This thesis is presented for the degree of Doctor of Philosophy of The University of Western Australia

Algorithms for Geometric Intersection Graphs

Andrew Michael Gozzard
Bachelor of Science (Physics)
Bachelor of Science (First Class Honours in Computer Science and Software Engineering)
25 July 2023

School of Computer Science and Software Engineering
Supervisors: Prof. Amitava Datta (Coordinating)
Prof. Gordon Royle
Dr Max Ward

Thesis Declaration

I, Andrew Gozzard, certify that:

This thesis has been substantially accomplished during enrolment in the degree.

This thesis does not contain material which has been accepted for the award of any other degree or diploma in my name at any university or other tertiary institution.

No part of this work will, in the future, be used in a submission in my name, for any other degree or diploma at any university or other tertiary institution without the prior approval of The University of Western Australia and where applicable, any partner institution responsible for the joint-award of this degree.

This thesis does not contain any material previously published or written by another person, except where due reference has been made in the text.

The work(s) are not in any way a violation or infringement of any copyright, trademark, patent, or other rights whatsoever of any person.

This thesis contains published work and/or work prepared for publication, some of which has been co-authored.

Signature:

Date: 25 July 2023

Abstract

I present an overview of several problems in the field of intersection graphs of geometric objects and algorithms for solving these problems. An introduction to geometric intersection graphs and a taxonomy of relevant classes is provided, as is a taxonomy of relevant problems. Here I present a brief overview of the main results of my work in this field.

The class of polygon-circle graphs is the class of intersection graphs of polygons inscribed in a common circle. Polygon-circle graphs have applications in bioinformatics, chemistry, and very large-scale integration of integrated circuits. The problems of polygon-circle graph recognition and representation have been of interest for some time, particularly given Koebe's announcement of a polynomial-time recognition algorithm [33] that was never successfully completed, and Pergel's later result that polygon-circle graph recognition is NP-complete [43]. I discuss methods for recognising and constructing intersection representations of polygon-circle graphs, and present a novel approach based on constructing optimal pseudocyclic deterministic finite automata that accept only valid alternating sequence [4] representations of a given polygon-circle graph. Empirical assessments find this method to be more performant than the alternative methods, and demonstrate the practicality of this approach by using it to show by exhaustive search that the 3-prism is the minimal non-polygon-circle graph, a result that leads to and is supported by the proof in the Chapter 3.

The class of interval filament graphs is the class of intersection graphs of curves constrained to be above intervals of the real line. While it was already known that polygon-circle graphs are a proper subclass of interval filament graphs [19, 29], the proof of this used an involved analysis of the thresholds at which random graphs in the limit to infinity develop certain properties, and so was not a constructive proof. Based on the experimental finding from Chapter 2, I present a constructive proof that the 3-prism is an interval filament graph but not a polygon-circle graph, thereby providing a constructive proof that polygon-circle graphs are a proper subset of interval filament graphs. This is the first such constructive proof of this property.

The class of circular-arc graphs is the class of intersection graphs of arcs of a common circle. These classes have applications in periodic and non-periodic scheduling tasks. The average path length minimisation problem asks in a weighted graph what edge from a set of candidate edges will result in the minimum possible average path length when added to a graph. I discuss solutions to the problem of average path length minimisation in circular-arc graphs, and present both methods that work in general graphs and others that take advantage of the optional requirement that the graph remain a circular-arc graph after the addition of the selected edge. These methods generalise to subclasses of circular-arc graphs including interval graphs.

The class of interval graphs is the class of intersection graphs of intervals of the real line. Interval graphs have applications in resource allocation and scheduling problems. The problem of finding a maximum internal spanning tree (that is, a spanning tree with the minimum possible number of leaves) in an interval graph has been a recent topic of investigation by Li, Feng, Jiang, and Zhu [38]. I discuss this result, and present a counterexample for which their algorithm does not work. I then discuss the relationship of this problem to the Hamiltonian path problem (finding a path that visits every vertex exactly once) and generalise a known greedy algorithm for Hamiltonian path in interval graphs [39] to derive a greedy algorithm for finding a maximum internal spanning tree for a given interval graph.

Acknowledgements

My heartfelt thanks to my family for their enduring support and my supervisors for their invaluable guidance. This work would not have been possible without them.

