

**Corollary 5.9.** There exists a graph $G$ satisfying $\text{LexCycle}(G) \geq an(G)$.

**Proof.** To see this, consider $H = \text{Starjoin}(G_3, G_4)$, constructed using the graphs in Figures 5.1 and 5.2, where $an(H) = 4$ and $\text{LexCycle}(H) \geq 12$. Take $\sigma_1^H = r, g_1, g_2, \sigma_1, \mu_1$, where $\sigma_1, \mu_1$ are as defined in the examples above. The cycle of $\text{LexBFS}^+$ orderings generated by $\sigma_1^H$ is necessarily of size $\geq 12$. \hfill \Box

Intuitively, we think of graphs with small LexCycle as “linearly structured”. We try to capture this idea in this work by focusing on graphs with small LexCycle value; i.e., $\text{LexCycle}(G) = 2$. Having $\text{LexCycle}(G) = 2$ means there exists two orderings $\sigma, \tau$ of $G$ where $\sigma = \text{LexBFS}^+(\tau) = v_1 \ldots v_n$ and $\tau = \text{LexBFS}^+(\sigma) = v_n \ldots v_1$. In this chapter, we want to understand whether these two orderings form some sort of a “spine” of $G$ that lies between $v_1$ and $v_n$. For this special case, we first show that Property 5.8 can be generalized to modular decomposition. We begin by stating two known facts about modular decomposition:

1. It is well known (see [41] for instance) that if $M \subseteq V$ is a module of $G$, and $\sigma$ a $\text{LexBFS}$ ordering of $G$, then $\sigma[M]$, the ordering of $M$ induced by $\sigma$, is a valid $\text{LexBFS}$ ordering of $G[M]$, the subgraph induced by $M$.

2. Given a modular decomposition $\mathcal{M} = \{M_1, M_2, \ldots, M_k\}$ of $G$, and $\sigma$ a $\text{LexBFS}$ ordering of $G$, we define the discovery time of a partition $M_i$ of $\mathcal{M}$ as $\max\{\sigma(v) : v \in M_i\}$. It is easy to see that $\sigma[M]$, the ordering the elements of $\mathcal{M}$ by their discovery time with respect to $\sigma$, yields a valid $\text{LexBFS}$ ordering of $G/\mathcal{M}$, the quotient graph of $G$ with respect to $\mathcal{M}$.

These two facts can be easily extended using $\text{LexBFS}^+$ as follows:

**Lemma 5.10.** Let $\tau$ be a total ordering of $G$, $M \subseteq V$ a module of $G$, and $\sigma = \text{LexBFS}^+(G, \tau)$, then $\sigma[M] = \text{LexBFS}^+(G[M], \tau[M])$. Furthermore, if $\mathcal{M}$ is a modular decomposition of $G$, then $\sigma[\mathcal{M}] = \text{LexBFS}^+(G/\mathcal{M}, \tau[\mathcal{M}])$.

**Proof.** For the first part of this statement, we just note that since the vertices of $M$ have the same neighbourhood outside $M$, no tie breaking rule can distinguish the vertices of $M$ from the outside, i.e., $V \setminus M$. 

Similarly for the second statement, one can consider every module $M_i$ of $\mathcal{M}$ as a unique vertex since all the vertices in $M_i$ behave the same with respect to $V \setminus M_i$, then it suffices to consider the $\text{LexBFS}^+$ ordering on the graph $G/\mathcal{M}$.

A graph class $\mathcal{G}$ is said to be **hereditary** if $\forall G \in \mathcal{G}$, every induced subgraph $H$ of $G$ also belongs to $\mathcal{G}$.

**Theorem 5.11.** Let $\mathcal{G}$ be a hereditary class of graphs. If for every prime graph $G \in \mathcal{G}$, $\text{LexCycle}(G) = 2$, then for every $G$ in $\mathcal{G}$, $\text{LexCycle}(G) = 2$.

**Proof.** The proof goes by induction on $|V|$ for $G \in \mathcal{G}$. Suppose that $G$ is not a prime graph, then it admits at least one non-trivial module $M$, such that $1 < |M| < |V|$. Let us consider the modular decomposition of $G$ which contains this non-trivial module $M$, $\mathcal{M} = \{M, \ldots\}$. Both graphs $G[M]$ and $G/\mathcal{M}$ have strictly fewer vertices than $G$. Using the induction hypothesis, and the fact that both $G[M]$ and $G/\mathcal{M}$ belong to $\mathcal{G}$, any series of $\text{LexBFS}^+$ applied on $G[M]$ or $G/\mathcal{M}$ reaches a cycle of length 2. Therefore any series of $\text{LexBFS}^+$ applied on $G$ reaches a cycle of length 2.

As an immediate consequence of the above theorem, we have:

**Corollary 5.12.** If $G$ is a cograph then $\text{LexCycle}(G) = 2$.

**Proof.** This is because the only prime cograph is the single vertex, and cographs are totally decomposable with respect to modular decomposition.

### 5.3 Graph Classes with Small LexCycle

In support of Conjecture 5.2, we show in this section that it holds for the subclass of domino-free cocomparability graphs. This class in particular includes the class of interval graphs and cobipartite graphs, but for these two subclasses we provide independent proofs which imply stronger results. For interval graphs we show that the two orderings of the LexCycle are left endpoint and right endpoint orderings with respect to some (the same) interval representation, and that such a cycle is reached in at most $n$ iterations of the multisweep algorithm. Moreover in the case of proper interval graphs, the LexCycle is reached in at most 5 iterations. The proof for cobipartite graphs is, first of all interesting for the different flavor of the proof, and secondly it provides an upper bound of $n^2$ iterations of the multisweep algorithm before reaching the cycle.

We note that while Dusart and Habib’s Theorem 5.1 guarantees reaching a cocomparability ordering in at most $n$ iterations of the multisweep algorithm applied on a cocomparability graph, we know of no non-trivial bound on when the cycle will be reached.
In the proofs to come, we often show the existence of some vertices in the orderings, the observation below explains how we choose such vertices.

**Observation 5.13.** Consider a sequence \( \{\sigma_i\}_{i \geq 1} = \sigma_1, \sigma_2, \ldots \) of \( \text{LexBFS}^+ \) sweeps on a cocomparability graph \( G \), where \( \sigma_{i+1} = \text{LexBFS}(\sigma_i) \). If there exists an edge \( ab \in E \) and two consecutive orders \( \sigma_j, \sigma_{j+1} \) such that \( a \prec_{j,j+1} b \) then vertex \( a \) must have a private neighbour \( c \) with respect to \( b \) that pulled \( a \) before \( b \) in \( \sigma_{j+1} \), and overruled the \( + \) rule. If such a scenario occurs, we always choose \( c \) as the leftmost such private neighbour of \( a \) with respect to \( b \) in \( \sigma_{j+1} \), and once again write \( c = \text{LMPN}(a|_{j+1} b) \).

### 5.3.1 Interval Graphs & Proper Interval Graphs

For these upcoming proofs, we remind the reader of the following: For an interval graph \( G(V, E) \), \( R \) is an intersection representation of \( G \) if for every \( v_i \), there is a non-empty interval \( I_i = [s_i, f_i] \) with start and finish times \( s_i < f_i \) in \( R \), such that for any two vertices \( v_i, v_j : v_i v_j \in E \) if and only if \( I_i \cap I_j \neq \emptyset \). A left endpoint ordering of \( R \) is the increasing ordering of the \( s_i \)'s for \( i \in [n] \). A right endpoint ordering of \( R \) is the decreasing ordering of the \( f_i \)'s for \( i \in [n] \). It is easy to see that an I-ordering is just a left endpoint ordering of a representation \( R \) of \( G \).

Recall that an interval graph is also a cocomparability graph, and while not every cocomparability ordering of an interval graph \( G \) is a left or right endpoint ordering of \( G \), every left endpoint ordering is just an I-ordering, and thus a cocomparability ordering. We begin with the following easy claim:

**Lemma 5.14.** Let \( G \) be an interval graph, and \( \sigma \) a cocomparability ordering of \( G \) that is not an I-ordering. Then \( \tau = \text{LexBFS}^+(G, \sigma) \) is an I-ordering of \( G \).

**Proof.** For sake of contradiction, suppose not. Then \( \tau \) must have a triple \( a \prec_\tau b \prec_\tau c \) where \( ac \in E \) and \( ab \notin E \). Thus the triple \( abc \) forms a bad triple in \( \tau \) and thus by the \( \text{LexBFS} \ C_4 \) Property (Property 2.13), there exists a vertex \( d \prec_\tau a \) such that \( d, a, b, c \) induces a \( C_4 \) in \( G \), a contradiction to \( G \) being chordal, and thus interval. \( \square \)

Therefore, by Theorems 2.12 and 5.1, starting with any vertex ordering of an interval graph the cycle of orderings obtained via a sequence of \( \text{LexBFS}^+ \) traversals will consist of I-orderings, and thus of left endpoint orderings.

**Lemma 5.15.** Given an I-ordering \( \sigma \) of \( G \) interval, there exists an interval representation \( R \) of \( G \) such that \( \sigma \) is the left endpoint ordering of \( R \) and \( \tau = \text{LexBFS}^+(G, \sigma) \) the right endpoint ordering of \( R \).
Proof. Let $\mathcal{R}$ be the interval representation of $G$ using $\sigma$. The claim of the lemma can be rephrased as: we can modify $\mathcal{R}$ by stretching intervals without affecting their overlap such that the resulting representation $\mathcal{R}'$ has left endpoint ordering $\sigma$ and right endpoint ordering $\tau$.

Let $\phi$ be the right endpoint (decreasing) ordering of the intervals in $\mathcal{R}$. Without loss of generality, we can assume all the start and finish times are distinct in $\mathcal{R}$.

Clearly if $\phi = \tau$ then we are done. Suppose not, consider the leftmost (first) difference between $\tau$ and $\phi$. In particular, Let $a, b$ (and their corresponding intervals $I_a, I_b$) be this difference, that is $a \prec_\tau b$ and $b \prec_\phi a$. Either (i) $a \prec_\sigma b$ or (ii) $b \prec_\sigma a$.

(i) Suppose $a \prec_\sigma b$, then $I_a$ starts before $I_b$. Furthermore, $I_a$ ends before $I_b$ since $b \prec_\phi a$, and $ab \in E$ since by the Flipping Lemma, $a, b$ did not flip from $\sigma$ to $\tau$. To create $\mathcal{R}'$, we can stretch the endpoint of $I_a$ to bypass $I_b$, and thus $\tau$ is a right endpoint ordering of $\mathcal{R}'$. Since we are not changing the left endpoints, $\sigma$ remains a left endpoint ordering of $\mathcal{R}'$. This stretching is problematic if the endpoint of $I_a$ causes an overlap with an interval $I_c$ but $ac \notin E$. This is only a problem if $cb \in E$, i.e., $I_c \cap I_b \neq \emptyset$. If this is the case, then $I_c$ is completely to the right of $I_a$ and thus $a \prec_\sigma c$ and $c \prec_\tau a$. Since $c$ is a private neighbour of $b$ with respect to $a$, we have the bad triple $c \prec_\tau a \prec_\tau b$ which causes an induced $C_4$ (using the LexBFS $C_4$ property) on an interval graph $G$. Thus no such $I_c$ exists, and we can produce $\mathcal{R}'$.

(ii) Suppose now that $b \prec_\sigma a$, then $I_b$ starts before $I_a$ and since $b \prec_\phi a$, $I_b$ ends after $I_a$, thus $I_a$ is fully contained in $I_b$. Once again, we can create $\mathcal{R}'$ by stretching the endpoint of $I_a$ to bypass $I_b$, and thus $\tau$ is a right endpoint ordering of $\mathcal{R}'$. Unless of course, a similar scenario described above with an $I_c$ occurs. Using again the Flipping lemma on $ac \notin E$, we get $c \prec_\tau a \prec_\tau b$, another bad triple that will induce a $C_4$ in an interval graph.

Thus in all cases, if $\sigma$ is an I-order of $G$, and $\tau = \text{LexBFS}^+(G, \sigma)$, then there exists an interval realization $\mathcal{R}'$ of $G$ where $\sigma$ is the left endpoint ordering, and $\tau$ is the right endpoint ordering of $\mathcal{R}'$. $\square$

Theorem 5.16. Let $G$ be an interval graph, $\sigma_0$ an arbitrary cocomparability order of $G$ and $\{\sigma_i\}_{i \geq 1}$ a sequence of $\text{LexBFS}^+$ orderings where $\sigma_1 = \text{LexBFS}^+(\sigma_0)$. Then $\sigma_1 = \sigma_3$.

Proof. Consider the following orderings:

$$\sigma_1 = \text{LexBFS}^+(\sigma_0) \quad \sigma_2 = \text{LexBFS}^+(\sigma_1) \quad \sigma_3 = \text{LexBFS}^+(\sigma_2)$$

Suppose, for sake of contradiction, that $\sigma_1 \neq \sigma_3$. Let $k$ denote the index of the first (leftmost) vertex where $\sigma_1$ and $\sigma_3$ differ. In particular, let $a$ (resp. $b$) denote the $k^{th}$ vertex of $\sigma_1$ (resp. $\sigma_3$). Let $S$ denote the set of vertices preceeding $a$ in $\sigma_1$ and $b$ in $\sigma_3$. 

Since the ordering of the vertices of $S$ is the same in both $\sigma_1$ and $\sigma_3$, and $a, b$ were chosen in different LexBFS orderings, it follows that $\text{label}(a) = \text{label}(b)$ in both $\sigma_1$ and $\sigma_3$ when both $a$ and $b$ were being chosen. Therefore $N(a) \cap S = N(b) \cap S$. So if $a$ were chosen before $b$ in $\sigma_1$ then the + rule must have been used to break ties between $\text{label}(a) = \text{label}(b)$. This implies $b \prec_0 a$, similarly $a \prec_2 b$. The ordering of the pair $a, b$ is thus as follows:

\[
\begin{align*}
\sigma_0: & \quad \ldots b \ldots a \ldots \\
\sigma_1: & \quad \ldots a \ldots b \ldots \\
\sigma_2: & \quad \ldots a \ldots b \ldots \\
\sigma_3: & \quad \ldots b \ldots a \ldots 
\end{align*}
\]

Using the Flipping Lemma, it is easy to see that $ab \in E$. Since $a \prec_{1,2} b$, choose vertex $c$ as $c = LMPN(a \mid 2b)$. Therefore $c \prec_2 a \prec_2 b$ and $ac \in E, bc \notin E$.

Since $\sigma_0$ is a cocomparability order, by Theorem 2.12, $\sigma_1, \sigma_2, \sigma_3$ are cocomparability orderings. Using the Flipping Lemma on the non-edge $bc$, we have $c \prec_2 a \prec_2 b$ implies $c \prec_0 b$. Therefore in $\sigma_0$, $c \prec_0 b \prec_0 a$ and $ac \in E, bc \notin E$. Using the LexBFS 4 Point Condition (Theorem 2.9), there exists a vertex $d$ in $\sigma_0$ such that $d \prec_0 c \prec_0 b \prec_0 a$ and $db \in E, da \notin E$. By the LexBFS $C_4$ Property (Property 2.13), $dc \in E$ and the quadruple $abdc$ forms a $C_4$ in $G$, thereby contradicting $G$ being a chordal, and thus interval graph. \qed

Using Lemma 5.15 and Theorem 5.16, we conclude with the following corollary:

**Corollary 5.17.** Interval graphs have $\text{LexCycle} = 2$ and the two orderings $\sigma, \tau$ that witness this cycle are left endpoint and right endpoint orderings of the same I-ordering.

**Proof.** Let $G$ be an interval graph. Consider a sequence of orderings $\{\sigma_i\}_{i \geq 1}$ computed via a series of consecutive LexBFS$^+$ on $G$. By Theorem 5.1, there is a $j \leq n$ for which $\sigma_j$ is a cocomparability order. The first half of the claim now follows from Theorem 5.16.

The second half of the claim follows from Lemma 5.15. Indeed let $\sigma = \text{LexBFS}^+(\tau)$ and $\tau = \text{LexBFS}^+(\sigma)$, then by Lemma 5.15, there exists an interval representation $\mathcal{R}$ where $\sigma$ is the left endpoint ordering of $\mathcal{R}$ and LexBFS$^+(\sigma)$ is the right endpoint ordering of $\mathcal{R}$. Thus $\tau$ is the right endpoint ordering of $\mathcal{R}$. \qed

Although the orderings we obtain are left endpoint and right endpoint orderings, we note that these orderings are not necessarily duals of one another as one can see in the example below:
Proper Interval Graphs. For proper interval graphs, we show that any two orderings that characterize the cycle must be duals. To this end, the following claim is crucial. It shows that the Flipping Lemma holds for edges as well when $G$ is a proper interval graph.

**Claim 5.18.** Let $\sigma$ a PI-ordering (Definition 2.5) of $G$, and $\tau = \LexBFS^+(\sigma)$. For every edge $uv \in E$, $u \prec_\sigma v$ if and only if $v \prec_\tau u$.

**Proof.** Suppose not. Let $x, y$ be a pair of vertices such that $xy \in E$ and $x \prec_\sigma y, x \prec_\tau y$. Since the pair maintained the same order on consecutive sweeps, the $^+$ rule was not used to break ties between $x$ and $y$, and thus there must exist a private neighbour $z$ of $x$ with respect to $y$, such that $z \prec_\tau x \prec_\tau y$ and $zx \in E, zy /\notin E$. Using the Flipping Lemma, this implies $x \prec_\sigma y \prec_\sigma z$ with $xy, xz \in E$ and $yz /\notin E$, which contradicts $\sigma$ being a PI-order.

**Theorem 5.19.** Let $G$ be a proper interval graph and $\sigma$ a PI-ordering of $G$, then $\LexBFS^+(\sigma) = \sigma$.

**Proof.** Let $\sigma$ a PI-ordering of $G$. Consider the ordering $\tau = \LexBFS^+(\sigma)$. Using the Flipping Lemma and Claim 5.18, it follows that both the edges and non edges of $G$ are flipped in $\tau$. Thus $\sigma = \tau$.

Therefore, using Theorem 5.19 and Theorem 5.20 stated below, we get Corollary 5.21.

**Theorem 5.20.** [11] A graph $G$ is a proper interval graph if and only if the third $\LexBFS^+$ sweep on $G$ is a PI-ordering.

**Corollary 5.21.** If $G$ is a proper interval graph, Algorithm SimpleTO (Algorithm 5.1) stops at $\sigma_5 = \sigma_3$.

**Proof.** By Theorem 5.20, we know that $\sigma_3$ is a PI-ordering. Using Theorem 5.19, we conclude that Algorithm 5.1 applied on a PI-ordering computes $\sigma_4 = \sigma_3^d$ and $\sigma_5 = \sigma_4^d = (\sigma_3^d)^d = \sigma_3$. 

Since proper interval graphs have LexCycle two, where the two orderings are duals of one another, a natural conjecture to raise then is that orderings are duals only if the graph
is a proper interval graph. This is indeed true for interval graphs. We mean by this that an interval graph \( G \) has dual orderings if and only if \( G \) is a proper interval graph. However, there are arbitrary graphs for which LexCycle orderings are duals, yet the graphs are not interval graphs, in fact, not even asteroidal triple free graphs, as illustrated below:

![Graph with an asteroidal triple (c, g, j) with two orderings duals of one another.](image)

Figure 5.4: A graph with an asteroidal triple \((c, g, j)\) with two orderings duals of one another.

### 5.3.2 Co-bipartite Graphs

Let \( G(V = A \cup B, E) \) be a co-bipartite graph (the complement of a bipartite graph), where both \( A \) and \( B \) are cliques. Notice that any ordering \( \sigma \) on \( V \) obtained by first placing all the vertices of \( A \) in any order followed by the vertices of \( B \) in any order is a cocomparability ordering. In particular, such an ordering is precisely how any LexBFS cocomparability ordering of \( G \) is constructed, as shown by Lemma 5.23 below. We first show the following easy observation.

**Claim 5.22.** Let \( G \) be a co-bipartite graph, and let \( \sigma \) be a cocomparability ordering of \( G \). In any triple of the form \( a \prec \sigma b \prec \sigma c \), either \( ab \in E \) or \( bc \in E \).

**Proof.** Suppose otherwise, then if \( ac \in E \), we contradict \( \sigma \) being a cocomparability ordering, and if \( ac \not\in E \) the triple \( abc \) forms a stable set of size 3, which is impossible since \( G \) is co-bipartite.

**Lemma 5.23.** Let \( G \) be a co-bipartite graph, and let \( \sigma = x_1, x_2, \ldots, x_n \) be a LexBFS cocomparability ordering of \( G \). There exists \( i \in [n] \) such that \( \{x_1, \ldots, x_i\} \) and \( \{x_{i+1}, \ldots, x_n\} \) are both cliques.