This research was supported by an Australian Government Research Training Program (RTP) Scholarship, an Australian Postgraduate Award Scholarship, and a UWA Safety-Net Top-Up Scholarship.

Contents

List of Figures

List of Publications

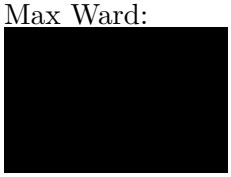
In the course of my research I have authored or co-authored a number of works that have either been published or are being prepared for publication. Below is a list of these works, including formal declarations of authorship by all relevant parties. Note that due to complications with my original project, I changed research focus and supervision in the middle of my candidature. As a result I have contributed to several publications from prior to my change in focus that do not appear in the body of this thesis, but are included in this list for the sake of completeness.

A Maximum Weight Clique Algorithm For Dense Circle Graphs With Many Shared Endpoints

This work has been published (see Ward, Gozzard, and Datta [50]). I was not the first author, but I made substantial intellectual contributions to the findings published. This work does not appear substantially in this thesis.

Authorship Declaration

Max Ward:



Andrew Gozzard:

Amitava Datta:

Abstract

Circle graphs are derived from the intersections of a set of chords. They have applications in VLSI design, bioinformatics, and chemistry. Some intractable problems on general graphs

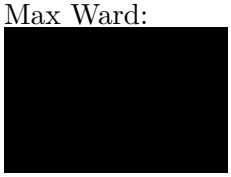
can be solved in polynomial time on circle graphs. As such, the study of circle graph algorithms has received attention. State-of-the-art algorithms for finding the maximum weight clique of a circle graph are very efficient when the graph is sparse. However, these algorithms require $\Theta(n^2)$ time when the graph is dense. We present an algorithm that is a factor of $\sqrt{n}$ faster for dense graphs in which many chord endpoints are shared. We also argue that these assumptions are practical.

Converting a network into a small-world network: Fast algorithms for minimizing average path length through link addition

This work has been published (see Gozzard, Ward, and Datta [25]). This research was originally conducted as part of my Honours degree. This work appears in Section 4.3.

Authorship Declaration

Andrew Gozzard:


Max Ward:


Amitava Datta:


Abstract

The average path length in a network is an important parameter for measuring the end-to-end delay for message delivery. The delay between an arbitrary pair of nodes is smaller if the average path length is low. It is possible to reduce the average path length of a network by adding one or more additional links between pairs of nodes. However, a naïve algorithm is often very expensive for determining which additional link can reduce the average path length in a network the most. In this paper, we present two efficient algorithms to minimize the average network path length by link addition. Our algorithms can process significantly larger networks compared to the naïve algorithm. We present simple implementations of our algorithms, as well as performance studies.

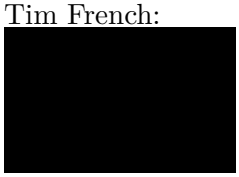
Path Zones: A Geometric Model for Sets of Transitions in Timed Automata

This work has been prepared for publication but has not been published. This work does not appear in this thesis.

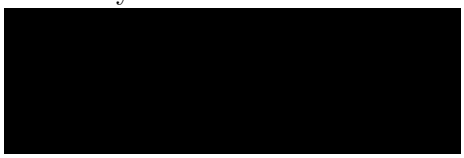
Authorship Declaration

Andrew Gozzard:

Tim French:



Mark Reynolds:



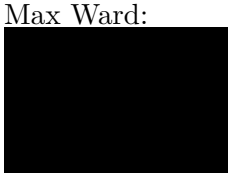
Abstract

Analysing timed automata to identify best- and worst-case execution times requires an understanding of iterations of loops through automaton locations. We present a geometric model for the set of all time-like transitions in a timed automaton compatible with a particular sequence of locations and location switches. This representation, called a *path zone*, allows us to understand the time-like properties of a path independent from its initial clock configuration. Defining composition on these path zones enables us to efficiently compute path zones corresponding to a path and analyse repetitive structures such as loops. We show that this composition can be computed in polynomial time, and give an $O(|X|^9)$ algorithm for doing so by way of example. We are interested in being able to efficiently analyse loops in timed automata to more efficiently compute their best- and worst-case execution times, and present techniques for doing so using path zones.