**Proof.** Let \( i \) be the largest index in \( \sigma \) such that \( \{x_1, \ldots, x_i\} \) is a clique. Suppose \( \{x_{i+1}, \ldots, x_n\} \) is not a clique, and consider a pair of vertices \( x_j, x_k \) where \( x_jx_k \not\in E \) and \( i + 1 \leq j < k \). By the choice of \( i \), vertex \( x_{i+1} \) is not universal to \( \{x_1, \ldots, x_i\} \). Since \( \sigma \) is a LexBFS ordering, vertex \( x_j \) is also not universal to \( \{x_1, \ldots, x_i\} \) for otherwise label\((x_j)\) would be lexicographically greater than label\((x_{i+1})\) implying \( j < i + 1 \); unless \( i + 1 = j \), in which case it is obviously true. Let \( x_p \in \{x_1, \ldots, x_i\} \) be a vertex not adjacent to \( x_j \). We thus have \( x_p \prec \sigma x_j \prec \sigma x_k \) and both \( x_p, x_j, x_jx_k \not\in E \). A contradiction to Claim 5.22 above.

Since co-bipartite graphs are cocomparability graphs, by Theorem 5.1, after a certain number \( t \leq n \) of iterations, a series of LexBFS\(^+ \) sweeps yields a cocomparability ordering \( \sigma_t \).
By Lemma 5.23, this ordering consists of the vertices of one clique $A$ followed by another clique $B$.

Assume $a_1, \ldots, a_p, b_q, \ldots, b_1$ is the ordering of $\sigma_t$ (the reason why the indices of $B$ are reversed will be clear soon). Consider a $p \times q$ matrix $M_t$ defined as follows:

$$M_t[i,j] = \begin{cases} 
1 \text{ if } a_i b_j \in E \\
0 \text{ otherwise}
\end{cases}$$

An easy but crucial property that follows from the definition of LexBFS is the following: the columns of this matrix $M_t$ are sorted lexicographically in increasing order (for any vectors $X$ and $Y$, of the same length, the lexicographic order is defined by $X <_{\text{lex}} Y$ if the smallest integer $k$ for which $X_k \neq Y_k$ satisfies $X_k < Y_k$). This is because when we construct $M_t$, we reverse the indices of $B$. For every pair $b, b' \in B, b <_{\sigma_t} b'$ means vertex $b$ has a larger lexicographic label than $b'$ with respect to the ordering of $A$. Thus, the adjacency vector of $b$ with respect to $a_1, \ldots, a_p$, is lexicographically larger than the adjacency vector of $b'$ with respect to $a_1, \ldots, a_p$. Placing the indices of $B$ in reverse in $M_t$ guarantees that the columns of $M_t$ are sorted lexicographically in non-decreasing order.

Consider $\sigma_{t+1} = \text{LexBFS}^+(\sigma_t)$, and notice that $\sigma_{t+1}$ begins with the vertices of $B$ in the ordering $b_1, b_2, \ldots, b_q$ followed by the vertices of $A$ which are sorted exactly by sorting the corresponding rows of $M_t$ lexicographically in non-decreasing order (the first vertex to appear after $b_q$ being the maximal row, that is the one we put at the bottom of the matrix – i.e., once again placing the indices of $A$ according to $\sigma_{t+1}$ in reverse order in $M_{t+1}$). But then to obtain $\sigma_{t+2}$ we just need to sort the columns lexicographically, and so on.

Therefore to prove that LexCycle = 2 for cobipartite graphs, it suffices to show that this process must converge to a fixed point: That is, after some number of steps, we get a matrix such that both rows and columns are sorted lexicographically, which implies we have reached a 2 cycle. This is guaranteed by the following lemma (which we state for $0-1$ matrices, but is in fact true for any integer valued matrix):

**Lemma 5.24.** Let $M$ be a matrix with $\{0, 1\}$ entries. Define a sequence of matrices $\{M_i\}_{i \geq 1}$ as follows:

1. $M_0 = M$
2. if $i$ is even, $M_i$ is obtained by sorting the rows of $M_{i-1}$ in increasing lexicographical order.
3. if $i$ is odd, $M_i$ is obtained by sorting the columns of $M_{i-1}$ in increasing lexicographical order.
There exists a \( k \leq n^2 \) for which \( M_{k-1} = M_k \).

Proof. For a matrix \( M_k \), let \( M_k^{ij} \) denote the submatrix of \( M_k \) induced by the first \( i \) rows and \( j \) columns, let \( R_k^i \) and \( C_k^j \) denote the \( i^{th} \) row and \( j^{th} \) column of \( M_k \) respectively. For every \( k \), we define a vector \( X_k \) obtained by reading the entries of the matrix \( M_k \) from left to right, top to bottom. I.e., \( X_k \) is the concatenation of the rows of \( M_k \). We prove the statement of the theorem in two parts:

Part (1). For every \( k \in [n^2] \), \( X_k \) is always lexicographically smaller than \( X_{k-1} \).

Part (2). The smallest index at which two consecutive vectors \( X_k \) and \( X_{k+1} \) differ strictly increases. Thus there exists a \( k \leq n^2 \) for which \( X_{k-1} = X_k \).

Part (1). We prove the first claim by contradiction. Suppose for some \( k \), \( X_{k-1} \) is lexicographically smaller than \( X_k \). Assume the first index in which \( X_{k-1} \) and \( X_k \) differ corresponds to the entry with coordinates \((i, j)\) in both matrices \( M_{k-1}, M_k \), and that it is equal to 0 in \( X_{k-1} \) and 1 in \( X_k \). This implies, in particular, that the submatrices obtained from \( M_{k-1}^{ij} \) and \( M_k^{ij} \) are identical except for the entry \([i, j]\).

We consider the case when \( k \) is even, the case of odd \( k \) being analogous. If \( k \) is even, then \( M_k \) was obtained from \( M_{k-1} \) by sorting its rows in non-decreasing lexicographical order.

Consider \( R_k^i \), the \( i^{th} \) row of \( M_k \). Then every row of \( M_{k-1} \), which is lexicographically smaller than \( R_k^i \) in the first \( j \) coordinates, is present in \( M_{k-1}^{ij} \). However the number of such rows in \( M_{k-1}^{ij} \) is one more than in \( M_k^{ij} \) (the last row also being lexicographically smaller than \( R_k^i \)), which is a contradiction. Thus \( X_k \) is lexicographically smaller than \( X_{k-1} \).

Part (2). We next show that for any three consecutive vectors \( X_{k-1}, X_k, X_{k+1} \) the first index in which \( X_{k-1}, X_k \) differ is strictly smaller than the first index in which \( X_k, X_{k+1} \) differ.

Assume the first index in which \( X_{k-1}, X_k \) differ corresponds to the entry with coordinates \((i, j)\) in both matrices \( M_{k-1}, M_k \). And assume the first index in which \( X_k, X_{k+1} \) differ corresponds to the entry with coordinates \((s, t)\) in both matrices \( M_k, M_{k-1} \).

Notice first that the entry \((i, j)\) is different than \((s, t)\) (it cannot be that \( i = s \) and \( j = t \)), for otherwise \( X_k \) would be lexicographically smaller than \( X_{k+1} \) since \( M_k[i, j] = 0 = M_k[s, t] \) and \( M_{k+1}[i, j] = 1 = M_{k+1}[s, t] \); thereby contradicting Part (1).

For sake of contradiction, suppose that entry \((s, t)\) appears before \((i, j)\) in \( X_k \). Therefore either Case (i). \( s < i \) or Case (ii). \( s = i \) but \( t < j \).
Case (i): By Part (1), we know that:

\[
M_{k-1}[i,j] = 1 \\
M_k[i,j] = 0 \\
M_k[s,t] = 1 = M_{k-1}[s,t] \\
M_{k+1}[s,t] = 0
\]

Where the third equality follows because the submatrices \( M_{k-1}^{i-1,j} \), \( M_{k}^{i-1,j} \) are identical.

Once again, we consider the case when \( k \) is even, the case of odd \( k \) being analogous. If \( k \) is even, then \( M_k \) was obtained from \( M_{k-1} \) by sorting its rows in non-decreasing lexicographical order, and \( M_{k-1} \) and \( M_{k+1} \) are both obtained by sorting the columns in non-decreasing lexicographical order.

By construction, the submatrices \( M_{k-1}^{s,t} \), \( M_k^{s,t} \), \( M_{k+1}^{s,t} \) are all identical except for the entry \([s,t]\). In particular, the column \( C_k^{s,t} \) is lexicographically larger than \( C_{k+1}^{s,t} \), since the first \( s-1 \) entries in these columns are identical and \( M_{k-1}[s,t] = 1, M_{k+1}[s,t] = 0 \)

Since \( M_{k+1} \) is a non-decreasing lexicographic ordering of the columns of \( M_k \), \( C_{k+1}^{s,t} \) appears as a later column in \( M_k \). Call this column \( C_k^h \) for \( h > t \). Since the submatrices \( M_{k-1}^{i-1,j}, M_{k-1}^{i-1,j} \) are identical, and \( s < i \), the first \( s \) entries of \( C_k^h \) are identical to those of \( C_{k-1}^i \). Thus \( C_k^h \) is lexicographically smaller than \( C_{k-1}^i \), yet \( t < h \). This is a contradiction since \( M_{k-1} \) was obtained by lexicographically ordering the columns in non-decreasing order.

Case (ii): Suppose \( s = i \) and \( t < j \). A similar argument holds here as well. Column \( C_{k+1}^t \) appears as \( C_k^h \) for \( h > t \), and since \( t < j \), the first \( i(=s) \) entries of \( C_k^h \) are identical to the first \( i(=s) \) entries of \( C_{k-1}^i \). Thus \( t < h \) yet \( C_{k-1}^h \) is lexicographically smaller than \( C_{k-1}^t \). This contradicts \( M_{k-1} \) having columns in non-decreasing lexicographic order.

Therefore Part (2) implies that after at most \( n^2 \) iterations, \( X_{n^2-1} = X_{n^2} \), and so \( M_{n^2-1} = M_{n^2} \).

We conclude with the following corollary:

**Corollary 5.25.** Cobipartite graphs have LexCycle = 2, and this cycle is reached in fewer than \( n^2 \) LexBFS \(^+\) sweeps.

### 5.3.3 Domino-free Cocomparability Graphs

By Theorem 2.12, all the orderings we are dealing with in this section are LexBFS cocomparability orderings.

**Theorem 5.26.** Domino-free cocomparability graphs have LexCycle = 2.
Proof. Suppose not, and let $G(V,E)$ be a domino-free cocomparability graph. By Corollary 5.3, $G$ must have a loop of even size. Let $\sigma_1, \ldots, \sigma_k$ be a $\text{LexBFS}^+$ cycle with even $k > 2$. We know that such a cycle must exist since the number of $\text{LexBFS}$ orderings of $G$ is finite. For two consecutive orderings of the same parity

$$\sigma_i = u_1, u_2, \ldots, u_n \quad \text{and} \quad \sigma_{i+2} = v_1, v_2, \ldots, v_n \quad \text{for} \quad i \in [k] \mod k,$$

let $\text{diff}(i)$ denote the index of the first (leftmost) vertex that is different in $\sigma_i, \sigma_{i+2}$:

$$\text{diff}(i) = \min_{j \in [n]} \text{ such that } u_j \neq v_j, \text{ and for all } p < j : u_p = v_p$$

Using the cycle $\sigma_1, \sigma_2, \ldots, \sigma_k$ and $\{\text{diff}(i)\}_{i \in [k]}$, we “shift” the start of the cycle to $\pi_1, \pi_2, \ldots, \pi_k$ where $\pi_1$ is chosen as a $\sigma_i$ with minimum $\text{diff}(i)$. If there is a tie, we pick a random ordering $\sigma_i$ of minimum $\text{diff}(i)$ to be the start of the cycle.

Let $a, b$ be the first (leftmost) difference between $\pi_1, \pi_3$. For $\pi_1 = u_1, u_2, \ldots, u_n, \pi_3 = v_1, v_2, \ldots, v_n$, and $j = \text{diff}(1)$, we have $u_i = v_i, \forall i < j$ and $u_j = a, v_j = b$. Thus $a \prec_1 b$ and $b \prec_3 a$.

Let $S = \{u_1, \ldots, u_{j-1}\} = \{v_1, \ldots, v_{j-1}\}$, then $\pi_1[S] = \pi_3[S]$. Since $a$ was chosen in $\pi_1$ and $b$ in $\pi_3$ after the same initial ordering $S$ on both sweeps, it follows that at the time $a$ (resp. $b$) was chosen in $\pi_1$ (resp. $\pi_3$), $b$ (resp. $a$) had the same label, and thus $\text{label}(a) = \text{label}(b)$ at iteration $j$ in both $\pi_1, \pi_3$. In particular $S \cap N(a) = S \cap N(b)$.

Therefore when $a$ was chosen in $\pi_1$, the $^+$ rule was applied to break ties between $a$ and $b$ and so $b \prec_k a$. Similarly, we must have $a \prec_2 b$. We thus have

$$\pi_k = \ldots b \ldots a \ldots \quad \text{and} \quad \pi_2 = \ldots a \ldots b \ldots$$
$$\pi_1 = S, a \ldots b \ldots \quad \text{and} \quad \pi_3 = S, b \ldots a \ldots$$

Since $a \prec_{1,2} b$, choose vertex $c$ as $c = \text{LMPN}(a|_2 b)$. Using the Flipping Lemma on $b$ and $c$, we place vertex $c$ in the remaining orderings as follows

$$\pi_k = \ldots c \ldots b \ldots a \ldots \quad \text{and} \quad \pi_2 = \ldots c \ldots a \ldots b \ldots$$
$$\pi_1 = S, a \ldots b \ldots c \ldots \quad \text{and} \quad \pi_3 = S, b \ldots a \ldots \quad \text{and} \quad b \prec_3 c$$

This gives rise to a bad $\text{LexBFS}$ triple in $\pi_k$ where $c \prec_k b \prec_k a$ and $ca \in E, cb \notin E$. By the $\text{LexBFS}$ 4 Point Condition (Theorem 2.9) and the $C_4$ property (Property 2.13), choose
vertex $d$ as $d = \text{LMPN}(b|a)$, $dc \in E$. We again use the Flipping Lemma on $ad \notin E$ to place $d$ in the remaining orderings

\[
\begin{align*}
\pi_k &= \ldots d \ldots c \ldots b \ldots a \ldots & \pi_2 &= \ldots c \ldots d \ldots a \ldots b \ldots \\
\pi_1 &= S, a \ldots b \ldots c \ldots & \pi_3 &= S, b \ldots a \ldots d \ldots \\
\end{align*}
\]

In $\pi_2$, the Flipping Lemma places $d \prec_2 a$, and by the choice of $c$ as $\text{LMPN}(a|_2 b)$, it follows that no private neighbour of $b$ with respect to $a$ could be placed before $c$ in $\pi_2$. Therefore we can conclude that $c \prec_2 d \prec_2 a$.

It remains to place $d$ in $\pi_1$ and $c$ in $\pi_3$. We start with vertex $d$ in $\pi_1$. We know that $a \prec_1 d$. This gives rise to three cases: Either (i) $c \prec_1 d$, or (ii) $a \prec_1 d \prec_1 b$, or (iii) $b \prec_1 d \prec_1 c$.

(i). If $c \prec_1 d$ then $c \prec_1 d$, so choose vertex $e$ as $e = \text{LMPN}(c|_2 d)$. This means $ed \notin E$, and since $da \notin E$ and $e \prec_2 d \prec_2 a$, it follows that $ea \notin E$ for otherwise if $ea \in E$, $e \prec_2 d \prec_2 a$ contradicts $\pi_2$ being a cocomparability ordering.

Furthermore, by the choice of vertex $c$ as $\text{LMPN}(a|_2 b)$, and the facts that $e \prec_2 c$ and $ea \notin E$, it follows that $eb \notin E$, otherwise $e$ would be a private neighbour of $b$ with respect to $a$ that is to the left of $c$ in $\pi_2$. Using the Flipping Lemma, we place vertex $e$ in the remaining orderings, and in particular, placing vertex $e$ in $\pi_k$ gives rise to a bad \textbf{LexBFS} triple $e, d, c$.

By the \textbf{LexBFS} 4 Point Condition, there must exist a vertex $f$ chosen as $f = \text{LMPN}(d|_1 c)$ and $fe \in E$. Using the same argument as above, one can show that $fc \notin E$ and $eb \notin E$ implies $fb \notin E$, and given the choice of $d$ in $\pi_1$ and $fb \notin E$, then $fa \notin E$. We therefore have the following induced domino $abedf$; a contradiction to $G$ being domino-free.

(ii). If $a \prec_1 d \prec_1 b$, then $a, d, b$ forms a bad \textbf{LexBFS} triple, and thus by the \textbf{LexBFS} 4 Point Condition, choose vertex $e \prec_1 a$ as $e = \text{LMPN}(d|_1 b)$, therefore $eb \notin E$. By the $C_4$ Property, $ea \notin E$. Since $e \prec_1 a$, it follows $e \in S$. But then $ea \in E, eb \notin E$ implies $label(a) \neq label(b)$ when $a, b$ were chosen; a contradiction to $S \cap N(a) = S \cap N(b)$.

(iii). We thus must have $b \prec_1 d \prec_1 c$, in which case we still have a bad \textbf{LexBFS} triple given by $a, d, c$ in $\pi_1$. Choose vertex $e \prec_1 a$ as $e = \text{LMPN}(d|_1 c)$. By the \textbf{LexBFS} $C_4$ Property, $ea \in E$, and since $e \prec_1 a$, it follows $e \in S$, and thus $eb \in E$ since $S \cap N(a) = S \cap N(b)$. Since $\pi_1[S] = \pi_3[S]$, it follows that $e$ appears in $\pi_3$ in $S$, and thus $e$ is the LMPN($d|_3 c$) as well. Therefore $d \prec_3 c$. The orderings look as follows:

\[
\begin{align*}
\pi_k &= \ldots d \ldots c \ldots b \ldots a \ldots & \pi_2 &= \ldots c \ldots d \ldots a \ldots b \ldots \\
\pi_1 &= \ldots e \ldots a \ldots b \ldots d \ldots c \ldots & \pi_3 &= \ldots e \ldots b \ldots a \ldots d \ldots c \ldots \\
\end{align*}
\]

Consider the ordering of the edge $cd$ in $\pi_{k-1}$. If $d \preceq_{k-1} c$, we use the same argument as
above to exhibit a domino as follows: If $d \prec_{k-1} c$, then $d \prec_{k-1,k} c$, so choose a vertex $p = \text{LMPN}(d|_{k,c})$. Therefore $pc \notin E$, and since $cb \notin E$ and $p \prec_k c \prec_k b$, it follows that $pb \notin E$ as well, otherwise we contradict $\pi_k$ being a cocomparability ordering. Moreover, given the choice of vertex $d$ in $\pi_k$ as the $\text{LMPN}(b|_{k,a})$ and the fact that $p \prec_k d, pb \notin E$, it follows that $pa \notin E$ as well. We then use the Flipping Lemma to place vertex $p$ in $\pi_2$. This gives rise to a bad LexBFS triple $p,c,d$ in $\pi_2$. Choose vertex $q \prec_2 p$ as $q = \text{LMPN}(c|_{2,d})$. Again, one can show that $qa,qb \notin E$, and thus the $C_4$ $abcdpq$ are induced, therefore giving a domino; a contradiction to $G$ being domino-free.

Therefore when placing the edge $cd$ in $\pi_{k-1}$, we must have $c \prec_{k-1} d$.

Consider the first (leftmost) difference between $\pi_{k-1}$ and $\pi_1$. Let $S'$ be the set of initial vertices that is the same in $\pi_{k-1}$ and $\pi_1$. By the choice of $\pi_1$ as the start of the cycle $\pi_1, \pi_2, \ldots, \pi_k$, and in particular as the ordering with minimum $\text{diff}(1)$, we know that $|S| \leq |S'|$. Since $S$ and $S'$ are both initial orders of $\pi_1$, it follows that $S \subseteq S'$, and the ordering of the vertices in $S$ is the same in $S'$ in $\pi_1$; $\pi_1[S] \subseteq \pi_1[S']$. In particular vertex $e$ as constructed above appears in $S'$ as the leftmost private neighbour of $d$ with respect to $c$ in $\pi_1$, and thus also in $\pi_{k-1}$, vertex $e$ is $\text{LMPN}(d|_{k-1,c})$. Therefore $d \prec_{k-1,1} c$, a contradiction to $c \prec_{k-1} d$.

Notice that in all cases, we never assumed that $S \neq \emptyset$. The existence of an element in $S$ was always forced by bad LexBFS triples. If $S$ was empty, then case (i) would still produce a domino, and cases (ii), (iii) would not be possible since $e \in S$ was forced by LexBFS.

To conclude, if $G$ is a domino-free cocomparability graph, then it cannot have a cycle of size $k > 2$, and thus must have a 2-cycle.

5.3.4 Trees

Trees are not AT-free graphs, and therefore not cocomparability graphs. Consider for instance the graph below, a claw where every edge is subdivided, this graph is a tree but contains an asteroidal triple $(u,v,w)$.

![Figure 5.5: A subdivided claw.](image)

Since trees are acyclic, it is easy to see that LexBFS and BFS orderings are equivalent on trees since no vertex gets assigned a label of size $\geq 2$, and thus the interaction between
private neighbours and cycles does not play a role for trees as it does for cocomparability graphs. In order to prove that trees also have LexCycle two, we use the following old result of Jordan from 1869 about diametral properties of BFS on trees. In fact, the ordering described by Jordan is a distance layering of the tree.

**Lemma 5.27.** [45] Let \( T(V,E) \) be a tree, \( \sigma_1 \) a BFS ordering of \( T \) ending at vertex \( x \), and \( \sigma_2 = \text{BFS}^+(\sigma_1) \) ending at vertex \( y \). The unique path from \( x \) to \( y \) in \( T \) is a diametral path.

Similar to proper interval graphs, we show that trees reach a cycle relatively quickly, in particular an infinite sequence \( \{\sigma_i\}_{i \geq 1} \) of BFS+ orderings will alternate between two orderings \( \sigma_2 \) and \( \sigma_3 \).

**Theorem 5.28.** Let \( T(V,E) \) be a tree and \( \{\sigma_i\}_{i \geq 1} \) be a sequence of consecutive BFS+ orderings of \( T \), then \( \sigma_2 = \sigma_4 \).

**Proof.** Let \( T \) be a tree of diameter \( k \). Consider a sequence of BFS+ orderings. By Lemma 5.27, the second ordering \( \sigma_2 \) has a first and last vertex, labeled \( x_0 \) and \( x_k \), where \( P = x_0x_1 \cdots x_k \) is a diametral path.

For each vertex \( y \) of \( T \), define \( f(y) = i \) if \( x_i \) is the closest vertex of \( P \) to \( y \). In any BFS \( \sigma \) of \( T \) starting with \( x_0 \), vertices are partitioned into \( k + 1 \) sets \( S_1, \ldots, S_{k+1} \), where \( S_i = \{ v : d(x_0, v) = i \}, i \in [k + 1] \). It is easy to see that the vertices of every partition set \( S_i \) appear consecutively in \( \sigma \). We refine this ordering as follows: Inside each partition set \( S_i \), order the vertices by increasing \( f \)-value. If \( u,v \in S_i \) and \( f(u) < f(v) \) then \( u \prec_\sigma v \), and if \( f(u) = f(v) \), we order \( u,v \) arbitrarily.

Consider a vertex \( x_i \) and let \( N'(x_i) \) be neighbours of \( x_i \) not on \( P \), i.e., \( N(x_i) \setminus \{x_{i-1}, x_{i+1}\} \) (with adjustment of indices when \( i = 0 \) or \( i = k \)).

In a BFS ordering, once \( x_i \) is numbered then all the vertices in \( N'(x_i) \) are labeled \( \sigma(x_i) \) (the position of \( x_i \) in \( \sigma \)), and will receive no other label. This because trees have no cycles, thus every vertex has a lexicographic label of size one. Furthermore, numbering one of these vertices in \( N'(x_i) \) will have no effect on the label of any vertex whose label starts with \( \sigma(x_i) \). Therefore, only a tie breaking rule can order vertices of \( N'(x_i) \) in \( \sigma \).

Let \( N'_2(x_i) \) be the set of vertices at distance 2 from \( x_i \), and whose paths to \( x_i \) go through vertices of \( N'(x_i) \). These vertices are partitioned in \( \sigma \) into consecutive groups based on the vertex connecting them to \( x_i \). Inside each such group, again only a tie breaking rule can be used to order them.

In general, we define subsets of \( N'_p(x_i) \) in a similar manner where a tie breaking rule is the only way to order vertices in \( N'_p(x_i) \).

Notice that if the tie breaking rule is the \( + \) rule, then the ordering inside each \( N'_j(x_i) \) is the reverse (dual) of the previous ordering.
With this observation in mind, our goal is to show that $\sigma_2 = \sigma_4$. To this end we first show that $\sigma_4$ also starts with $x_0$. That is to say that $\sigma_3$, which starts with $x_k$, ends with $x_0$. Applying the previous discussion on $\sigma_3$, with $\sigma_2$ fed to the + rule, we observe that for the set of vertices at distance $i$ from $x_k$, $x_{k-i}$ is the last vertex to be numbered. In particular, since $k$ is the diameter of $T$, $x_0$ is the last vertex of $\sigma_3$ and thus the first vertex of $\sigma_4$. Hence the orders coincide unless, possibly, in places where a tie breaking rule is applied, but when induced on such sets $\sigma_2$ and $\sigma_4$ are both duals of $\sigma_3$, therefore they are identical.

5.4 Conclusion

Towards proving Conjecture 5.2 (i.e., LexCycle(cocomparability graphs) = 2), we showed that a number of sub-classes of cocomparability graphs all have LexCycle two. One good way towards proving Conjecture 5.2 is to start by proving that $k$-ladder-free cocomparability graphs have LexCycle two, for fixed $k$. We define a $k$-ladder to be an induced graph of $k$ chained $C_4$. More precisely, a ladder is a graph $H(V_H,E_H)$ where $V_H = \{x,x_1,x_2,\ldots,x_k,y,y_1,\ldots,y_k\}$ and $E_H = \{(x,y),(x,x_1),(y,y_1)\} \cup \{(x_i,y_i) : i \in [k]\}$, as illustrated in Figure 5.6.

![Figure 5.6: A k-ladder.](image)

Notice that interval graphs are equivalent to 1-ladder-free cocomparability graphs, and domino-free graphs are precisely 2-ladder-free cocomparability graphs. Therefore $k$-ladder-free cocomparability graphs are a good candidate towards proving LexCycle = 2 for cocomparability graphs.

A word on runtime for arbitrary cocomparability graphs: Although the conjecture is still open for cocomparability graphs, experimentally one can show that the convergence happens relatively quickly, but not always, as shown by a graph family $\{G_n\}_{n \geq 2}$ we present below. The graph family, experimentally, takes $O(n)$ LexBFS+ sweeps before converging. We describe an example in the family in terms of its complement, as it is easier to picture the graph, and the LexBFS traversals of the complement are easier to parse. Let $G_n(V = A \cup B, E)$ be a comparability graph on $2n + 2$ vertices, where both $A$ and $B$ are chains, i.e., $A = a_1,a_2,\ldots,a_n,B = x,y,b_1,b_2,\ldots,b_n$, and the only edges in $E$ are of the form $E = \{(a_ia_{i+1}) : i \in [n-1]\} \cup \{(xy),(yb_1)\} \cup \{(b_jb_{j+1}) : j \in [n-1]\}$. The initial comparability
ordering is constructed by collecting the odd indexed vertices first, then the even indexed ones as follows:

- Initially we start $\tau$ with $x, a_1$.
- In general, if the last element in $\tau$ is $a_i$ and $i$ is odd, while $i$ is in a valid range, append $b_i, b_{i+2}, a_{i+2}$ to $\tau$ and repeat.
- If $n$ is even, append $b_n, a_n$ to $\tau$, otherwise append $a_{n-1}, b_{n-1}$ to $\tau$.
- Again while $i$ is in a valid range, we append the even indexed vertices $a_i, b_i, b_{i-2}, a_{i-2}$ to $\tau$.
- Append $y$ to $\tau$.

The ordering $\tau$ as constructed is a transitive orientation of the graph, and thus is a cocomparability ordering in the complement. We perform a series of $\text{LexBFS}^+$ sweeps where $\sigma_1 = \text{LexBFS}^+(\tau)$ in the complement, i.e., the cocomparability graph. Every subsequent $^+$ sweep will proceed to “gather” the elements of $A$ close to each other, resulting in an ordering that once it moves to chain $A$ remains in $A$ until all its elements have been visited. An intuitive way to see why this must happen is to notice in the complement, the vertices of $A$ are universal to $B$ and thus must have a strong pull. Experimentally, this 2-chain graph family takes $O(n)$ $\text{LexBFS}^+$ sweeps before converging. Figure 5.7 below is an example for $n = 6$.

![Diagram](image)

$\tau = x, a_1, b_1, b_3, a_3, a_5, b_5, b_6, a_6, a_4, b_4, b_2, a_2, y$

$\sigma_1 = \text{LexBFS}^+(\tau) = y, a_2, b_2, b_4, a_4, b_6, a_6, b_5, a_5, b_3, a_3, x, b_1$

$\sigma_2 = \text{LexBFS}^+(\sigma_1) = b_1, x, a_1, b_3, a_3, b_5, a_5, b_6, a_6, b_4, a_4, a_2, b_2, y$

$\sigma_3 = \text{LexBFS}^+(\sigma_2) = y, b_2, a_2, a_4, b_4, b_6, a_6, b_5, a_1, a_1, a_3, b_3, x, b_1$

$\sigma_4 = \text{LexBFS}^+(\sigma_3) = b_1, x, b_3, a_3, a_1, a_5, b_5, b_6, a_6, a_4, a_2, b_4, b_2, y$

$\sigma_5 = \text{LexBFS}^+(\sigma_4) = y, b_2, b_4, a_2, a_4, b_4, b_6, a_6, b_5, a_1, a_3, b_3, x, b_1$

$\sigma_6 = \text{LexBFS}^+(\sigma_5) = b_1, x, b_3, a_3, a_5, b_5, a_1, a_5, a_6, a_4, a_2, b_6, b_4, b_2, y$

$\sigma_7 = \text{LexBFS}^+(\sigma_6) = y, b_2, b_4, b_6, a_2, a_4, a_6, a_5, a_1, a_3, b_5, b_3, x, b_1$

$\sigma_8 = \text{LexBFS}^+(\sigma_7) = b_1, x, b_3, a_5, a_1, a_5, a_6, a_4, a_2, b_6, b_4, b_2, y = \sigma_6$

Figure 5.7: $G_6$, A comparability graph; $\tau$ a cocomparability ordering of the complement of $G_6$ and a series of $\text{LexBFS}^+$ of the corresponding cocomparability graph.

**Other Graph Searches:** One could raise a similar “cycle question” for different graph searches; in particular $\text{LexDFS}$, especially since it has been a more useful optimization tool
on cocomparability graphs. Unfortunately, this is not the case as shown by the example below, where $G$ is a cocomparability graph as witnessed by the following cocomparability ordering $\tau = a, c, e, f, g, d, b$. However doing a sequence of $\text{LexDFS}^+$ on $G$ cycles before we even reach a cocomparability ordering, and the cycle has size four.

\[
\begin{align*}
\sigma_1 &= \text{LexDFS}(G) = a, c, d, b, f, g, e \\
\sigma_2 &= \text{LexDFS}^+(\sigma_1) = e, g, f, d, c, a, b \\
\sigma_3 &= \text{LexDFS}^+(\sigma_2) = b, d, c, a, g, e, f \\
\sigma_4 &= \text{LexDFS}^+(\sigma_3) = f, e, g, c, d, b, a \\
\sigma_5 &= \text{LexDFS}^+(\sigma_4) = a, c, d, b, f, g, e = \sigma_1
\end{align*}
\]

Figure 5.8: A sequence of $\text{LexDFS}^+$ orderings on a cocomparability graph, that cycles after 5 iterations, and none of the orderings is a cocomparability ordering.
Chapter 6

Vertex Orderings Based Algorithms on Weighted Graphs

Overview. So far, we have focused on problems on unweighted cocomparability graphs. In this chapter, we ask similar questions on weighted graphs. Given a weighted cocomparability graph, are there “good” vertex orderings that can lead to fast algorithms? We answer the question affirmatively by presenting the first linear time robust algorithm to compute a maximum weighted independent set (WMIS) on cocomparability graphs. We then move to the induced matching problem, where we show how to reduce the problem to an independent set problem via vertex orderings in linear time, on graph classes defined by forbidden orderings on three vertices. We then focus on the class of cocomparability graphs, and use our WMIS algorithm to give the fastest algorithm to compute a maximum weighted induced matching on this graph class (WMIM). The results in this section are based on [48, 43].

6.1 A Robust Linear Time Algorithm for the Weighted MIS on Cocomparability Graphs

It should be no surprise that the structure we gained from a plain graph search is not necessarily useful when weights are involved on edges or vertices. We will see that the cocomparability structure is enough for this problem, in the sense that the umbrella-free property suffices to compute an optimal solution. Once again, we avoid computing the complement of a cocomparability graph but exploit its structure in this algorithm. The maximum weight clique problem on a comparability graph can be computed in linear time by computing a maximum weight chain of a corresponding poset [36].
6.1.1 Overview of The Algorithm

Let $G(V, E, w)$ be a vertex weighted cocomparability graph and let $X \subseteq V$ be the subset of vertices with non-positive weight, i.e., $X = \{v : w(v) \leq 0\}$. Any vertex $v \in X$ that belongs to an independent set $S$ will not increase the total weight of $S$. Therefore if $X \neq \emptyset$, we can restrict ourselves to $G[V \setminus X]$, which is also a cocomparability graph that can easily be computed in $O(m + n)$ time. Thus we can let $G(V, E, w)$ be a cocomparability graph with a positive weight function $w : V \rightarrow \mathbb{R}_{>0}$. Using an algorithm (ccorder($G$)) in [58], we compute a cocomparability order $\sigma$ of $G$ in $O(m + n)$ time where $\sigma = v_1 \prec_\sigma v_2 \prec_\sigma \cdots \prec_\sigma v_n = \text{ccorder}(G)$. We then construct a new permutation $\tau$ of the vertices as follows: we process one vertex at a time according to the order imposed by $\sigma$ from left to right. To each $v_i$ we associate an updated weight $\tilde{w}(v_i)$ and an [independent] set $S_{v_i}$ (containing $v_i$) of total weight $\tilde{w}(v_i)$. The vertices from $v_1$ to $v_i$ are then reordered such that the new ordering is non-decreasing with respect to their updated weights $\tilde{w}$; $\tau_i$ denotes the resulting permutation on the processed vertices $v_1, \ldots, v_i$. In other words, for vertices $v_k, v_j$ ($1 \leq k, j \leq i, k \neq j$),

$$\text{if } v_k \prec_\tau v_j \text{ then } \tilde{w}(v_k) \leq \tilde{w}(v_j). \quad (6.1)$$

Initially $\tau_1$ is just $\{v_1\}$, $\tilde{w}(v_1) = w(v_1)$, and $S_{v_1} = \{v_1\}$. For every vertex $v_i$ ($i > 1$), we scan through $\tau_{i-1}$ from right to left, looking for the rightmost non-neighbour of $v_i$. Let $u$ denote such a vertex (if it exists); $\tilde{w}(v_i)$ and $S_{v_i}$ are then set to

$$\tilde{w}(v_i) = w(v_i) + \tilde{w}(u)$$

$$S_{v_i} = \{v_i\} \cup S_u.$$ 

If no such vertex $u$ exists, then

$$\tilde{w}(v_i) = w(v_i)$$

$$S_{v_i} = \{v_i\}.$$ 

$\tau_i$ is the permutation of $\{v_1, \ldots, v_i\}$ created by inserting $v_i$ into $\tau_{i-1}$ such that (6.1) holds and thus preserving the non-decreasing order of the updated weights. Since the weights are strictly positive, it is easy to see that $\tilde{w}(v_i) = w(v_i) + \tilde{w}(u)$ implies $\tilde{w}(v_i) > \tilde{w}(u)$ and thus also implies $u \prec_\tau v_i$.

If there exists a vertex $x$ in $\tau_{i-1}$ such that $\tilde{w}(x) = \tilde{w}(v_i)$, then $v_i$ is inserted to the right of vertex $x$ in $\tau_{i-1}$. We say that a vertex $v_i$ has been processed as soon as it is inserted into $\tau_{i-1}$ and thus $\tau_i$ is created. When all vertices are processed, we have determined $\tau_n$. We
return $S_z$ as a maximum weight independent set of $G$ and $\tilde{w}(z)$ as its corresponding total weight, where $z$ is the rightmost vertex in $\tau_n$.

We now present the formal description of the algorithm; recall that $\text{ccorder}(G)$ is the procedure presented in [58] to compute a cocomparability order in $O(m + n)$ time.

Algorithm 6.1 CCWMIS

\begin{algorithmic}[1]
\STATE $\sigma \leftarrow \text{ccorder}(G)$ \Comment{$\sigma = (v_1, v_2, \ldots, v_n)$}
\FOR{$i \leftarrow 1$ \TO $n$}
\STATE $\tilde{w}(v_i) \leftarrow w(v_i)$
\STATE $S_{v_i} \leftarrow \{v_i\}$
\ENDFOR
\STATE $\tau_1 \leftarrow (v_1)$
\FOR{$i \leftarrow 2$ \TO $n$}
\STATE Choose $u$ to be the rightmost non-neighbour of $v_i$ with respect to $\tau_{i-1}$
\IF{$u$ exists}
\STATE $\tilde{w}(v_i) \leftarrow w(v_i) + \tilde{w}(u)$
\STATE $S_{v_i} \leftarrow \{v_i\} \cup S_u$
\ENDIF
\STATE $\tau_i \leftarrow \text{insert}(v_i, \tau_{i-1})$ \Comment{Insert $v_i$ into $\tau_{i-1}$ such that $\tau_i$ stays ordered with respect to $\tilde{w}(\cdot)$}
\ENDFOR
\STATE $z \leftarrow$ the rightmost vertex in $\tau_n$
\RETURN $S_z$ and $\tilde{w}(z)$
\end{algorithmic}

Example.

We illustrate the algorithm using a cocomparability graph and a corresponding cocomparability ordering given in Figure 6.1. Table 6.1 shows how $\tau_i$ is created by the algorithm. Recall that the vertices are processed in $\sigma$’s order and vertex $v_i$ is inserted into $\tau_{i-1}$ according to its updated weight.
Figure 6.1: A cocomparability graph with a valid cocomparability ordering; positive weights are given below the vertices, with $0 < \epsilon < \frac{1}{6}$.

<table>
<thead>
<tr>
<th>$v_i$</th>
<th>$u$</th>
<th>$\mathcal{S}_{v_i}$</th>
<th>$\tilde{w}(v_i)$</th>
<th>$\tau_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>-</td>
<td>${v_1}$</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>-</td>
<td>${v_2}$</td>
<td>0.5</td>
<td>$v_2, v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$v_1$</td>
<td>${v_3, v_1}$</td>
<td>$2 - \epsilon$</td>
<td>$v_2, v_1, v_3$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>-</td>
<td>${v_4}$</td>
<td>$3\epsilon$</td>
<td>$v_4, v_2, v_1, v_3$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>$v_1$</td>
<td>${v_5, v_1}$</td>
<td>5</td>
<td>$v_4, v_2, v_1, v_3, v_5$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>$v_3$</td>
<td>${v_6, v_3, v_1}$</td>
<td>$4 - 2\epsilon$</td>
<td>$v_4, v_2, v_1, v_3, v_6, v_5$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>$v_6$</td>
<td>${v_7, v_6, v_3, v_1}$</td>
<td>$6 - 2\epsilon$</td>
<td>$v_4, v_2, v_1, v_3, v_6, v_5, v_7$</td>
</tr>
</tbody>
</table>

Table 6.1: Step by step construction of the ordering $\tau_n$ as computed by Algorithm CCWMIS. At iteration $i$, $v_i$ is being processed to create $\tau_i$; $u$ denotes the rightmost non-neighbour in $\tau_{i-1}$ of the vertex being processed; "-" means no such vertex $u$ exists. By Algorithm CCWMIS $z = v_7$ and thus a maximum weight independent set of the graph in Figure 6.1 is $\mathcal{S}_z = \{v_7, v_6, v_3, v_1\}$ with weight $\tilde{w}(z) = 6 - 2\epsilon$.

### 6.1.2 Correctness and Robustness of the Algorithm

Recall that $\mathcal{S}_{v_i}$ is the set associated with $v_i$ recursively constructed by finding $u$, the rightmost non-neighbour of $v_i$ in $\tau_{i-1}$; in other words $\mathcal{S}_{v_i}$ denotes a set of vertices including $v_i$ whose weights sum up to $\tilde{w}(v_i)$. Therefore $w(\mathcal{S}_{v_i}) = \tilde{w}(v_i)$. For all $i$, $\mathcal{S}_{v_i}$ is initialized to $\{v_i\}$ in Step 4 of Algorithm CCWMIS and is updated accordingly in Step 11.