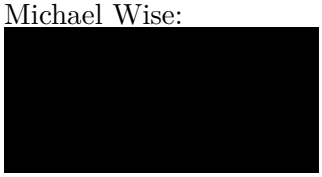
A Faster Algorithm for Maximum Induced Matchings on Circle Graphs

This work has been published (see Ward, Gozzard, Wise, and Datta [51]). I was not the first author, but I made substantial intellectual contributions to the findings published. This work does not appear substantially in this thesis.

Authorship Declaration

Max Ward:


Andrew Gozzard:


Michael Wise:


Amitava Datta:


Abstract

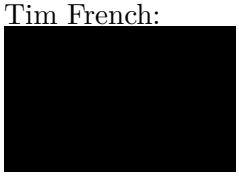
Circle graphs have applications to RNA bioinformatics, computational chemistry, and VLSI design. Additionally, many problems that are intractable on general graphs are efficient for circle graphs. This has driven research into algorithms for circle graphs. One well known graph problem is to find a maximum induced matching. This is NP-Hard, even for bipartite graphs. No algorithm for this problem that works directly on circle graphs has been proposed. However, since circle graphs are included in interval filament graphs, algorithms for this class can be applied to circle graphs. Unfortunately, this entails a large computational cost of $O(|V|^6)$ time. We propose an algorithm that operates directly on circle graphs, and requires only $O(|V|^3)$ time.

A Modal Aleatoric Calculus for Probabilistic Reasoning

This work has been published (see French, Gozzard, and Reynolds [15, 16]). This work is related to my original research focus, and does not appear in this thesis.

Authorship Declaration

Tim French:



Andrew Gozzard:

Mark Reynolds:

Abstract

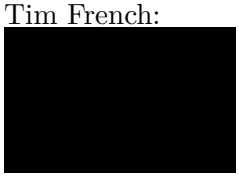
We consider multi-agent systems where agents actions and beliefs are determined aleatorically, or "by the throw of dice". This system consists of possible worlds that assign distributions to independent random variables, and agents who assign probabilities to these possible worlds. We present a novel syntax and semantics for such system, and show that they generalise Modal Logic. We also give a sound and complete calculus for reasoning in the base semantics, and a sound calculus for the full modal semantics, that we conjecture to be complete. Finally we discuss some application to reasoning about game playing agents.

Dynamic Aleatoric Reasoning in Games of Bluffing and Chance

This work has been published (see [18]). This work is related to my original research focus, and does not appear in this thesis.

Authorship Declaration

Tim French:



Andrew Gozzard:

Mark Reynolds:

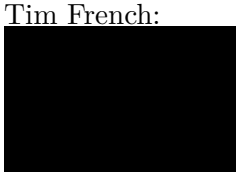

Abstract

Games of chance and bluffing, such as bridge, The Resistance, and poker allow epistemic reasoning. Players know their own cards while being uncertain of opponents'. Success generally involves reducing your uncertainty without reducing that of your opponents. Reasoning in such games requires a mix of logical (deducing what is possible) and probabilistic (what is likely). We present a *dynamic aleatoric logic* for epistemic reasoning in such games.

Aleatoric Dynamic Epistemic Logic for Learning Agents

This work has been published (see [17]). This work is related to my original research focus, and does not appear in this thesis.

Authorship Declaration

Tim French:


Andrew Gozzard:


Mark Reynolds:


Abstract

We propose a generalisation of dynamic epistemic logic, where propositions are aleatoric: that is, rather than having true/false values, propositions have odds of being true. Agents in such a system suppose a probability distribution of possible worlds, and based on observations are able to refine this probability distribution to match their observations. We demonstrate this logic with respect to some games of chance.

Polygon-Circle Graph Recognition Using Pseudocyclic Automata

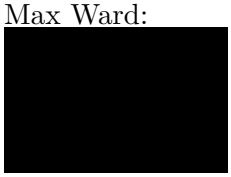
This work has been prepared for publication but has not been published. This work appears in Chapters 2 and 3.