**Lemma 6.1.** For all $i$, on entry to Step 13 of Algorithm CCWMIS the set $\mathcal{S}_{v_i}$ is an independent set.

**Proof.** The proof is by induction on $i$. For $i = 1$ the set $\mathcal{S}_{v_1} = \{v_1\}$ as initialized in Step 4 is an independent set.

Suppose the lemma holds for all $j \in \{1, \ldots, i - 1\}$ and look at vertex $v_i$. Obviously, if there is no $u$ as defined in Step 8, then $v_i$ is universal to the vertices in $\tau_{i-1}$ and thus we have $\mathcal{S}_{v_i} = \{v_i\}$ as initialized in Step 4. Consider now the case that there is such a vertex $u$
and assume for contradiction that $i$ is the first iteration where the set $S_{v_i}$ computed in Step 11 is not an independent set. At iteration $i$, $v_i$ is being processed; let $v_j$ for $j < i$ (i.e., $u$) be the rightmost non-neighbour of $v_i$ in $\tau_{i-1}$ which means $v_j$ was processed before $v_i$ and thus:

$$v_j \prec v_i$$

$$v_j v_i \notin E$$

$$S_{v_i} = \{v_i\} \cup S_{v_j} \quad (6.2)$$

$$S_{v_j}$$ is an independent set by the induction hypothesis. $\quad (6.3)$

Given (6.2) and (6.3), if $S_{v_i}$ is not an independent set, there must exist a vertex $a \in S_{v_j}$ where $av_i \in E$, and by (6.3), $av_j \notin E$. Furthermore, we know that $a \prec v_j$, since for creating $\tau_j$ vertex $v_j$ was inserted into $\tau_{j-1}$ to the right of its rightmost non-neighbour in $\tau_{j-1}$. Thus the ordering of the triple $(a, v_j, v_i)$ implied by the cocomparability ordering $\sigma$ is $a \prec v_j \prec v_i$. However, the edge $av_i$ flying over $v_j$ contradicts $\sigma$ being a cocomparability order; therefore on entry to Step 13 of Algorithm CCWMIS $S_{v_i}$ is an independent set. 

**Lemma 6.2.** For all $i$, on entry to Step 13 of Algorithm CCWMIS in the graph $G[v_1, \ldots, v_i]$ the set $S_{v_i}$ is of maximum weight among the independent sets containing $v_i$.

**Proof.** The proof is again by induction on $i$. For $i = 1$, the maximum weight independent set in $G[v_1]$ is just $S_{v_1} = \{v_1\}$ with $\bar{w}(v_1) = w(v_1)$.

Suppose now that the claim holds for all $j \in \{1, \ldots, i - 1\}$ and let $v_i$ be the vertex considered. Further, let $u$ be the rightmost vertex that is non-adjacent to $v_i$ in $\tau_{i-1}$. If no such vertex $u$ exists, then $v_i$ is universal to all vertices in $\tau_{i-1}$, and $S_{v_i} = \{v_i\}$ as initialized in Step 4 is the only independent set in $G[v_1, \ldots, v_i]$ that contains $v_i$, and thus has maximum weight among all independent sets containing $v_i$. Suppose now that such a vertex $u$ exists. Since $\tau_{i-1}$ contains a non-decreasing order of the updated weights $\bar{w}(v)$ for $v \in \{v_1, \ldots, v_{i-1}\}$, we thus know that $S_u$ is an independent set containing $u$ with the maximum weight such that $v_i$ can be added to $S_u$ and by Lemma 6.1 maintains independency. If there were another independent set $S_a$ for $a \in \{v_1, \ldots, v_{i-1}\}$ such that $\bar{w}(a) > \bar{w}(u)$ and $a$ is non-adjacent to $v_i$, then $u \prec_{\tau_{i-1}} a$ thereby contradicting Algorithm CCWMIS choosing $a$. Therefore $S_{v_i}$ satisfies the lemma. 

**Lemma 6.3.** For $1 \leq i \leq n$, let $z_i$ be the rightmost vertex of $\tau_i$, then $S_{z_i}$ is a maximum weight independent set in $G[v_1, \ldots, v_i]$.

**Proof.** The proof is again by induction on $i$. For $i = 1$ the lemma is obvious.
Suppose the claim holds for all \( j \in \{1, \ldots, i - 1\} \). Consider \( z_{i-1} \), the rightmost vertex of \( \tau_{i-1} \), and let \( \tilde{w}(z_{i-1}) \) be its corresponding updated weight. Because insertion to \( \tau_{i-1} \) is done rightmost in non-decreasing order:

\[
\tilde{w}(z_{i-1}) \geq \tilde{w}(a), \forall a \in \{v_1, \ldots, v_{i-1}\}.
\]

By the induction hypothesis, \( S_{z_{i-1}} \) is a maximum weight independent set in \( G[v_1, \ldots, v_{i-1}] \).

When processing \( v_i \), we scan \( \tau_{i-1} \) from right to left to insert \( v_i \) and maintain the non-decreasing order of \( \tau_{i-1} \). Either \( \tilde{w}(v_i) \geq \tilde{w}(z_{i-1}) \), in which case, \( v_i \) is the rightmost vertex of \( \tau_i \), and hence \( z_i = v_i \) and \( S_{v_i} \) is a maximum weight independent set in \( G[v_1, \ldots, v_i] \), or \( \tilde{w}(v_i) < \tilde{w}(z_{i-1}) \) and so \( z_i = z_{i-1} \) and \( S_{z_i} = S_{z_{i-1}} \) remains a maximum weight independent set in \( G[v_1, \ldots, v_i] \).

**Theorem 6.4.** Algorithm CCWMIS computes a maximum weight independent set of \( G \) when \( G \) is a cocomparability graph.

**Proof.** This follows directly from Lemma 6.3. \(\square\)

### 6.1.2.1 Robustness of the Algorithm

To make the algorithm robust, it suffices to scan the neighbourhood of the vertex being processed to check if one of its neighbours appears earlier in the sub-solution. More precisely, let \( v_i \) be the vertex being processed, and let \( v_j \) be the rightmost non-neighbour of \( v_i \) in \( \tau_{i-1} \). The algorithm would create \( S_i \) as \( S_j \cup \{v_i\} \). Before creating \( S_j \), we scan \( N(v_i) \) to see if for some \( x \in S_j, xv_i \in E \). This operation takes \( \mathcal{O}(d_{v_i}) \) time. If such an \( x \) exists, we modify the algorithm to return \( x, v_j, v_i \) as an umbrella. We say that the algorithm *breaks* when this umbrella is found. As \( v_i \) is the leftmost such vertex in \( \sigma \), it follows that \( xv_j, v_jv_i \notin E, xv_i \in E \). Since vertices are processed in the \( \sigma \) ordering, this umbrella occurs in \( \sigma \), thus contradicting \( \sigma \) being a cocomparability order.

Suppose the algorithm returns a solution \( S \) even though the graph is not a cocomparability graph. Then it can be shown that \( S \) is still a maximum weight independent set. For this let \( A = \{a_1, a_2, \ldots, a_k\} \) be an optimal solution of \( G \) such that \( a_1 \prec_\sigma a_2 \prec_\sigma \ldots \prec_\sigma a_k \). Note that Lemma 6.2 still holds for \( G \); for otherwise let \( v_i \) be the first vertex to break Lemma 6.2. Suppose \( S_i = S_j \cup \{v_i\} \) is not a maximum weight independent set that contains \( v_i \) in \( G[v_1, \ldots, v_i] \): Either (i) there exists a set \( S_h \) found by the algorithm for some vertex \( v_h \) such that \( S_h \cup \{v_i\} \) has a bigger weight than \( S_i \), or (ii) there exists a set of vertices \( B = \{b_1, b_2, \ldots, b_l\} \) that was not created by the algorithm such that \( B \subseteq \{v_1, \ldots, v_{i-1}\} \) and \( B \cup \{v_i\} \) has a bigger weight than \( S_i \).
(i) Suppose such an $S_h$ exists; then $v_j \prec_{\tau_{i-1}} v_h$ and $S_h$ would have been chosen by the algorithm.

(ii) Suppose that the set $B$ exists as defined above and let $b_1 \prec_{\sigma} b_2 \prec_{\sigma} \ldots \prec_{\sigma} b_t$ be the ordering of the elements of $B$ as processed by the algorithm, i.e., as ordered by $\sigma$. By the choice of $v_i$, $b_t$ satisfies Lemma 6.2, and thus $b_t$ belongs to a set, call it $S_f$, where $b_t = v_t$ for some $\ell \in \{1, \ldots, n\}$, such that $\tilde{w}(v_t) \geq \sum_{s=1}^{t} w(b_s)$. By the choice of $S_f$, we know that $\tilde{w}(v_j) \geq \tilde{w}(v_t) \geq \sum_{s=1}^{t} w(b_s)$, thus $S_i = S_j \cup \{v_i\}$ still satisfies Lemma 6.2.

Therefore when processing the elements of $A$ as ordered by $\sigma$, there exists a set $S_f$ (where $v_f = a_k$) that is a maximum weight independent set in $G[v_1, \ldots, v_f = a_k]$ containing $a_k$. Consequently $S_f$ is an optimal solution, and since the algorithm does not break (i.e., does not return an umbrella as described above), $S_f$ remains an optimal solution throughout all iterations and is returned by the algorithm. The same argument holds if there exists more than one optimal solution. Therefore the algorithm is robust as it either returns an umbrella (showing that the given ordering was not a cocomparability order) or an optimum solution.

### 6.1.3 Implementation and Running Time

To show that Algorithm CCWMIS has complexity $O(m + n)$ time, we have to explain some implementation details. We assume we are given an adjacency list representation of $G(V, E)$. Using the algorithm in [58], ccorder($G$), we compute a cocomparability ordering $\sigma$ of the vertices of $G$, i.e., Step 1 of Algorithm CCWMIS is computed in $O(m + n)$ time. The ordering $\sigma = \{v_1, v_2, \ldots, v_n\}$ is implemented using a doubly linked list.

In the remainder of the analysis, we denote by $u$ the rightmost non-neighbour in $\tau_{i-1}$ of vertex $v_i$, if such a vertex $u$ exists. In order to determine $u$, we create an array $A$ of size $n$ initialized to $A[k] = 0$, $\forall 1 \leq k \leq n$. At iteration $i$ we update $A$ such that $A[j] = i$ if and only if $v_j v_i \in E$; i.e., we keep the $i^{th}$ row of the adjacency matrix of $G$, where $A[j] = i$ stands for $v_i v_j \in E$ and $A[j] < i$ for $v_i v_j \notin E$. Now for determining $u$ it suffices to scan $\tau_{i-1}$ from right to left looking for the first vertex $v_j \in \tau_{i-1}$ such that $A[j] \neq i$; this vertex is then chosen to be $u$ and we create a pointer $p$ for vertex $v_i$ that points to this rightmost non-neighbour $u$ (this pointer is necessary to output the maximum independent set at the end). Once the weight of $v_i$ is updated to $\tilde{w}(v_i) = \tilde{w}(u) + w(v_i)$, we scan $\tau_{i-1}$ from right to left once again to insert $v_i$ into $\tau_{i-1}$. To this end, we update the doubly linked lists pointers appropriately to maintain the increasing order of the weights in the new ordering. Since the pointer $p$ keeps track of vertex $u$, we thus only need to scan the linked list of $\tau_{i-1}$ up to pointer $p$.

Now we can study the complexity of this algorithm. For every $v_i$, we first scan its adjacency list to update $A$, then we scan $\tau_{i-1}$ from right to left to determine $u$, and finally
scan $\tau_{i-1}$ a second time to insert $v_i$ and maintain the non-decreasing order of the updated weights. Setting the array $A$ requires scanning $v_i$’s adjacency list in any order and for every $v_h$ in $v_i$’s list we set $A[h] = i$. This operation takes $O(d_{vi})$ steps where $d_{vi}$ denotes the degree of $v_i$.

Scanning $\tau_{i-1}$ from right to left to determine $u$ requires at most $O(d_{vi})$ checks to see whether $A[j] < i$ for $v_j \in \tau_{i-1}$. If such a vertex $u$ exists then in constant time we update $\tilde{w}(v_i)$; similarly, in constant time we create the above mentioned pointer $p$ that points to $u$. Otherwise, if $u$ does not exist, $v_i$ must be universal to all vertices in $\tau_{i-1}$, and again it will cost at most $O(d_{vi})$ checks to conclude that no such $u$ exists. Consequently, Step 8 of Algorithm CCWMIS takes $O(d_{vi})$ time per vertex $v_i$.

Finally we need to insert $v_i$ into $\tau_{i-1}$ to create $\tau_i$. Since the weights of all vertices are strictly positive, we have $\tilde{w}(v_i) = w(v_i) + \tilde{w}(u) > \tilde{w}(u)$ and thus $u <_{\tau_i} v_i$. Since it takes at most $O(d_{vi})$ steps to determine $u$ and $\tilde{w}(u) < \tilde{w}(v_i)$, it takes at most $O(d_{vi})$ comparisons to insert $v_i$ into $\tau_{i-1}$ when scanning $\tau_{i-1}$ from right to left. Thus Step 13 of Algorithm CCWMIS takes at most $O(d_{vi})$ steps per vertex as well.

Step 15 can easily be determined in constant time with the use of a righthand end pointer of $\tau_{i-1}$. Thus all operations take at most $O(d_{vi})$ time per vertex $v_i$; consequently when all vertices are processed, the for-loop in Steps 7-14 of Algorithm CCWMIS takes at most $O(m + n)$ time in total. As already mentioned, step 1 is done in linear time [58], and clearly the for-loop in Steps 2-6 takes linear time too. Creating $S_z$ in Step 16 also takes linear time as it suffices to start at $\tau_n$’s righthand end pointer to find $z$ and then unravel the $p$ pointers starting with $z$’s $p$ pointer. We therefore conclude with the following theorem.

**Theorem 6.5.** If $G(V, E, w)$ is a weighted cocomparability graph, then a maximum weight independent set of $G$ can be computed in $O(m + n)$ time.

### 6.1.3.1 Minimum Vertex Cover

It is a well known fact that for a maximum independent set $S$ of a graph $G$, the set $V \setminus S$ is a minimum vertex cover of $G$. If we choose $S$ to be the independent set returned by Algorithm CCWMIS we therefore get the following corollary.

**Corollary 6.6.** If $G(V, E, w)$ is a weighted cocomparability graph, with weight function $w : V \rightarrow \mathbb{R}_{>0}$, a minimum weight vertex cover of $G$ can be computed in $O(m + n)$ time.

### 6.2 Weighted Induced Matching via Vertex Orderings

We now switch to a different problem in the second half of the chapter, the induced matching problem. Recall an induced matching in $G$ is a matching that forms an induced subgraph
of \( G \), i.e., every pair of the edges in the induced matching is at distance at least two in \( G \). Induced Matching was introduced in [73] by Stockmeyer and Vazirani, as an extension of the matching problem (known as the marriage problem) to the “risk-free” marriage problem. Stockmeyer and Vazirani showed that maximum induced matching is NP-complete on bipartite graphs. The same result was also proven by Cameron in [5]. The problem remains NP-complete even on bipartite graphs of degree three [56]; and unless \( P = NP \), it is also hard to approximate to within a factor of \( n^{1-\epsilon} \) and \( \Delta G^{1-\epsilon} \) [28]. Since its introduction, the problem has been studied extensively. Induced matchings appear in many real-world applications. For instance, the problem can be used to model uninterrupted communications between broadcasters and receivers [38]; in [1], it was used to model the maximum number of concurrent transmissions in wireless ad hoc networks, and in [53], it was used to extract and discover storylines from search results.

On the parameterized side, the problem was shown to be \( \text{W}[1] \)-hard in general, but planar graphs admit a linear size kernel [63]. And on the tractable side, induced matching is polynomially solvable for a number of graph classes, including trees, weakly chordal, asteroidal-triple free, and circular arc graphs, as well as graphs of bounded clique width [5, 6, 7, 8, 37, 38, 46]. We refer the reader to [28], a survey by Duckworth et al. that contains most of the references and complexity results.

Most of the graph classes for which the problem is tractable have well defined intersection models. One of the main techniques used to show the problem is tractable for a graph class \( \mathcal{G} \), is to show that given an intersection representation of a graph \( G \in \mathcal{G} \), there exists an intersection representation of a graph \( H \in \mathcal{G} \), such that \( L^2(G) = H \), where \( L^2(G) \) is the square of the line graph of \( G \). In other words, one can show that these graph classes are closed under the operation of “taking the square of the line graph” (the \( L^2(\cdot) \) operation).

Since computing a matching (resp. an induced matching) on a graph \( G \in \mathcal{G} \) is equivalent to computing an independent set on \( L(G) \), the line graph of \( G \), (resp. on \( L^2(G) \), the square of \( L(G) \)), by showing closure under \( L^2(\cdot) \), the induced matching problem is tractable on \( \mathcal{G} \) if and only if computing an independent set is tractable on \( \mathcal{G} \).

In this section, we use vertex ordering characterizations to show that certain graph classes are closed under \( L^2(\cdot) \). In particular, one can observe that lexicographic orderings on the edges of a given vertex ordering of \( G \) produces an ordering on the vertices of \( L^2(G) \). Since many graph classes are characterized by vertex orderings, and are closed under the square of the line graph operation, it is natural to ask what these orderings on the edges produce as vertex orderings on \( L^2(G) \); for instance in [3], Brandstädt and Hoàng showed how to compute perfect elimination orderings of \( L^2(G) \) when \( G \) is chordal.

We show that almost all forbidden patterns on three vertices are “preserved” under the \( L^2(\cdot) \) operation. This general theorem shows that graph families with certain vertex orderings
characterizations are closed under the $L^2(\cdot)$ operation; and these orderings of $L^2(G)$ can be computed in linear time in the size of $G$. This property gives, in our opinion, the most natural way to approach this closure operation, and unifies the results on structural graph classes that have relied on geometric intersection models to show closure. Furthermore, being able to compute vertex orderings directly can be exploited algorithmically, since algorithms on the graph classes covered often rely on their vertex ordering characterizations. As a corollary, we get that threshold, interval, and cocomparability graphs - among other classes - are all closed under $L^2(\cdot)$, and their corresponding vertex ordering characterizations are all preserved under $L^2(\cdot)$.

We then end this section by presenting a faster algorithm to compute a maximum weight induced matching for cocomparability graphs. Induced matching on cocomparability graphs has been studied first by Golumbic and Lewenstein in [38], then by Cameron in [6], where they both gave different proofs to show that cocomparability graphs are closed under the $L^2(\cdot)$ operation. In [38], they showed that this closure holds for $k$-trapezoid graphs using the intersection representation of $k$-trapezoid graphs; since cocomparability graphs are the union over all $k$-trapezoid graphs, the result holds for cocomparability graphs as well. Whereas in [6], Cameron used the intersection model of cocomparability graphs (the intersection of continuous curves between two parallel lines [40]) to conclude the result directly; we use cocomparability orderings and the $L^2(\cdot)$ closure to present an $O(mn)$ time algorithm to compute a maximum weighted induced matching for this graph class, which is an improvement over the $O(n^4)$ time algorithm for the unweighted case - a bound one can achieve by computing $L^2(G)$ and running the algorithm in [13] on it.

### 6.2.1 Closure Under $L^2(\cdot)$.

Many well-known classes of graphs can be characterized by vertex orderings avoiding some forbidden patterns, see for example the classification studied in [22] and further studied in [44]. Chordal, interval, split, threshold, proper interval, and cocomparability graphs are a few examples of such graph families, most of which we have already defined. Split graphs are the graphs that can be partitioned into a clique and an independent set, and threshold graphs are the graphs that can be constructed from a single vertex by repeating the following two operations: 1. adding an independent vertex, 2. adding a universal vertex. Both graph classes also have vertex ordering characterizations that we will illustrate in the next section. We will show that almost all patterns on three vertices are preserved under $L^2(\cdot)$, the operation of computing the square of the line graph. To do so, we construct an ordering on the vertices of $L^2(G)$, and thus on the edges of the original graph $G$, by collecting one edge at a time in a lexicographic manner.

Let $L(G)(E, L(E))$ be the line graph of $G(V, E)$, and consider $L^2(G)(E, L^2(E))$. Two
vertices \( e_i, e_j \) in \( L^2(G) \) are adjacent, if and only if they have one of the configurations in \( G \) and \( L(G) \) as shown in Fig. 6.2. In particular, one can see that two vertices are not adjacent in \( L^2(G) \) if their corresponding edges induce a \( 2K_2 \) in \( G \).

\[
\begin{array}{c}
\text{Figure 6.2: Configurations of } e_i, e_j \in E \text{ such that } e_i e_j \in L^2(E), \text{ and their representation in } L(G).
\end{array}
\]

For a given graph \( G = (V, E) \), let \( \sigma = v_1, \ldots, v_n \) be a total ordering of \( V \). Using \( \sigma \), we construct a new ordering \( \pi = e_1, \ldots, e_m \) on \( E \) as follows: For any two edges \( e_i = ab \) and \( e_j = uv \) where \( a \prec_\sigma b \) and \( u \prec_\sigma v \), we place \( e_i \prec_\pi e_j \) if:

\[
\text{Rule } (\star) : \quad e_i \prec_\pi e_j \iff \begin{cases} a \prec_\sigma u & \text{if } a \neq u \\ a = u \text{ and } b \prec_\sigma v & \text{otherwise} \end{cases}
\]

We write \( \pi^*(\sigma) \) to denote the ordering constructed using the \( \star \) rule on \( \sigma \). The ordering \( \pi^*(\sigma) \) is the lexicographic ordering of \( E \) induced by \( \sigma \), similar to the one used on chordal graphs in [3]. We will use \( \phi^* \) to denote the ordering \( \pi^*(\sigma) \) on \( L(G) \), including the edges \( L(E) \), not just the vertices of \( L(G) \); and use \( \sigma^* \) to denote the ordering \( \pi^*(\sigma) \) on \( L^2(G) \), including the edges \( L^2(E) \).

**Theorem 6.7.** Given a graph \( G(V, E) \), its corresponding \( L^2(G)(E, L^2(E)) \), and \( \sigma \) an ordering of \( V \), if \( \sigma \) is \( p_i \)-free for a pattern \( p_i \) in Fig. 6.3, then \( \sigma^* \) is \( p_i \)-free as well.

Given \( \sigma = v_1, v_2, \ldots, v_n \), one can think of the (\( \star \)) rule as a lexicographic ordering of the edges, where the lexicographic label of every edge \( v_i v_j \) is \( ij \), and \( v_i v_j \prec_\pi v_s v_t \) if the label \( ij \) is lexicographically smaller than \( st \).

\[
\begin{array}{c}
\text{Figure 6.3: A list of forbidden patterns on three vertices. Dashed lines denote non-edges.}
\end{array}
\]

Notice that the pattern \( p_4 \) forms an umbrella over the middle vertex. Thus the \( p_4 \)-free orderings are precisely cocomparability orderings.

Before proving Theorem 6.7, we give an example where a pattern on three vertices is not preserved under the \( L^2(\cdot) \) operation. Consider the pattern \( q \) illustrated below.
Chapter 6. Vertex Orderings Based Algorithms on Weighted Graphs

Figure 6.4: Pattern $q$ on $x, y, z$ (left), and a graph with a $q$-free vertex ordering (right).

An ordering $\sigma$ of a graph $G$ can be pattern $q$-free, yet $\pi^*$ is not, as shown in the graph in Figure 6.4, where $\sigma = a, c, d, b, e, f$ is $q$-free, and using the ($\ast$) rule, we have $e_1 \prec_{\sigma^*} e_2 \prec_{\sigma^*} e_3$, with $e_1 = ab, e_2 = cd, e_3 = ef$ and $e_1 e_2, e_1 e_3 \in E(L^2(G)), e_2 e_3 \notin E(L^2(G))$.

We remind the reader the definition of $\text{edist}_G(e, f)$ between edges $e$ and $f$ is the minimum distance over all shortest paths connecting an endpoint of $e$ to an endpoint of $f$.

**Proof of Theorem 6.7.** The proof is by contradiction, where we show if $\sigma^*$ has an induced triple that satisfies a given pattern, then $\sigma$ must also contain such a pattern. Call such a triple $e_1 \prec_{\sigma^*} e_2 \prec_{\sigma^*} e_3$.

For pattern $p_1$ for instance, in Fig. 6.3 above, we have:

Throughout the proof, let $e_1 = ab, e_2 = cd$, and $e_3 = ef$. Without loss of generality, suppose $a \prec \sigma b, c \prec \sigma d$, and $e \prec \sigma f$. Thus $a \preceq_\sigma c \preceq_\sigma e \prec_\sigma f$.

When a triple of vertices $x, y, z$ induces a pattern $p_1$, we write $x, y, z \equiv p_1$. For the ordering $\prec_\sigma$ associated with $\sigma$, we drop the subscript and use $<$ instead, whereas we write $\prec_*$ to refer to the ordering $\prec_{\sigma^*}$. Recall that two vertices in $\sigma^*$ are not adjacent iff they induce a $2K_2$ in $G$, and similarly, adjacent vertices in $\sigma^*$ must have $\text{edist}_G \leq 1$ (Fig. 6.2).

- $p_1$: The configuration in $p_1$ implies the following adjacencies in $G$:

  \[
  \begin{align*}
  ac, ad \notin E & \quad \text{and} \quad bc, bd \notin E \\
  \text{edist}_G(e_1, e_3) \leq 1 & \quad \implies \quad ae \vee af \vee be \vee bf \in E \\
  \text{edist}_G(e_2, e_3) \leq 1 & \quad \implies \quad ce \vee cf \vee de \vee df \in E
  \end{align*}
  \]

Using the ($\ast$) rule, $e_1 \prec_* e_2 \prec_* e_3$ implies $a \prec e \preceq c \preceq e$. Since $ac \notin E$, it follows that vertices $a$ and $c$ do not share a common neighbour to the right of $c$ in $\sigma$, otherwise the triple would induce a $p_1$ pattern.

We have $c \preceq e$. Suppose first that $c = e$, then $ae, be \notin E$ by (6.26). By (6.27), this leaves at least one of $af, bf \in E$. Since $a \prec e \prec f$, it follows that $af \notin E$ otherwise
$a, e, f \equiv p_1$. Thus $bf \in E$. Notice that $e \prec b$ for otherwise $b, e, f \equiv p_1$, and also $b \prec f$ otherwise $a, f, b \equiv p_1$. Thus $e \prec b \prec f$, but this implies $e, b, f \equiv p_1$.

Thus $c \neq e$, and $a \prec c \prec e \prec f$.

Next, we show that $ae \notin E$ and $af \notin E$. Suppose first that $ae \in E$, then $ce \notin E$ otherwise $a, c, e \equiv p_1$. Furthermore $ce \notin E$ implies $cf \notin E$ otherwise $c, e, f \equiv p_1$. Thus by (6.28), either $de \in E$ or $df \in E$ or both. Suppose first $df \in E$. We place $d$ in $\sigma$. Notice that $d \prec f$ otherwise $c, f, d \equiv p_1$. Next we place $d$ with respect to $e$. Either $e \prec d$ or $d \prec e$.

Suppose that $e \prec d$: Then $de \notin E$ otherwise $c, e, d \equiv p_1$. Thus, again by (6.28), either $cf \in E$ or $df \in E$ or both. If $cf \in E$, then $c, e, f \equiv p_1$ since $c \prec e \prec f$. Thus $cf \notin E$ and $df \in E$. But then $e, d, f \equiv p_1$ since we showed $d \prec f$. In all cases we produce a $p_1$ pattern, therefore $e \notin d$.

Thus $d \prec e$; this implies $de \notin E$ otherwise $a, d, e \equiv p_1$. Thus by (6.28), either $cf \in E$ or $df \in E$ or both. If $cf \in E$ then $c, e, f \equiv p_1$. If $df \in E$ then $d, e, f \equiv p_1$. In all cases we find a $p_1$ pattern in $\sigma$. Therefore $ae \notin E$, and since $e \prec f$, it follows that $af \notin E$ as well, otherwise $a, e, f \equiv p_1$.

We have the following:

$$a \prec c \prec e \prec f \tag{6.7}$$
$$ae \notin E \quad \text{and} \quad af \notin E \tag{6.8}$$

We now place vertex $b$ in $\sigma$. Either (i) $b \prec c$ or (ii) $c \prec b$. Clearly $b \neq e$ since $ab \in E, ae \notin E$.

(i) Suppose first $b \prec c$. We have $bc \notin E$, thus $b$ and $c$ do not share any common neighbours to the right of $c$, otherwise $b, c$ and this common right neighbour would induce a $p_1$ pattern. By (6.27) and (6.30), either $be \in E$ or $bf \in E$, or both. Notice first that $b \neq f$ by (6.30). If $be \notin E$ then $bf \in E$ is forced. This implies $b, e, f \equiv p_1$, since by assumption $b \prec c$, and we have $c \prec e \prec f$. Thus $be \in E$. This implies $ce \notin E$ otherwise $b, c, e \equiv p_1$. We place $d$ in the ordering, given that $cd \in E$ and $c \prec d$. Either $d \prec e$ or $e \prec d$. If $d \prec e$ then $b \prec c \prec d \prec e$. And since $bd \notin E$, it follows $de \notin E$ otherwise $b, d, e \equiv p_1$. Thus either $cf \in E$ which implies $c, e, f \equiv p_1$, or $df \in E$ which implies $d, e, f \equiv p_1$. Therefore $e \prec d$. This gives two cases; either $d \prec f$ or $f \prec d$. If the former, then $ed \notin E$ for otherwise $c, e, d \equiv p_1$. Furthermore $ed \notin E$ implies $df \notin E$, otherwise $e, d, f \equiv p_1$. Thus $cf \in E$, but then $c, e, f \equiv p_1$. Thus $f \prec d$, in which case $cf \notin E$ for otherwise $c, e, f \equiv p_1$. Furthermore, $cf \notin E$ implies $fd \notin E$ otherwise $c, f, d \equiv p_1$. Thus $ed \in E$, but then $c, e, d \equiv p_1$. In all cases, we produce a
p_1 in \sigma. Thus c \prec b.

\(\textbf{ii}\) We thus have c \prec b and either e \prec b or b \prec e. Suppose first that e \prec b, then be \notin E otherwise a,e,b \equiv p_1. Thus bf \in E. If b \prec f then e,b,f \equiv p_1, and if f \prec b, then a,f,b \equiv p_1 since af \notin E by (6.30). Thus b \prec e. We now have the following ordering:

\[ a \prec c \prec b \prec e \prec f \] 

(6.9)

If be \notin E, then bf \notin E otherwise b,e,f \equiv p_1. Thus be \in E. This implies ce \notin E otherwise c,b,e \equiv p_1. This in turn implies cf \notin E otherwise c,e,f \equiv p_1. Thus either de \in E or df \in E. Notice that in both cases, it cannot be that f \prec d otherwise ed \in E implies c,e,d \equiv p_1 and df \in E implies c,f,d \equiv p_1. Thus d \prec f. If e \prec d \prec f, then ed \notin E implies df \in E, which in turn implies e,d,f \equiv p_1. On the other hand, ed \notin E implies c,e,d \equiv p_1. Thus d \prec e. If b \prec d \prec e, then since bd \notin E and be \in E, it follows that de \notin E implies df \in E, which in turns leads to d,e,f \equiv p_1. Thus d \prec b, in which case de \notin E otherwise d,b,e \equiv p_1. de \notin E implies df \in E but then d,e,f \equiv p_1. In all cases, we always produce a p_1 in \sigma.

- **p_2**: Given the ordering in \sigma^* and \sigma, it follows that a \preceq c \prec e \prec f. We place vertex b in \sigma. If b \prec e, then a,b,e \equiv p_2 since e_1e_3 \notin L^2(E), and thus ae,be,af, bf \notin E. Thus e \prec b. We next place vertex f in \sigma. If f \prec b then e,f,b \equiv p_2 and if b \prec f then a,b,f \equiv p_2. Thus the claim of the theorem holds for p_2.

- **p_3**: For this pattern, it suffices to notice that a \prec e \prec f always produces a p_3 in \sigma.

- **p_4**: This configuration in \sigma^* implies the following adjacencies in G:

\[ \begin{align*}
ac, ad \notin E & \quad \text{and} \quad bc, bd \notin E \quad (6.10) \\
ec, de \notin E & \quad \text{and} \quad cf, df \notin E \quad (6.11) \\
\text{edist}_G(e_1, e_3) \leq 1 \quad \implies \quad a \vee e \vee be \vee bf \in E \quad (6.12) \\
a \prec c \prec e \prec f \quad (6.13)
\end{align*} \]

Let’s begin by trying to satisfy (6.34). Notice first that ae,af \notin E, for otherwise a,c,e \equiv p_4 or a,c,f \equiv p_4 would form an umbrella over c. This also implies that e \neq b and f \neq b since ab \in E and a \prec c \prec e \prec f. Thus either be \in E or bf \in E. We consider two cases: either b \prec c or c \prec b. If the former then b,c,e \equiv p_4 if be \in E or b,c,f \equiv p_4 if bf \in E. If the latter then a,c,b \equiv p_4. In all cases, we would form a p_4 in \sigma. Therefore if \sigma^* has a p_4 pattern, then \sigma must have a p_4 pattern.


- \( p_5 \): Since \( e_1, e_2, e_3 \) form a stable set, it follows that \( a < c < e \) and \( ac \land ae \land ce \notin E \), thus \( a, c, e \equiv p_5 \).

Once again, for all five patterns, if \( \sigma^* \) has a \( p_i \) induced pattern then so does \( \sigma \).

\[ \square \]

### 6.2.2 Vertex Orderings of \( L^2(G) \) in Linear Time

**Implementation:** Since the \( \star \) rule is just a lexicographic ordering on the edges, it is easy to compute in linear time using standard ordering techniques, as follows: Suppose \( G \) is given as adjacency lists, and let \( \sigma = v_1, v_2, \ldots, v_n \) be a total ordering of \( G \). For every \( w \in V \), we sort the adjacency list of \( w \) according to \( \sigma \). That is for every pair \( v_i, v_j \in N(w) \), if \( v_i \prec_{\sigma} v_j \) then \( v_i \) appears before \( v_j \) in \( N(w) \). This can be done in \( O(m + n) \) time, as we did in Chapter 3 (cf. Section 3.6). We next construct the ordering \( \pi^*(\sigma) \) on the edges of \( G \) as follows: Initially \( \pi^*(\sigma) \) is empty. We scan \( \sigma \) from left to right, for every \( v_i \) in \( \sigma \), and every neighbour \( v_j \) of \( v_i \) such that \( i < j \), we append \( e_k = v_iv_j \) to \( \pi^*(\sigma) \). Adding these edges requires scanning \( N(w) \) for every \( w \in V \). Thus this process takes \( O(m + n) \) time. It is easy to see that this construction satisfies the \( (\star) \) rule. The ordering \( \pi^*(\sigma) \) we produce at the end of this process is precisely the ordering of the \( \text{vertices} \) of \( \pi^*(\sigma), \phi^*, \) and \( \sigma^* \). Recall that these three orderings differ only in their edge sets and not on the ordering of their vertices.

Therefore if a graph family \( \mathcal{G} \) is characterized by the absence of patterns listed in Fig. 6.3, then if computing an independent set on \( G \in \mathcal{G} \) is tractable, and uses the vertex ordering characterization of \( \mathcal{G} \), it follows that computing a maximum induced matching on \( G \) is also tractable and reduces to computing an independent set on \( L^2(G) \in \mathcal{G} \) using \( \sigma^* \).

Since we focus on graph families with forbidden patterns on three vertices (as shown in Fig. 6.3), we illustrate the consequences of Theorem 6.7 by listing in Table 6.2 a number of graph families characterized by the absence of the patterns listed in Fig. 6.3. Corollary 6.8 below follows immediately. For chordal graphs, Brandstädter and Hoàng gave a stronger result where they showed that not only is \( \sigma^* \) a \( p_1 \)-free ordering, but that it is also a \textbf{LexBFS} ordering [3].

<table>
<thead>
<tr>
<th>( \mathcal{G} )</th>
<th>Forbidden Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold</td>
<td>( p_1 ) and ( p_2 )</td>
</tr>
<tr>
<td>Interval</td>
<td>( p_1 ) and ( p_4 )</td>
</tr>
<tr>
<td>Split</td>
<td>( p_1 ) and ( p_3 )</td>
</tr>
<tr>
<td>Cocomparability</td>
<td>( p_4 )</td>
</tr>
<tr>
<td>Chordal</td>
<td>( p_1 )</td>
</tr>
</tbody>
</table>

Table 6.2: \( G \in \mathcal{G} \) iff \( \exists \sigma \) of \( G \) that does not have any corresponding induced pattern [22].
Corollary 6.8. Vertex ordering characterizations of threshold, interval, split, cocomparability, and chordal graphs are all closed under the $L^2(\cdot)$ operation, and computing these orderings of $L^2(\cdot)$ can be done in linear time in the size of $G$.

Corollary 6.9. Maximum cardinality induced matching on the graphs in Table 6.2 can be computed in linear time.

Corollary 6.10. Any graph class characterized by a vertex ordering forbidding a subset of the patterns in Figure 6.3 is closed under the $L^2(\cdot)$ operation.

6.2.3 Application: $\mathcal{O}(mn)$ Time WMIM on Cocomparability Graphs

In [13], Corneil et al. presented a simple greedy algorithm for the maximum cardinality independent set problem on cocomparability graphs, that uses the lifting framework mentioned previously. It works as follows:

- Compute $\sigma$ a cocomparability ordering in linear time using $ccorder(G)$ [58].
- Compute $\tau = \text{LexDFS}^+(\sigma)$ in linear time using our algorithm in Chapter 3.
- Greedily collect an independent set by scanning $\tau$ from right to left.

Since cocomparability orderings are preserved with the $\star$ rule, and since $\text{LexDFS}^+$ also preserves cocomparability orderings, it is only natural to ask whether we can use either the $\star$ rule to compute $\text{LexDFS}$ cocomparability orderings in linear time for $L^2(G)$. If so, then the unweighted induced matching problem on cocomparability graphs would be solved in linear time. Unfortunately, this is not the case, as we show with the bad example below. Consider $\sigma$, the $\text{LexDFS}$ cocomparability ordering below (Fig. 6.5). When computing $\pi^+(\sigma)$ we get the following ordering on the edges:

$$ab \prec ba \prec be \prec ac \prec c\beta \prec cd \prec \beta e \prec ed$$

When adding the appropriate square edges to the above ordering, we get a bad LexDFS triple $e_1 \prec e_2 \prec e_3$, where $e_1 = ab, e_2 = cd, e_3 = ed$, and $e_1e_3, e_2e_3 \in E(L^2(G)), e_1e_2 \notin E(L^2(G))$, yet every vertex in the ordering between $e_1$ and $e_2$ is adjacent to $e_3$, thus $e_2$ does not have a LexDFS private neighbour with respect to $e_3$ (Theorem 2.11), so $\pi^+(\sigma)$ is not a LexDFS ordering despite $\sigma$ being one.
A second attempt is to use our algorithm 6.1, CCWMIS which does not require the LexDFS structure, only cocomparability orderings. This approach yields an $O(mn)$ time algorithm for the weighted version of the problem.

Let $G(V, E, w)$ be an edge weighted cocomparability graph where $w : E \to \mathbb{R}_{>0}$. Since cocomparability graphs are closed under the $L^2(\cdot)$ operation, $L^2(G)(E, L^2(E), w)$ is a vertex weighted cocomparability graph. We compute a maximum weight independent set of $L^2(G)$ as shown in Algorithm CCWMIM below.

**Algorithm 6.2** Cocomparability Weighted Maximum Induced Matching (CCWMIM)

**Input:** $G(V, E, w)$ an edge weighted cocomparability graph where $w : E \to \mathbb{R}_{>0}$

**Output:** A maximum weight induced matching of $G$

1: Compute $\sigma \leftarrow \text{ccorder}(G)$ a cocomparability ordering of $G$  \hspace{1cm} $\triangleright \sigma = (v_1, v_2, \ldots, v_n)$
2: Compute $\pi^*(\sigma) = e_1, e_2, \ldots, e_m$ using $\star$ rule. $\triangleright$ This yields a cocomparability ordering of $L^2(G)$.
3: Use Algorithm CCWMIS and $\pi^*(\sigma)$ to compute a maximum weight independent set of $L^2(G)$

By Theorem 6.5, Algorithm CCWMIS takes $O(m + n)$ time. Thus, CCWMIS will take $O(|E| + |L^2(E)|)$ time on $L^2(G)$. When $G$ is dense, CCWMIS on $L^2(G)$ takes $O(n^4)$ time.

Before giving a careful implementation and analysis to achieve an $O(mn)$ running time, we illustrate Algorithm CCWMIM in Fig. 6.6 and Fig. 6.7, which shows an edge weighted cocomparability graph $G(V, E, w)$, $\sigma$ a cocomparability ordering of $G$, $\phi^*$ an ordering of $L(G)$ constructed by the $\star$ rule, and $\sigma^*$ the corresponding ordering on the vertices of $L^2(G)$. Table 6.8 shows the step by step execution of CCWMIS using $\pi^*(\sigma)$ as the ordering computed in Step 1 of the algorithm.
Chapter 6. Vertex Orderings Based Algorithms on Weighted Graphs

Figure 6.6: An edge weighted $G$ and $\sigma$ a cocomparability ordering of $G$.