Authorship Declaration

Andrew Gozzard:


Amitava Datta:


Gordon Royle:


Max Ward:


Abstract

Polygon-circle graphs (intersection graphs of polygons inscribed in a circle) are useful as a generalization of a number of other intersection graphs. A recognition algorithm for polygon-circle graphs has been of interest for some time. Despite this, no algorithm better than brute-force enumeration of alternating sequences has yet been published. We present a novel algorithm for recognizing polygon-circle graphs, and use this algorithm to show by complete search that the 3-prism is the minimal non-polygon-circle graph. Our algorithm is able to verify this result in a matter of milliseconds compared to the more than six hours required by the brute-force algorithm. Our algorithm makes polygon-circle graph recognition feasible on significantly larger graphs than was previously possible.

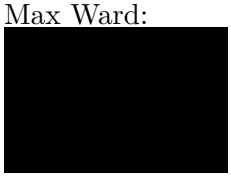
A Greedy Algorithm for Maximum Internal Spanning Tree in Interval Graphs

This work is being prepared for publication but has not been published. This work appears in Chapter 5.

Authorship Declaration

Andrew Gozzard:

Max Ward:



Amitava Datta:

Abstract

A *Maximum Internal Spanning Tree* (MIST) of a graph is a spanning tree with the maximum possible number of internal vertices or, equivalently, the minimum possible number of leaves. A Hamiltonian path of the graph, if one exists, must also be a MIST, as no spanning tree can have fewer than two leaves. It follows that in graphs in which finding a Hamiltonian path is NP-hard, MIST must also be NP-hard. However, there exist classes of graphs in which there are known polynomial-time algorithms for finding a Hamiltonian path. One such class is that of *interval graphs*, intersection graphs on intervals of a linear space, for which there exists an $O(|I|\log|I|)$ greedy algorithm to find a Hamiltonian path given an interval representation of the graph. We present an extension of this algorithm that is able to find a MIST of an interval graph, given its interval representation, in $O(|I|\log|I|)$ and a proof of correctness for this algorithm.

Chapter 1

Introduction

1.1 Overview of Contributions

I present several problems in the field of intersection graphs of geometric objects and algorithms for solving these problems.

Chapter 2 covers the problems of polygon-circle graph recognition and representation. Polygon-circle graphs have applications in bioinformatics, chemistry, and very large-scale integration of integrated circuits. The problems of polygon-circle graph recognition and representation have been of interest for some time, particularly given Koebe's announcement of a polynomial-time recognition algorithm [33] that was never successfully completed, and Pergel's later result that polygon-circle graph recognition is NP-complete [43]. I initially present Bouchet's alternating sequence representation of polygon-circle graphs [4] and use this to construct a naive, brute-force enumeration method for the representation problem. This approach leads to the exploration of many behaviourally equivalent states, which suggests the development of an $O(4^{|V|^2-|V|}|V|^4)$ dynamic programming solution that is much faster than the brute force method but requires $O(4^{|V|^2-|V|})$ memory. I then present another novel approach based on constructing optimal pseudocyclic deterministic finite automata that accept only valid alternating sequence [4] representations of a given polygon-circle graph. With appropriate construction techniques this algorithm requires $O(\sqrt{8}^{|V|^2-|V|}|V|^2)$ time and $O(\sqrt{8}^{|V|^2-|V|})$ memory at worst, though in practice it appears much better than this upper bound suggests. Empirical assessments show this algorithm to be more performant than the alternative methods, though it is of course still exponential in complexity. I demonstrate the practicality of this approach by using it to show by exhaustive search that the 3-prism is the minimal non-polygon-circle graph — a result that leads to, and is supported by, the proof in the Chapter 3.

Chapter 3 presents a constructive proof that polygon-circle graphs are a proper subclass of interval filament graphs. While this result was already known [19, 29], the proof of this used an involved analysis of the thresholds at which random graphs in the limit to infinity develop certain properties, and so was not a constructive proof. Based on the experimental finding from Chapter 2, I present a constructive proof that the 3-prism is an interval filament graph but not a polygon-circle graph, thereby providing a constructive proof that polygon-circle graphs are a proper subset of interval filament graphs. This marks the first such constructive proof of this property.