Figure 6.7: Left: $\phi^*$ an ordering of $L(G)$ produced by the ($\star$) rule. Right: $\sigma^*$ a cocomparability ordering of $L^2(G)$, produced by the ($\star$) rule. The red edges are the square edges of $\phi^*$.

<table>
<thead>
<tr>
<th>$e_i$</th>
<th>$u$</th>
<th>$S_{e_i}$</th>
<th>$w(S_{e_i})$</th>
<th>$\tau_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1 = ab$</td>
<td>-</td>
<td>${e_1}$</td>
<td>5</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$e_2 = be$</td>
<td>-</td>
<td>${e_2}$</td>
<td>1</td>
<td>$e_2, e_1$</td>
</tr>
<tr>
<td>$e_3 = ed$</td>
<td>-</td>
<td>${e_3}$</td>
<td>1.5</td>
<td>$e_2, e_3, e_1$</td>
</tr>
<tr>
<td>$e_4 = be$</td>
<td>-</td>
<td>${e_4}$</td>
<td>1</td>
<td>$e_2, e_4, e_3, e_1$</td>
</tr>
<tr>
<td>$e_5 = dc$</td>
<td>-</td>
<td>${e_5}$</td>
<td>1</td>
<td>$e_2, e_4, e_5, e_3, e_1$</td>
</tr>
<tr>
<td>$e_6 = cu$</td>
<td>-</td>
<td>${e_6}$</td>
<td>1</td>
<td>$e_2, e_4, e_5, e_6, e_3, e_1$</td>
</tr>
<tr>
<td>$e_7 = uv$</td>
<td>$e_1$</td>
<td>${e_1, e_7}$</td>
<td>7</td>
<td>$e_2, e_4, e_5, e_6, e_3, e_1, e_7$</td>
</tr>
</tbody>
</table>

Figure 6.8: A step by step illustration of Algorithm CCWMIS on the graph in Fig. 6.6 and Fig. 6.7. The algorithm returns $S_z = S_{e_7} = \{e_7, e_1\}$ of maximum weight 7.

6.2.3.1 Implementation & Analysis of CCWMIM

Suppose the graph $G(V, E, w)$, where $w : E \rightarrow \mathbb{R}_{>0}$, is given as adjacency lists. We compute $\sigma = v_1, \ldots, v_n$ in $O(m + n)$ time using $\text{ccorder}(G)$, the algorithm in [58]. We construct $\pi^*(\sigma)$ in $O(m + n)$ time.

Notice that we cannot use $\phi^*$ as input for the CCWMIS algorithm, since $\phi^*$ is not necessarily a cocomparability ordering. In fact, $L(G)$ is not necessarily a cocomparability graph; just consider the line graph of any large clique $K_p, p > 4$. Notice also that the square edges in $\sigma^*$ are necessary for Step 7 of the algorithm, when looking for a rightmost non-neighbour in $\tau_{i-1}$. 
We begin by looking at forbidden configurations of induced $2K_2$s in cocomparability orderings. Let $\sigma = v_1, \ldots, v_n$ be a cocomparability ordering. Let $e_i = ab$ and $e_j = uv$ be two edges that induce a $2K_2$ in $G$. Without loss of generality, suppose $a \prec_\sigma b$ and $u \prec_\sigma v$. Since $\sigma$ is a cocomparability ordering, the configurations of $e_i, e_j$ that have either $a \prec u \prec b \prec v$ or $a \prec u \prec v \prec b$ as orderings cannot occur in $\sigma$, for otherwise $\sigma$ would have an umbrella. This leaves the following configurations of the edges without umbrellas: $a \prec b \prec u \prec v$ or $u \prec v \prec a \prec b$.

Without loss of generality, suppose $a \prec_\sigma b \prec_\sigma u \prec_\sigma v$. Using the (⋆) rule, this configuration always forces $e_i \prec_\pi e_j$, i.e., $ab \prec_\pi uv$. Therefore, when we run Algorithm CCWMIS on $\pi^* = e_1, \ldots, e_m$, we process elements of $\pi^*$ from right to left, and thus we process $e_i = ab$ before processing $e_j = uv$.

Let $\tau = f_1, \ldots, f_m$ be the new ordering being constructed by the algorithm CCWMIS using $\pi^*$ as the ordering computed in Step 1. Initially, as per the algorithm, $\tau_1 = e_1$. In general, at iteration $i$, let $\tau_{i-1} = f_1, \ldots, f_{i-1}$ be the ordering constructed thus far. Suppose $e_i$ is the edge being processed. In Step 7 of Algorithm CCWMIS, looking for the rightmost non-neighbour of $e_i$ in $\tau_{i-1}$ is equivalent to looking for an edge $e$ that forms an induced $2K_2$ with $e_i$ in $\sigma$, such that $e$ is to the left of $e_i$ in $\sigma$. When processing vertex $e_i$ in $\pi^*$, we scan $\tau_{i-1}$ to find the rightmost non-neighbour of $e_i$ in $\tau_{i-1}$. Suppose such a vertex exists, and call it $f_j$. Since we are working in $L^2(G)$, to check if two vertices in $L^2(G)$ are adjacent, we need to check whether these edges are incident in $G$, or are at distance at most two in $L(G)$, as shown in Fig. 6.2. We proceed as follows.

Both $\sigma$ and $\pi^*$ are implemented using doubly linked lists.

We construct three arrays $A$, $B$ and $F$ of sizes $n, n, m$ respectively. Arrays $A$ and $B$ can both be merged into a single one, but to illustrate that we deal with end points of every edge separately, we chose to present the arrays individually. These two arrays are initialized to zero; $A[t] = B[t] = 0, \forall t \in [n]$, whereas $F$ is an array of pointers.

Every vertex $v_i$ in $\sigma = v_1, \ldots , v_n$ has a pointer to $A[t]$ and $B[t]$. Similarly, every vertex $e_i$ in $\pi^*$ has a pointer to $F[i]$. We sometimes abuse notation and talk about $A[w]$ to mean the position in array $A$ that vertex $w$ in $\sigma$ points to (Similarly for $B[w]$). Furthermore, when we talk about vertex $e_i = ab$ in $\pi^*$, we always assume that $a \prec_\sigma b$.

For every vertex $e_i = ab$ in $\pi^*$, its corresponding entry $F[i]$ has four pointers $p_i^a, p_i^b, q_i^a, q_i^b$ where $s$ (resp. $h$) is the position of vertex $a$ (resp. $b$) in $\sigma$. These four pointers $p_i^s, p_i^h, q_i^s, q_i^h$ point respectively to $A[s], A[h]$ and $B[s], B[h]$. When processing vertex $e_i$, where $e_i = ab$, we update $A$ as follows: For every neighbour $z$ of vertex $a$, we set $A[z] = i$. Similarly, for every neighbour $z$ of vertex $b$, we set $B[z] = i$. These updates to arrays $A$ and $B$ guarantee that every non-neighbour $w$ of $a$ has $A[w] \neq i$ and every non-neighbour $w$ of $b$ has $B[w] \neq i$. 

Therefore, for every edge \( f_j = v_i v_k \) in \( G \) that forms an induced \( 2K_2 \) with \( e_i = ab \), the following (†) condition holds:

\[
A[t] \neq i \land A[k] \neq i \land B[t] \neq i \land B[k] \neq i \quad (†)
\]

Thus, in order to find the rightmost non-neighbour of \( e_i \) in \( \tau_{i-1} \), we scan \( \tau_{i-1} \) from right to left, and for every vertex \( f_j = v_i v_k \) we encounter, we check if one of \( A[t], A[k], B[t], B[k] \) is equal to \( i \). We return the first vertex in \( \tau_{i-1} \) we encounter whose endpoints in \( G \) satisfy condition (†) above as the rightmost non-neighbour of \( e_i \) in \( \tau_{i-1} \). Updating arrays \( A \) and \( B \) requires \( O(\deg(a) + \deg(b)) \) time. When scanning \( \tau_{i-1} \), for every vertex \( f_j = v_i v_k \) in \( \tau_{i-1} \), we use the pointers \( p^j_i, p^k_i, q^j_i, q^k_i \) in \( F[j] \) to access \( A[t], A[k], B[t], B[k] \). Checking these four entries takes constant time using the pointers provided.

It remains to analyze the number of constant checks we do, i.e., how many \( f_j \) vertices we check. In particular, this reduces to bounding the degree of \( e_i \) in \( L^2(G) \).

Let \( \deg_1(e_i) \) denote the degree of \( e_i \) in \( L(G) \), and \( \deg_2(e_i) \) denote the degree of \( e_i \) in \( L^2(G) \). We have the following:

**Proposition 6.11.** For a given edge \( e_i = ab \) in \( G \), we have

\[
\deg_2(e_i) \leq \sum_{v:av \in E \atop v \neq b} \deg(v) + \sum_{v: bv \in E \atop v \neq a} \deg(v)
\]

**Proof.** It is clear that for a given edge \( e_i = ab \), \( \deg_1(e_i) = \deg(a) + \deg(b) - 2 \). On the other hand, when computing \( \deg_2(e_i) \), we take into account the degree of any vertex at distance at most two from either \( a \), or \( b \) in \( G \). In particular, the following holds:

\[
\deg_2(e_i) \leq \deg_1(e_i) + \sum_{v:av \in E \atop v \neq b} (\deg(v) - 1) + \sum_{v: bv \in E \atop v \neq a} (\deg(v) - 1)
\]

\[
\leq \deg(a) + \deg(b) - 2 + \left[ \sum_{v:av \in E \atop v \neq b} \deg(v) \right] - \deg(a) + 1 + \left[ \sum_{v: bv \in E \atop v \neq a} \deg(v) \right] - \deg(b) + 1
\]

\[
\leq \sum_{v:av \in E \atop v \neq b} \deg(v) + \sum_{v: bv \in E \atop v \neq a} \deg(v)
\]

The first inequality avoids counting edges twice, in particular if \( a, b, \) and \( v \) form a triangle. The -1s in the first inequality are to avoid counting the edge \( av \) in \( \deg(v) \), for every \( v \in N(a) \), similarly for \( b \). The +1s in the second inequality are for not counting edge \( ab \) for both \( a \) and \( b \) in \( \deg(a) \) and \( \deg(b) \). \( \square \)

When scanning \( \tau_{i-1} \) to find the rightmost non-neighbour of \( e_i \), we check \( O(\deg_2(e_i)) \)
vertices, where each check takes constant time using arrays $A$, $B$, and $F$. Since the weights are positive, $w(S(e_i)) = w(S(f_j)) + w(e_i) > w(S(f_j))$ if such an $f_j$ exists, and thus $f_j \prec_e e_i$. Therefore, inserting $e_i$ into $\tau_{i-1}$ to create $\tau_i$ will also take $O(\deg(e_i))$ time.

Summing over all vertices in $\pi^*$, of which there are $m = |E|$, we have $O(m + \sum e_i \deg(e_i))$ time to create $\tau_m$. It remains to bound $\sum e_i \deg(e_i)$.

**Proposition 6.12.** $\sum e_i \deg(e_i) \leq O(mn)$.

**Proof.** By Proposition 6.11, we have:

$$\sum e_i \deg(e_i) \leq \sum_{e_i = (a,b)} \left[ \sum_{v:a \in E} \deg(v) + \sum_{v:b \in E} \deg(v) \right]$$

For a given vertex $v$, $\deg(v)$ is used $\deg(v)$ times, one for every edge incident to $v$, thus

$$\sum e_i \deg(e_i) = \sum_{e_i = (a,b)} \left[ \sum_{v:a \in E} \deg(v) + \sum_{v:b \in E} \deg(v) \right]$$

$$\leq \deg(v_1) \cdot \deg(v_1) + \ldots + \deg(v_n) \cdot \deg(v_n)$$

$$\leq \deg(v_1) \cdot \Delta_G + \ldots + \deg(v_n) \cdot \Delta_G \leq 2m \cdot \Delta_G \leq O(mn)$$

Therefore, the total running time is $O(m + mn) = O(mn)$. The correctness and robustness of the algorithm follows from Theorem 6.7 as well as the correctness and robustness of Algorithm 6.1, which we give in [48]. By not constructing $L^2(G)$ explicitly, we also manage to maintain the space complexity to the size of $G$ only, i.e., $O(m + n)$ instead of the $O(mn)$ required to construct $L^2(G)$. We conclude with the following theorem:

**Theorem 6.13.** Let $G = (V, E, w)$ be an edge weighted cocomparability graph, where $w : E \to \mathbb{R}_{>0}$. A maximum weight induced matching on $G$ can be computed in $O(mn)$ time and $O(m + n)$ space.

### 6.3 Weighted Graph Searches

The algorithms above exploit the vertex ordering characterization of cocomparability graphs, but do not rely on graph searching to do so. And so, as previously noted, it is natural to explore whether there are “good” variants of graph searches for weighted graphs - whether the weights are on the vertices of the edges. Let $G(V, E, w)$ be an edge weighted graph with
Consider $\text{LexBFS}$ for instance, this graph search has two tie breaking rules; first by the BFS rule then by the lexicographic label rule. Suppose now that we add a third rule; the heaviest edge rule.

An intuitive first attempt to construct such an ordering – that maintains the BFS nature of the search – leads to two possible variants: 1. Breaking ties by lexicographic labels first then by weights. Or, 2. Breaking ties by weights then by lexicographic labels. It is easy to see however that the second variant can easily lose all the structure guaranteed by lexicographic labels.

Consider the algorithm $\omega.\text{LexBFS}$, which is a variant of Algorithm 2.1, $\text{LexBFS}$. The algorithm appends the edge weights to the label so the lexicographic pull has precedence over the weights.

**Algorithm 6.3 $\omega.\text{LexBFS}$**

**Input:** $G(V,E,w)$, where $w : E \rightarrow \mathbb{R}_{>0}$

**Output:** An ordering $\sigma = v_1, \ldots, v_n$ of $V$

1. Assign the label $\epsilon$ to all vertices
2. label$(s) \leftarrow \{n + 1\}$
3. for $i \leftarrow 1$ to $n$ do
4. Pick an unnumbered vertex $v$ with lexicographically largest label
5. $\sigma(i) \leftarrow v$ \quad ▷ This assigns to $v$ the number $i$
6. for each unnumbered vertex $u$ adjacent to $v$ do
7. Append $(n - i) \cdot w(vu)$ to label$(u)$
8. end for
9. end for

Ideally, one should be interested in a variant of the existing $\text{LexBFS}$ algorithm that, if reduced to the 0,1 case – where 1 means the existence of an edge, 0 a non-edge – produces a valid $\text{LexBFS}$ ordering. Algorithm $\omega.\text{LexBFS}$ does not achieve that, as illustrated in Fig. 6.9.

![Diagram](image)

<table>
<thead>
<tr>
<th>vertex</th>
<th>i=1</th>
<th>i=2</th>
<th>i=3</th>
<th>i=4</th>
<th>i=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$\epsilon$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>$\epsilon$</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>$\epsilon$</td>
<td>41</td>
<td>4131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>33</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>32</td>
<td>3224</td>
<td>3224</td>
</tr>
</tbody>
</table>

Figure 6.9: $\omega.\text{LexBFS}(G) = a, b, c, d, e$ is not a $\text{LexBFS}$ ordering of unweighted $G$.

However $\omega.\text{LexBFS}$ satisfies a “weighted 4 point condition” as well, where the condition is the following:
Property 6.14. Given a $\omega$.LexBFS ordering $\sigma$, for any triple $a \prec_{\sigma} b \prec_{\sigma} c$, if $w(ab) < w(ac)$ then there exists a vertex $d$ such that $d \prec_{\sigma} a$ such that $w(db) \geq w(ac)$ and $w(dc) < w(db)$.

Notice that $w(dc) < w(db)$ ensures that when vertex $d$ is visited, then vertices $b$ and $c$ are separated and they will not be placed together again in the same partition class.

Similarity First Search. Very recently (2017), Laurent and Seminaroti introduced a new graph search on edge weighted graphs: Similarity First Search – SFS for short [54]. SFS is a partition refinement based algorithm that mimics LexBFS on edge weighted graphs. It was introduced and used as a multisweep algorithm to recognize Robinson Similarity matrices.

Let $S^n$ denote the set of symmetric $n \times n$ matrices. A matrix $M \in S^n$ is a Robinson similarity matrix if its entries are monotone decreasing in all four directions from the diagonal. That is, for:

$$1 \leq i < j < k \leq n, \text{ we have } M_{ik} \leq \min\{M_{ij}, M_{jk}\} \quad (6.14)$$

The diagonals do not matter in this definition. Robinson matrices rise in similarity-like problems [54]. A Robinson matrix is either a similarity or a dissimilarity matrix depending on whether its entries are monotonically decreasing or increasing (respectively) when moving away from the diagonal. Without loss of generality, one can only consider non-negative Robinson similarity matrices [54], so let $M \in S^n$ be such a matrix and let $\sigma$ be a permutation of $[n]$. Let $M_{\sigma}$ be the matrix resulting from permuting both the rows and the columns of $M$ according to the ordering of $\sigma$.

Definition 6.15. For $M \in S^n$, if there exists a permutation $\sigma$ such that $M_{\sigma}$ is a Robinson matrix, then we call $\sigma$ a Robinson ordering of $M$.

In an old result of 1969 [66], Roberts noticed that a binary matrix $M$ is a Robinson similarity matrix if and only if its corresponding $G_M$ is a proper interval graph; that is $M$ is the adjacency matrix of $G_M$, alternatively $G_M$ is the graph $G(V,E)$ constructed from $M$ where $V = [n]$ and $E$ is the set of nonzero entries of $M$. It is easy to see this is the case indeed just by reducing the inequality of (6.14) to the 0,1 case, we get a PI-ordering. Using this result, and the fact that proper interval graphs can be recognized using 3 LexBFS+ sweeps (Theorem 5.20) [11], Laurent and Seminaroti introduced SFS as an extension of LexBFS to work on all Robinson matrices and not just the binary ones; in other words to work on edge weighted graphs. With this in mind, we now present their SFS algorithm.

The algorithm takes as input an $n \times n$ symmetric matrix and outputs a linear ordering of $[n]$. In particular, let $M \in S^n$, and consider the following sequence: $\sigma_1 = SFS(M), \sigma_2 = SFS(M_{\sigma_1}), \ldots, \sigma_n = SFS(M_{\sigma_{n-1}})$; then:
**Theorem 6.16.** [54] A symmetric $n \times n$ matrix $M$ is Robinson if and only if $O(n)$ SFS sweeps produce a Robinson ordering of $[n]$.

In other words, $\sigma_n = SFS(M_{\sigma_{n-1}})$ is a Robinson ordering.

**Algorithm 6.4 SFS($M$) [54]**

**Input:** A non-negative square matrix $M \in S^n$

**Output:** A linear ordering $\sigma$ of $n$

1. $\mathcal{P} \leftarrow (V)$ \hspace{5mm} $\triangleright$ Queue of unvisited vertices
2. for $i \leftarrow 1$ to $n$ do
3. \hspace{10mm} $S \leftarrow$ The first partition of $\mathcal{P}$
4. \hspace{10mm} Pick an arbitrary pivot $p$ in $S$
5. \hspace{10mm} $\sigma(p) \leftarrow i$ \hspace{10mm} $\triangleright$ $p$ appears in position $i$ of $\sigma$
6. \hspace{10mm} Remove $p$ from $\mathcal{P}$
7. \hspace{10mm} $N(p) \leftarrow$ The set of vertices $y \in \mathcal{P}$ where $M[p, y] > 0$
8. \hspace{10mm} $\psi_p \leftarrow$ The similarity partition of $N(p)$ with respect to $p$
9. \hspace{10mm} $\mathcal{P} \leftarrow \text{MRefine}(\mathcal{P}, \psi_p)$
10. end for
11. return $\sigma$

SFS is a partition refinement based algorithm. As explained at the end of Chapter 2, one way to implement LexBFS is via partition refinement; Step 9 of SFS is that step of the algorithm that first refines by neighbourhoods then by weights. As presented in [54]: let $\mathcal{P} = \{A_1, A_2, \ldots, A_k\}$ be the current partition at a given iteration. Formally, when $p$ is a pivot, $N(p) = \{y \in \mathcal{P} : M_{py} > 0\}$ and $\psi_p$ is a partition of $N(p)$ by decreasing order of weights. That is, if $N(p) = y_1 \geq y_2 \geq \ldots \geq y_t > 0$ then $\psi_p = \{B_1, B_2, \ldots, B_s\}$, where $B_1 = \max(N(p)), B_2 = \max(N(p) \setminus B_1), \ldots, B_s = \min(N(p))$. Algorithm MRefine takes $(\mathcal{P}, \psi(p))$ and refines every partition class of $\mathcal{P}$ in-situ where every $A_i \in \mathcal{P}$ produces $\{A_i \cap B_1, A_i \cap B_2, \ldots, A_i \cap B_s, A_i \setminus \{B_1 \cup B_2 \cup \ldots B_s\}\}$.

It turned out that the SFS algorithm is the corresponding partition refinement of Algorithm $\omega$.LexBFS. Although it was claimed in [54] that SFS on 0,1 matrices produces a LexBFS ordering of the corresponding graph, as shown in Fig. 6.9, this is not the case. However, despite this discrepancy, Laurent and Seminaroti successfully used SFS to recognize Robinson matrices.

Notice as a corollary of Theorem 6.16, to compute a maximum (or minimum) weight Hamilton path on weighted proper interval graphs, it suffices to scan a good Robinson ordering from left to right and always take the heaviest (lightest) weight edge.
Chapter 7

AT-free Graphs

Overview. Golumbic, Monma and Trotter showed that cocomparability graphs are strictly contained in AT-free graphs [39]. It is thus a natural question to ask whether algorithms on cocomparability graphs can be lifted to AT-free graphs. The lifting framework from interval to cocomparability graphs suggests that this latter class has some sort of underlying interval-like structure. AT-free graphs have been studied from different perspectives, many of which suggest this graph class also has an underlying linear structure; we list them below along with some properties of this graph class.

7.1 Structural Properties

The question of whether AT-free graphs have similar characterizing orderings to the ones we have seen so far, has been a long standing open problem and was answered affirmatively in 2015 by Corneil and Stacho [20]; but the structure on these orderings has not yet been exploited enough in order to develop efficient algorithms. This raises the question of whether this is the right type of linearity we need to study. We illustrate below a few different ways to capture the linear structure of AT-free graphs.

1. Dominating Pairs.

AT-free graphs were first studied as a separate graph class by Corneil, Olariu and Stewart in [17], where they gave the first evidence for linear structure on this graph family. In particular, they showed that every connected AT-free graph contains a dominating pair of vertices. Recall that this is a pair $x, y$ of vertices in $G$ such that every $xy$ path $P$ dominates $G$ – that is, every $v \in V$ is either on $P$ or hits $P$. Later, in [18], the same authors gave a linear time algorithm to compute such a pair using LexBFS. Their algorithm, presented
below, is very simple and always returns a dominating pair if \( G \) is AT-free. Notice that \( \tau \) is not a \( + \) sweep, just a LexBFS ordering that uses a LexBFS end vertex as its starting point.

**Algorithm 7.1 Dominating Pairs [18]**

**Input:** A connected AT-free graph \( G(V, E) \)

**Output:** A dominating pair \( \{x, y\} \) of \( G \)

1. \( \sigma \leftarrow \text{LexBFS}(G) \)
2. \( x \leftarrow \text{The end vertex of } \sigma \)
3. \( \tau \leftarrow \text{LexBFS}(G, x) \)
4. \( y \leftarrow \text{The end vertex of } \tau \)
5. return \( \{x, y\} \)

**Theorem 7.1.** [18] A connected AT-free graph always has a dominating pair \( \{x, y\} \) which can be computed in linear time.

2. Triangulation.

Linearity in AT-free graphs can also be seen through a result of Möhring in [62], where the author showed that every minimal triangulation of an AT-free graph is an interval graph. Given a graph \( G(V, E) \), a minimal triangulation of \( G \) is an inclusion minimal set of edges \( E' \) such that \( G'(V, E \cup E') \) is a chordal graph. Möhring showed that this new graph \( G' \) is an interval graph if \( G \) is AT-free.

In [17], Corneil et al. characterized AT-free graphs in terms of minimal triangulations by proving:

**Theorem 7.2.** A graph \( G(V, E) \) is an AT-free graph if and only if every minimal triangulation of \( G \) is an interval graph.

These two results already hint to a linear structure in AT-free graphs; especially this later one which exhibits interval-like structure. Another way linearity on AT-free graphs has been captured is via a vertex ordering, that is analogous to the perfect elimination ordering on chordal graphs (albeit not as strong).

3. Admissible Elimination Orderings.

**Definition 7.3.** Given three independent vertices, \( u, v, x \), we say that the pair \( u, v \) is unrelated with respect to \( x \), if there exists two paths, a \( ux \) path \( P_{ux} \) and a \( vx \) path \( P_{vx} \), such that \( v \) misses \( P_{ux} \) and \( u \) misses \( P_{vx} \).

If \( u \) and \( v \) are unrelated with respect to a vertex \( x \), we say that \( x \) belongs to the interval \( [u, v] \), and write \( x \in I[u, v] \). In other words, \( x \) is “between” \( u \) and \( v \).
Definition 7.4. A vertex $x$ is admissible if no two vertices are unrelated with respect to $x$.

Definition 7.5. An ordering $\sigma = v_1, \ldots, v_n$ is an admissible elimination ordering, AEO for short, if for all $i$, $v_i$ is admissible in $G[v_1, \ldots, v_i]$.

We illustrate these definitions in Fig. 7.1 below: On $G_1$ in particular, $x$ is non-admissible since $x \in I[u, v]$ and $\sigma = x, w, z, u, v$ is an AEO ordering of $G_1$

![Image](image.png)

Figure 7.1: From left to right: Illustration of $u, v$ unrelated vertices with respect to $x$. $G_1$ has the interval $x \in I[u, v]$; thus $u, v$ are unrelated with respect to $x$. $G_2$ contains the asteroidal triple $a, d, f$.

Corneil et al. proved the following:

Theorem 7.6. [17] If $G$ is AT-free then $\sigma = \text{LexBFS}(G)$ is an admissible elimination ordering.

Corollary 7.7. [17] The end vertex of a LexBFS ordering on an AT-free graph is admissible.

Therefore, for the above dominating pair algorithm, Step 3 required an admissible vertex to start the second LexBFS, Step 1 is one easy way to find such a vertex. The converse of Theorem 7.6 is not true however. That is, there are graphs with asteroidal triples that admit AEO orderings, as shown by $G_2$ in Fig. 7.1 above. This graph has an asteroidal triple $(a, d, f)$ and the ordering $\sigma = g, a, d, f, b, c, e$ is an AEO of $G$ and also a LexBFS ordering. This is why AEOs on AT-free graphs are not as strong as PEOs on chordal graphs. On the other hand, Corneil and Köhler showed that:

Theorem 7.8. [15] $G(V, E)$ is an AT-free graph if and only every LexBFS of $G$ is an admissible elimination ordering.
Although a constant number of LexBFS sweeps is known to not be enough to determine if a graph is an AT-free graph [20], the elimination ordering of admissible vertices proved useful in showing the existence of a vertex ordering characterization for this graph class. We will come back to this when introducing this characterization. For now, we present another way to capture structure on AT-free graphs; this time, via cocomparability orderings.

4. Strong 2-Cocomparability Orderings.
A second ordering that fails to characterize AT-free graphs but illustrates how AT-free graphs extend cocomparability graphs is the strong 2-cocomparability ordering. In [9], Chang et al. introduced a family of graphs that extends cocomparability orderings in the following way:

**Definition 7.9.** A vertex ordering $\sigma$ is a $t$-cocomparability ordering if for every triple $a \prec_\sigma b \prec_\sigma c$ such that $\text{dist}(a, c) \leq t$ then $\text{dist}(a, b) \leq t$, for $t > 1$.

A strong $t$-cocomparability ordering is a $t$-cocomparability ordering that satisfies the extra condition that if $\text{dist}(a, c) \leq t$ then $\text{dist}(a, b) \leq t$ or $bc \in E$. A graph $G(V, E)$ is a (strong) $t$-cocomparability graph if $G$ admits a (strong) $t$-cocomparability ordering.

Chang et al. showed that a graph $G$ is a $t$-cocomparability graph if and only if for every power $k \geq t$, $G^k$ is a cocomparability graph. In particular, they were able to show that every AT-free graph is a strong 2-cocomparability graph. Consequently, the square of every AT-free graph is a cocomparability graph. Unfortunately, the strong 2-cocomparability orderings are not a characterization of AT-free graphs either. In particular, the same $G_2$ in Fig. 7.1 admits a strong 2-cocomparability ordering $\sigma = a, b, c, d, e, f, g$.

Lastly, we have the vertex ordering characterization of AT-free graphs.

5. AT-F Orderings.

It was not until 2015 that a vertex ordering characterization of AT-free graphs was found by Corneil and Stacho who proved the following theorem:

**Theorem 7.10.** [20] A graph $G(V, E)$ is an AT-free graph if and only if there exists an ordering $\sigma$ of $V$ such that for all $x \in I[u, v]$, $x \prec_\sigma u$ or $x \prec_\sigma v$.

We call such orderings AT-free orderings. The theorem says that both endpoints of the interval cannot be placed behind their corresponding non-admissible vertices. There are, however, no restrictions on the placement of admissible vertices. It is worth noting that a cocomparability ordering is precisely an AT-free ordering where for every interval $I[u, v]$ containing $x$, the endpoints $u, v$ are on different sides of $x$. 

When Corneil and Stacho gave the above characterization of AT-free graphs, they also gave a constructive algorithm (in $O(mn)$ time) to compute such orderings. Their algorithm is not based on any “classical” graph search, and they gave examples of graphs where no LexBFS and no LexDFS ordering is an AT-free ordering. This is somewhat disappointing since all the graph classes we covered so far have at least one LexBFS based recognition algorithm. They however raised the question of whether there is a BFS or DFS ordering that is also an AT-free ordering. This year (2018), Beisegel gave a variant of BFS that computes an AT-free ordering by looking at all the intervals $I[u, v]$ of the graph, and selecting the right admissible and/or endpoint vertices to visit at every step [2]. Beisegel’s algorithm also takes $O(mn)$ time.

The question remains however: If we hope to lift algorithms from cocomparability graphs to AT-free graphs, do multiple sweeps help expose any structure? The dominating pair algorithm for instance uses two traversals. When exploring this idea, we did not come up with any nice algorithms, but found a few structural properties that we hope can be useful for future work on this topic. We present them below as well as illustrate some approaches that did not work.

From the LexBFS AEO property on AT-free graphs (Theorem 7.6), it is clear that the last vertex of any LexBFS ordering of an AT-free graph is admissible. One can show that for every end vertex $x$ of a LexBFS ordering on $G$ AT-free, there exists an AT-free ordering of $G$ where $x$ is also an end vertex. We say that a triple $u, v, x$ is bad with respect to an AT-free ordering $\sigma$ if $x \in I[u, v]$ and both $u, v$ are place before $x$ in $\sigma$, i.e., the triple $u, v, x$ violates the condition of Theorem 7.10 above.

**Lemma 7.11.** Let $\sigma$ be a LexBFS ordering of an AT-free graph $G$ ending at $x$. There exists an AT-free order $\tau$ of $G$ where $x$ is the end vertex of $\tau$.

**Proof.** Suppose not, and consider $\tau = u_1, \ldots, x, y, \ldots, u_n$ an AT-free order of $G$. Let $y$ be the leftmost vertex to the right of $x$ in $\tau$. I.e., vertex $y$ appears after vertex $x$ in $\tau$.

It is easy to see that if $xy \in E$, we can swap $x$ and $y$ in $\tau$ and create a new AT-free order $\tau'$ of $G$. Suppose $xy \notin E$. Suppose further that swapping $x$ and $y$ would create a bad triple. This implies in particular that $x \in [z, y]$ for some $z \prec x \prec y$ and swapping $x$ and $y$ would violate the condition of Theorem 7.10. This also implies that $x$ is not admissible. This contradicts Corollary 7.7.

Therefore it suffices to repeatedly swap $x$ in new orderings $\tau'$ until it is at the end of an AT-free ordering of $G$. □

As one can see through a number of results in this thesis, the Flipping Lemma (Lemma 2.15) is one of the most useful tools for multisweep algorithms on cocomparability graphs. In an attempt to extend this property to arbitrary graphs, we get the following:
Lemma 7.12 (The Private Neighbour Path). Let $\sigma, \pi$ be two LexBFS orderings of $G$ where $\pi = \text{LexBFS}^+(\sigma)$. For any two vertices $a, b$ such that $a \prec \sigma b$ and $a \prec \pi b$, there exists an induced path $Q$ starting at $a$, of length at least one, where all the vertices (except possibly $a$) in $Q$ are non adjacent to $b$.

For the purpose of the proof, recall that $\text{label}(v)$ is the label of vertex $v$ as assigned by Algorithm LexBFS (Algorithm 2.1), when $v$ was visited.

Proof. Consider the ordering of $a$ and $b$ in $\pi$, at the time $a$ was chosen by LexBFS, either $a$ and $b$ had equal labels or $\text{label}(a) \geq_{\text{lexico}} \text{label}(b)$. If $a$ and $b$ had equal lexicographic labels then the $+$ rule would have broken ties by choosing $b$ first since $a \prec \sigma b$. Therefore $\text{label}(a) \geq_{\text{lexico}} \text{label}(b)$; that is there exists a private neighbour of $a$ with respect to $b$, call it $a_1$, in $\pi$ such that $a_1 \prec \pi a \prec \pi b$ and $a_1 a \in E, a_1 b \notin E$. Choose $a_1$ to be the leftmost such private neighbour of $a$ in $\pi$.

If $a_1 \prec \sigma b$ then $a_1 \prec \sigma b$ and $a_1 \prec \pi b$, and by the argument above, there exists $a_2$, a private neighbour of $a_1$ with respect to $b$ in $\pi$ such that $a_2 \prec \pi a_1 \prec \pi a \prec \pi b$. By the choice of $a_1$ as the leftmost private neighbour of $a$ in $\pi$ with respect to $b$, it follows that $a_2 a \notin E$. Again, choose $a_2$ to be the leftmost such private neighbour of $a_1$ in $\pi$.

By repeating this argument, and by the finiteness of the graph, we eventually get to a vertex $a_j$ that has a private neighbour $a_{j+1}$ with respect to $b$ in $\pi$, and flips ordering with $b$ from $\sigma$ to $\pi$, that is:

$$a_{j+1} \prec \pi a_j \prec \pi \ldots \prec \pi a_1 \prec \pi a \prec \pi b$$

$$a \prec \sigma a_1 \prec \sigma \ldots \prec \sigma a_j \prec \sigma b \prec \sigma a_{j+1}$$

Thus the induced path constructed by concatenating the edges $a_{j+1}a_j \cdot a_ja_{j-1} \cdot \ldots \cdot a_2a_1 \cdot a_1a$ avoids $b$ with the exception of possibly $ab$ in $E$. \qed

A well known property on cocomparability graphs says that the end vertex of an arbitrary LexBFS ordering on $G$ cocomparability is a sink in a poset [50]; as a corollary we get the following:

Corollary 7.13. Consider a sequence of LexBFS$^+$ orderings on a cocomparability graph:

$$\sigma_1 = \text{LexBFS}(G, a) = a \ldots x \quad \sigma_2 = \text{LexBFS}^+(\sigma_1) = x \ldots y \quad \sigma_3 = \text{LexBFS}^+(\sigma_2) = y \ldots z$$

Then $\text{dist}(x, z) \leq 1$.

Proof. If this is not the case, then $xz \notin E$, which means $x$ and $z$ are comparable in a poset. This contradicts both being sinks, with the possible special case that $xz$ is a maximal chain
in the poset. That is, without loss of generality $x$ is a sink, and $z$ is a source. Since $y$ is also an end vertex of a LexBFS, $y$ is also a sink (or a source). If $y$ is a sink then $x\parallel y$ and $xy \in E$, which using $\sigma_2$ implies that $x$ is universal in $G$ and thus adjacent to $z$. If $y$ is a source then $y\parallel z$ and $yz \in E$, which using $\sigma_3$ implies $y$ is universal in $G$, and in particular $yx \in E$. Using this in $\sigma_2$ and the fact that $z <_{\sigma_2} y$ implies that $x$ is also universal in $G$, and thus adjacent to $z$.

Therefore $\text{dist}(x,z) \leq 1$.

In AT-free graphs, we get the following:

Lemma 7.14. Consider a sequence of LexBFS$^+$ orderings on an AT-free graph:

$$
\sigma_1 = \text{LexBFS}(G,a) = a \ldots x \quad \sigma_2 = \text{LexBFS}^+(\sigma_1) = x \ldots y \quad \sigma_3 = \text{LexBFS}^+(\sigma_2) = y \ldots z
$$

Then $\text{dist}(x,z) \leq 2$.

Recall that $P_{ab}$ denotes an induced $ab$ path in the graph. If $c$ is a vertex on $P_{ab}$, then we write $P_{ac}$ to refer to the induced subpath from $a$ to $c$ in $P_{ab}$. If $c$ is $b$’s neighbour on $P_{ab}$, then we can write $P_{ab} = P_{ac} \cdot cb$ to refer to the concatenation of $P_{ac}$ with the edge $cb$. We also write $|P_{ab}|$ to denote the length of this path.

Proof. Suppose for sake of contradiction that $z \neq x$ and $xz \notin E$. By Theorem 7.1, both $(x,y)$ and $(y,z)$ are dominating pairs in $G$. Let $P_{yx}$ be a shortest induced $yx$ path. Then $P_{yx}$ is dominating, and $z$ is either on $P_{yx}$ or has a neighbour on $P_{yx}$. If the former, then since $P_{yx}$ is induced, the subpath $P_{yz}$ is induced and thus $x$ misses $P_{yz}$ unless $xz \in E$, which is a contradiction to our assumption. But then $(y,z)$ is a dominating pair, which means $P_{yz}$ must be dominating. Thus $z$ has a neighbour on $P_{yx}$, let $w$ be such a neighbour, and choose $w$ as the leftmost vertex on $P_{yx} = y \ldots w \ldots x$ that is adjacent to $z$. Thus for all $v \in P_{yw}, vz \notin E$.

If $w = y$, then $yz \in E$. This implies that $G$ is a clique since $y$ is a LexBFS end vertex and $\sigma_3$ starts at $y$ and ends at $z$, which in turn implies $zx \in E$. So suppose $w \neq y$, and suppose further that $|P_{wx}| > 1$, then $P_{yw} \cdot wz$ is a dominating path that misses $x$. Thus $|P_{wx}| = 1$, i.e., $wx \in E$. Therefore

$$
\text{dist}(x,z) = 2 = |xw \cdot wz| 
$$

(7.1)

To prove the next theorem, we need the simple key observation:
Fact 7.15. If $G$ is AT-free and $x \in I[y, z]$, then $y \notin I[x, z]$ and $z \notin I[x, y]$ otherwise $x, y, z$ is an asteroidal triple.

Claim 7.16. Let $x$ be an admissible vertex and $y$ a non-admissible neighbour of $x$, where $y \in I[a, b]$ for some pair $a, b \in V$. At least one of $a, b$ is a neighbour of $x$.

Proof. Suppose not. Let $P_{ya}, P_{yb}$ be two induced paths that witness $y \in I[a, b]$, then $b$ misses $P_{ya}$ and $a$ misses $P_{yb}$. If $x$ misses both $P_{ya}$ and $P_{yb}$, then the two paths $Q_{xa} = xy \cdot P_{ya}$ and $Q_{xb} = xy \cdot P_{yb}$ witness $x \in I[a, b]$ – a contradiction to $x$ being admissible. Therefore, either $x$ is adjacent to one of $a, b$ – in which case we are done – or $x$ lies in one of $P_{ya}, P_{yb}$.

Without loss of generality, suppose $x$ lies in $P_{ya}$. Suppose further that $xa$ is not an edge of $P_{ya}$ (otherwise we’re done). We can write $P_{ya}$ as $P_{ya} = P_{xa} = xy \cdot P_{xa}$, since $xy \in E$ and $P_{ya}$ is induced. Now consider the two paths $P_{xa}$ and $xy \cdot P_{yb}$. Since $xa \notin E$, $a$ misses $P_{yb}$ and $b$ misses $P_{ya}$, it follows that $a$ misses $xy \cdot P_{yb}$ and $b$ misses $P_{xa}$. Thus $x \in I[a, b]$ unless $|P_{xa}| = 1$, i.e., $xa \in E$. Therefore it must be that at least one of $a, b$ is a neighbour of $x$. □

Theorem 7.17. Let $G$ be an AT-free graph. Every admissible vertex in $G$ has at least one admissible neighbour.

Proof. Suppose not, and let $x$ be an admissible vertex such that $\forall y \in N(x), \exists a, b \in V$ where $y \in I[a, b]$. The idea of the proof is to show that for every non-admissible neighbour of $x$, we must introduce a series of new vertices, one of which – at least – must be a new neighbour of $x$. By finiteness of $N(x)$, this cannot keep going. By finiteness of $G$, we cannot keep introducing new vertices to the graph, thus we must reuse some previously introduced vertices. In doing so, we either introduce a new neighbour of $x$, or show that a non-neighbour of $x$ must be a neighbour.

Let $y \in N(x)$ where $y \in I[a, b]$ for some pair $a, b$. By Claim 7.16, at least one of $a, b$ must be a neighbour of $x$.

We first show that if both $a, b \in N(x)$, then each of $a, b$ will introduce a new vertex: Vertex $d$ coming from $a$, and vertex $q$ coming from $b$.

Case (i) Suppose both $a, b \in N(x)$.

By assumption, both $a, b$ are non-admissible. Suppose $a \in I[c, d], b \in I[p, q]$. Notice that if all four vertices $c, d, p, q$ are new (i.e., not one of $\{y, b\}$), then by Claim 7.16, at least one of $c, d$ is a neighbour of $x$, and at least one of $p, q$ is a neighbour of $x$. This introduces new vertices to $N(x)$ for which we repeat the same argument.

Therefore, suppose some of these vertices are not new. By Fact 7.15, we know that it cannot be that both $c \in \{y, b\}$ and $d \in \{y, b\}$, since $y \in I[a, b]$ and $a \in I[c, d]$. In particular, without loss of generality, for $a \in I[c, d]$, suppose $c \in \{y, b\}$. Similarly, without loss of generality, for $b \in I[p, q]$, suppose $p \in \{y, a\}$.
This gives four possible cases \(a \in I[c, d]\) where \(c \in \{y, b\}\) and \(b \in I[p, q]\) where \(p \in \{y, a\}\). We present the cases in detail.

Recall that \(y \in I[a, b]\) and let \(P_{ya}, P_{yb}\) be the corresponding paths of this interval.

1. \(a \in I[y, d]\) and \(b \in I[y, q]\).

Let \(Q_{ay}, Q_{ad}\) be the paths that witness \(a \in I[y, d]\).

Let \(Q_{by}, Q_{bq}\) be the paths that witness \(b \in I[y, q]\).

Notice that \(d\) misses \(Q_{bq}\) otherwise \(a, y, b\) forms an asteroidal triple. Notice also that \(q\) misses \(Q_{ad}\) otherwise \(a, y, b\) forms an asteroidal triple.

Therefore using the paths \(x a \cdot Q_{ad}\) and \(x b \cdot Q_{bq}\), we have \(x \in I[d, q]\).

2. \(a \in I[y, d]\) and \(b \in I[a, q]\).

Let \(Q_{ay}, Q_{ad}\) be the paths that witness \(a \in I[y, d]\).

Let \(Q_{ba}, Q_{bq}\) be the paths that witness \(b \in I[a, q]\).

Vertex \(a\) misses \(Q_{bq}\) since \(b \in I[a, q]\). This implies \(d\) misses \(Q_{bq}\), for otherwise let \(w\) be the neighbor of \(d\) on \(Q_{bq}\), and let \(Q_{wb}\) be the subpath of \(Q_{bq}\) from \(w\) to \(b\). Then, vertex \(a\) misses the path \(dw \cdot Q_{wb} \cdot P_{yb}\), which implies \(a, y, d\) forms an asteroidal triple, since \(a \in I[y, d]\).

Similarly, vertex \(y\) misses \(Q_{ad}\) sine \(a \in I[y, d]\). This implies vertex \(b\) misses \(Q_{ad}\) otherwise \(a, y, b\) is an asteroidal triple. This, in turn, implies vertex \(q\) misses \(Q_{ad}\), otherwise let \(z\) be the neighbour of \(q\) on \(Q_{ad}\), and let \(Q_{za}\) be the subpath of \(Q_{ad}\) from \(z\) to \(a\). Then vertex \(b\) misses the path \(qz \cdot Q_{za}\), which implies \(a, q, b\) forms an asteroidal triple, since \(b \in I[a, q]\).

Therefore, using the paths \(xa \cdot Q_{ad}\) and \(xb \cdot Q_{bq}\), we have \(x \in I[d, q]\).

3. \(a \in I[b, d]\) and \(b \in I[y, q]\).

This case is symmetric to case (2) above, with \(a\) and \(b\) switched.

4. \(a \in I[b, d]\) and \(b \in I[a, q]\).

Let \(Q_{ab}, Q_{ad}\) be the paths that witness \(a \in I[b, d]\).

Let \(R_{ba}, R_{bq}\) be the paths that witness \(b \in I[a, q]\).

Vertex \(d\) misses \(R_{bq}\), for otherwise let \(w\) be the neighbour of \(d\) on \(R_{bq}\), and let \(R_{wb}\) be the subpath on \(R_{bq}\) from \(w\) to \(b\). Then vertex \(a\) misses \(dw \cdot R_{wb}\), which implies \(a, b, d\) is an asteroidal triple, since \(a \in I[b, d]\).
Similarly, vertex $q$ misses $Q_{ad}$, for otherwise, let $z$ be the neighbour of $q$ on $Q_{ad}$, and let $Q_{za}$ be the subpath of $Q_{ad}$ from $z$ to $a$. Then vertex $b$ misses $qz \cdot Q_{za}$, which implies $a, b, q$ is an asteroidal triple, since $b \in I[a, q]$.

Therefore, using the paths $xa \cdot Q_{ad}$ and $xb \cdot R_{bq}$, we have $x \in I[d, q]$.

Thus, in all four cases, we end up with $x \in I[d, q]$, both of which are two new vertices, and one of which at least must be adjacent to $x$. We summarize this case as follows: If $y \in I[a, b]$, $y \in N(x)$, and both endpoints of the interval are neighbours of $x$ (i.e., $a, b \in N(x)$), then we can always find a pair $d, q$ as constructed above where at least one of $d, q \in N(x)$. This new neighbour of $x$ is also non-admissible by assumption, and thus lies in an interval. Repeating this argument will eventually contradict the finiteness of $G$ and $N(x)$ in particular.

Having handled Case (i) where both $a, b \in N(x)$, we turn to:

**Case (ii)** Suppose exactly one of $a, b \in N(x)$.

For the remainder of the proof, we assume that exactly one of $a, b \in N(x)$. In particular, for any triple $a, y, b$ where $y \in I[a, b]$ and $y \in N(x)$, it cannot be that all three $a, y, b \in N(x)$. Without loss of generality, suppose $ax \in E$. We will show that in this case also, we always introduce new neighbours of $x$.

Since $ax \in E$, by assumption, $a$ is non-admissible, and thus there exists two vertices $c, d$ such that $a \in I[c, d]$. If both vertices are new vertices, then by Claim 7.16, one of $c, d$ is a neighbour of $x$, and we repeat the same argument.

Thus, suppose one of $c, d$ is not a new vertex. Without loss of generality, suppose vertex $c$ is not new. Then $c \in \{y, b\}$. We will show, whether $c = y$ or $c = b$, we always end up with $x \in I[b, d]$. Thus $x$ is adjacent to one of $b, d$. But by the Case (i), we know that $xb \notin E$, thus $d$, the new vertex, is a neighbour of $x$.

1. Suppose $c = y$. Then $y \in I[a, b]$ and $a \in I[y, d]$ and $a, y \in N(x), b \notin N(x)$.

   Let $Q_{ay}, Q_{ad}$ be paths that witness $a \in I[y, d]$. Notice that $b$ misses $Q_{ad}$ otherwise $a, y, b$ is an asteroidal triple, since $y \in I[a, b]$. Thus using the paths $xy \cdot P_{yb}$ and $xa \cdot Q_{ad}$, we have $x \in I[b, d]$ – unless one of $b, d \in N(x)$.

   Since $b \notin N(x)$, it must be that $d \in N(x)$. But then we have $a \in I[y, d]$, where all three vertices $a, y, d$ are neighbours of $x$ (Case (i)).

2. Suppose $c = b$. Then $y \in I[a, b]$ and $a \in I[b, d]$ and $a, y \in N(x), b \notin N(x)$.

   Let $Q_{ab}, Q_{ad}$ be paths that witness $a \in I[b, d]$. Notice that vertex $d$ misses $P_{yb}$, for otherwise let $w$ be the neighbour of $P_{yb}$, and let $P_{wb}$ be the subpath of $P_{yb}$ from vertex $w$ to vertex $b$. Then, vertex $a$ misses $dw \cdot P_{wb}$, which implies $a, b, d$ is an asteroidal triple, since $a \in I[b, d]$. Thus using $xy \cdot P_{yb}$ and $xa \cdot Q_{ad}$, we have $x \in I[b, d]$, and thus $d$ must be a neighbour of $x$. 

We next show that vertex $d$, this new neighbour of $x$, plays the same role as vertex $a$, and thus will in turn introduce at least one new neighbour of $x$.

We have $y \in I[a, b], a \in I[b, d]$ and $y, a, d \in N(x)$. Notice that $dy \notin E$, for otherwise, $dy \in E$ implies vertex $a$ misses the path $dy \cdot P_{yb}$. Thus, $a, b, d$ would be an asteroidal triple since $a \in I[b, d]$. Notice also that $d$ misses $P_{yb}$ for this same reason (otherwise $a, b, d$ would be an asteroidal triple). Since $y \in I[a, b]$, vertex $b$ misses $P_{ya}$. Since $a \in I[b, d]$, vertex $b$ also misses $Q_{ad}$. Thus vertex $b$ misses $P_{ya} \cdot Q_{ad}$.

Let $R_{yd} = P_{ya} \cdot Q_{ad}$.

Thus, $dy \notin E$, $d$ misses $P_{yb}$ and $b$ misses $R_{yd}$. This implies $y \in I[d, b]$. And $d \in N(x)$.

Therefore vertex $d$ plays the same role as vertex $a$, as follows: We started with

$$y \in I[a, b], a \in N(x), b \notin N(x)$$

We found a new neighbour $d$ of $x$ such that

$$y \in I[d, b], d \in N(x), b \notin N(x)$$

By repeating the same argument of vertex $a$ on vertex $d$, we would introduce a new vertex $t$ such that

$$y \in I[t, b], t \in N(x), b \notin N(x)$$

By finiteness of $N(x)$, this process cannot continue.

Therefore, at least one neighbour of $x$ must be admissible. \hfill \Box

**Corollary 7.18.** Every AT-free graph has at least 3 admissible vertices or is a clique.

**Proof.** Every AT-free graph has at least two admissible vertices (unless $|V| = 1$). To obtain such a pair of admissible vertices, it suffices to run Algorithm 7.1 on $G$ [18]. Using Theorem 7.17, each one of these two admissible vertices has an admissible neighbour. Thus every $G$ with large enough diameter has at least four admissible vertices, with the exception of AT-free graphs that have diameter at most two. \hfill \Box

This corollary is reminiscent of the classical property on chordal graphs by Dirac:

**Theorem 7.19.** [25] Every chordal graph that is not a clique has at least two non-adjacent simplicial vertices.
We finish this chapter with some failed attempts at lifting algorithms from cocomparability graphs to AT-free graphs. In particular, we illustrate this through the minimum path cover problem. The problem asks for a path cover of minimum cardinality. If the cover is of size one, then the graph has a Hamilton path. The algorithm on cocomparability graphs works as follows [12]:

- Compute $\sigma$ a cocomparability ordering of $G$ in linear time using the algorithm in [58].
- Compute $\tau = \text{LexDFS}^+(\sigma)$ using our algorithm in Chapter 3.
- Construct a path cover by always choosing the rightmost available neighbour in $\tau$. By definition, this means the first vertex in the first path constructed is the last vertex in $\tau$. This last step is known as the RMN search (rightmost neighbour search).

We illustrate the algorithm below.

![Figure 7.2: An illustration of the minimum path cover algorithm.](image)

One way to try to lift this algorithm to AT-free graphs is by looking for the “right” vertex ordering on AT-free graphs that we can preprocess with some graph search – if necessary. A good candidate would be the AT-free ordering introduced in [20], but there is a second ordering, that is arguably, more structured. In the same work [20], Corneil and Stacho proved that another ordering, called LexComp, is also an AT-free ordering if $G$ is AT-free. This ordering is in fact what they use to compute AT-free orderings in $O(mn)$ time. The LexComp ordering is constructed as follows: For every vertex $v$, compute a vector $v = (|C_1| \geq |C_2| \geq \ldots \geq |C_t|)$ where $C_1, \ldots, C_t$ are the connected components of $G \setminus N[v]$, ordered in non-increasing size. The ordering $\sigma = v_1, \ldots, v_n$ is then a LexComp ordering if for every pair $u <_\sigma v$, the vector $v$ is lexicographically larger than $u$.

**Theorem 7.20.** [20] Let $G$ be an AT-free graph. Every LexComp ordering is an AT-free ordering.
The converse is not true, not every AT-free ordering is a LexComp ordering; but this gives us two orderings to work with on this graph class. Unfortunately, neither LexBFS nor LexDFS preserve the structure of either AT-free or LexComp orderings. That is, applying the + rule on top of these vertex orderings does not preserve the orderings properties, in contrast with LexBFS and LexDFS on cocomparability orderings (the $C_4$ properties 2.13 and 2.14).

Consider the graph $G_1$ in Fig. 7.3 for instance with the following interval $e, f \in [a, b]$. A valid AT-free (not Lexcomp) ordering is $f, e, d, c, b, a$ and a valid LexComp ordering is $(c, d)(e, f)(a, b)$ (where the brackets indicate a tie in the ordering for vertices with equal lexicographic vectors). It is not hard to see that applying LexDFS+ and LexBFS+ on these orderings does not maintain the AT-free and LexComp orderings. This means that the naive “preprocess the vertex ordering to extract more structure” fails on AT-free graphs – at least, with these orderings.

In fact, even on cocomparability graphs, if we use a LexComp or an AT-free ordering as input for the minimum path cover algorithm, instead of a cocomparability ordering, the algorithm fails to compute an optimal solution. Consider for instance the graphs in Fig. 7.3 below, both of which are cocomparability graphs. In brackets, we write the corresponding LexComp vector for every vertex.

For $G_1$, the ordering $\sigma = f, e, d, c, b, a$ is a valid AT-free (not cocomparability) ordering: edge $db$ forms an umbrella over $c$. Running the minimum path cover algorithm on this ordering (i.e., computing $\text{RMN}(\text{LexDFS}^+(\sigma))$) gives a path cover of $P_1 = f, e, d, b$ and $P_2 = c, a$ when the graph clearly has a Hamilton path.

Similarly, by running the minimum path cover algorithm on $G_2$ on the LexComp ordering $\sigma = (c)(g, fed)(ba)$, which is not a cocomparability ordering as witnessed by the umbrella $ca$ over $f$, we get a path cover of size 2 when the graph has a Hamilton path.

![Figure 7.3: Cocomparability graphs where the minimum path cover algorithm fails when given an AT-free ordering ($G_1$) and a LexComp ordering ($G_2$).](image-url)
Given these negative results, one could approach the problem by first trying to understand the structure of AT-free and LexComp orderings on cocomparability graphs: *Must* the minimum path cover algorithm start with a cocomparability ordering, or is it possible to modify the algorithm to work with a less structured ordering? Notice that doing so might give rise to a new bump number algorithm.
Chapter 8

Future Work

We presented new algorithms and structural results on cocomparability graphs. One of the main driving motivations is to find structure in vertex orderings that we can exploit algorithmically. This leaves a number of open problems, some of which we already have encountered, others we present below.

1. Some immediate questions that follow from this work:

1.1. Is there a linear time LexDFS algorithm on arbitrary graphs? If not, how about super-classes of cocomparability graphs? AT-free graph for instance.

1.2. Are there other problems on cocomparability graphs that fit into the lifting framework?

1.3. Is there a similar lifting framework from cocomparability graphs to AT-free graphs? Clearly, as we mentioned in Chapter 7, the naive approach does not work, and there does not seem to be a way of combining a constant number of graph traversals to exhibit the right structure on AT-free graphs. Perhaps, we need other vertex ordering characterizations of this graph class.

1.4. Can we use the ladder structure presented in Chapter 5 to prove that cocomparability graphs have LexCycle two?

1.5. Is there a linear time algorithm to compute a maximum (weighted) induced matching on cocomparability graphs?

1.6. Can the ⋆ rule presented in Chapter 6 work for graph classes defined on forbidden patterns of four or more vertices?

These are some of the immediate questions that follow from the results we have. We present below other questions; in particular, we give some directions of research that are worth exploring:
2. Bipartite Matching & Colouring cocomparability graphs. It remains open whether the “lifting framework” can be used for a colouring algorithm on cocomparability graphs. This problem is of particular interest since it reduces to computing a bipartite matching. What kind of search preprocessing is needed to lift the greedy interval colouring algorithm to cocomparability graphs? Both LexDFS and LexBFS preprocessing do not work. It is worth noting that LexBFS has been successful in colouring algorithms for other graph classes, chordal and Meyniel graphs for instance [32, 68]. Would combining different graph searches work?

3. Graph Searching in Large Networks & Modified Weighted Searches. So far LexDFS has been mostly useful on cocomparability graphs only. Can it be used for other graph classes? How about on large networks? In [21] it was used successfully to find compact clusters in large communities. However, the first question to raise is whether it can be implemented in linear time for arbitrary graphs. Can we compress large networks through some graph decomposition scheme, and use graph searching on this smaller quotient graph? On the other hand, SFS is the first successful step towards extending this search toolbox to weighted graphs. Are there other problems that can benefit from this weighted search? Can the variant of ω.LexBFS that breaks ties by weight first then by label have any applications?

4. Graph Searching & Parametrized Algorithms. All of the work mentioned so far has focused on well structured graph classes. It is natural to wonder if these techniques can also be used on arbitrary graphs, or less well-behaved graph classes. This was one of our motivations for introducing the “linear structure” graph parameter LexCycle. How close to “linear” are graph classes with larger LexCycle? Suppose we know that a graph class has bounded LexCycle, can this parameter be used to come up with simpler algorithms? Can we lift algorithms from graphs with small LexCycle to those with bounded LexCycle? Can these techniques give us good approximation algorithms?

5. Convexity in Graphs. Given the success these techniques have had on various graph classes, it is natural to ask why do these algorithms work – most of which are of a greedy nature. In an attempt to answer this question, with Dragan and Habib, we started looking at why these algorithms work, not from a correctness point of view, but from the underlying antimatroid structure [26]. For every result we have, we can find some sort of greedoid, or more precisely a convex geometry, that captures the convexity of these orderings. Convex geometries, antimatroids, and greedoids have appeared in the literature under various names and settings – we refer the reader to the monograph Greedoids by Korte, Lovász, and Schrader [49] for more on this.
In [26], we present a framework that captures the success of graph searching in what one might call “convex-like” graph classes. A natural question to ask is: Does every graph class with a characterizing ordering define a convex geometry of some sort? And if so, can graph searching produce such an ordering? For instance, it is well known that LexBFS always ends on specific vertices for different graph classes: For chordal graphs for instance, the last vertex is always simplicial. For cocomparability graphs, the last vertex is either a source or a sink in a corresponding poset.

6. **Algorithms on Posets.** One research theme we are interested in is a unifying view of algorithms on posets and cocomparability graphs. We successfully used LexDFS for a bump number algorithm, which raises the question of whether other problems can fall in the same category. For instance, the problem of computing linear extensions with minimum jump is NP-hard in general, but becomes tractable for various poset classes. A dichotomy however is still not clear. It is of interest to ask whether graph searching can help find a dichotomy between posets with tractable jump number and those without.
Bibliography


Appendices
Appendix A

Refinement Example

we consider the following example. Suppose \texttt{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 \( \pi \) imposes its ordering), and pivots are popped out of \( S \) in the order of \( x \) then \( y \) then \( z \). Fig. A.1, A.2, and A.3 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 \( c \)

\begin{figure}[h]
\centering
\begin{subfigure}{0.3\textwidth}
\includegraphics[width=\textwidth]{pivot_setup}
\caption{The initial setup for pivot \( x \)}
\end{subfigure}
\hspace{0.5cm}
\begin{subfigure}{0.3\textwidth}
\includegraphics[width=\textwidth]{pivot_reordering}
\caption{The new reordering of \( P \)}
\end{subfigure}
\hspace{0.5cm}
\begin{subfigure}{0.3\textwidth}
\includegraphics[width=\textwidth]{pivot_splitting}
\caption{Splitting \( P \) at \( c \)}
\end{subfigure}
\caption{Processing pivot \( x \)}
\end{figure}

The next pivot \( y \) will refine according to the newly created subpartitions as follows:
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 $z$.

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