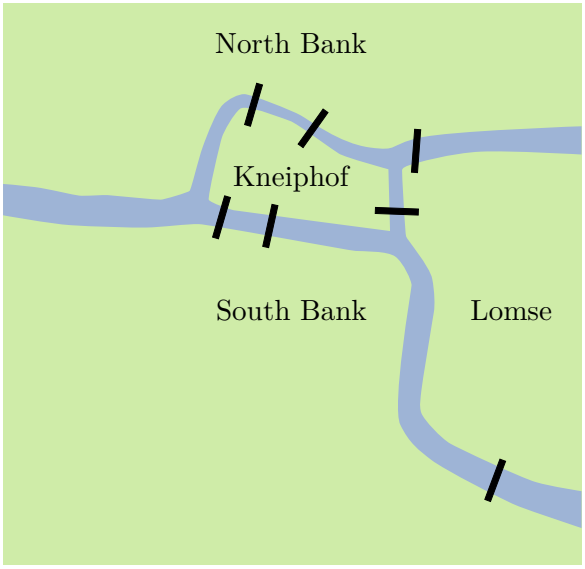
Chapter 4 covers my work on average path length minimisation through shortcut edge addition in circular-arc graphs. Circular-arc graphs and their subclass of interval graphs have applications in periodic and non-periodic scheduling tasks respectively. Ward and Datta [49] proposed a straightforward $O(|V|^3 + |S||V|^2)$ solution for general graphs by updating the distance matrix for each candidate edge. I present an algorithm of my own design based on computing the weight thresholds at which any new edge becomes a part of the shortest paths. Using cumulative sums of ordered ranges of these thresholds we can compute the change in total shortest path lengths in $O(|V|^3 \log|V| + |V||S| \log|V|)$, resulting in better performance for large $|S|$. Using an $O(|V|^2)$ all-pairs shortest path algorithm for circular-arc graphs by Saha, Pal, and Pal [45] it is then possible to construct an $O(|S||V|^2)$ algorithm, which is faster than Ward and Datta's algorithm in general, and faster than the threshold algorithm for $|S| \in O(|V| \log |V|)$, but is still outperformed by the threshold algorithm for large $|S|$. These methods generalise to subclasses of circular-arc graphs, including interval graphs.

Chapter 5 discusses algorithms for solving the maximum internal spanning tree (MIST) problem in interval graphs. Firstly we consider the state of the art algorithm by Li, Feng, Jiang, and Zhu [38] with stated time complexity $O(|I|^2)$ where $|I|$ is the number of intervals in the interval graph. I present a counterexample for which their algorithm fails to find a MIST and discuss lemmas used in their proof of correctness that appear not to hold in this case. The MIST problem can be considered a generalisation of the Hamiltonian path problem, because if a Hamiltonian path exists, it would also be a valid MIST. There exists a polynomial-time greedy algorithm for the Hamiltonian path problem in interval graphs by Manacher, Mankus, and Smith [39]. I present a novel generalisation of this algorithm to solve the MIST problem, and prove its correctness. This algorithm can be implemented in $O(|I| \log |I|)$ time, outperforming existing solutions, and is the only correct polynomial-time interval MIST algorithm of which I am aware.

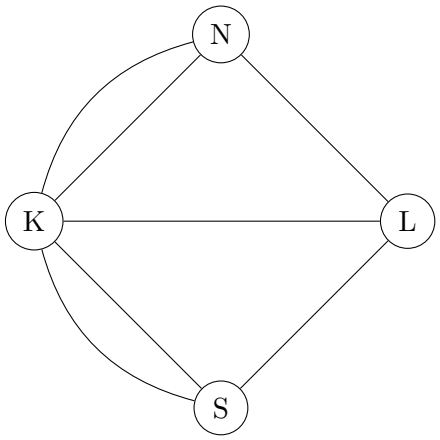
1.2 Graphs

The study of graphs as mathematical objects is typically considered to date back to Euler's paper on the Seven Bridges of Königsberg published in 1736. Königsberg spanned either side of the Pregel river and two islands, Kneiphof and Lomse, in the river itself, with the land masses connected by seven bridges as depicted in Figure 1.1. An open question at the time asked if there was any route through the city that would cross each bridge exactly once. By abstracting the land masses as vertices and the bridges as edges, Euler was able to reason about the properties any route through the city must have. In particular, for any vertex that is not the start or the end of the route, for every time we enter the vertex we must also exit it. This means that each vertex other than the start and end must have an equal number of edges crossed each way, and so must have an even number of edges in total. Therefore, since all four vertices had an odd number of edges, there could not possibly be a route that visited all of the bridges exactly once.

Definition 1.2.1. Simple Graph: A *simple graph* is a pair $G = (V, E)$ of a set of vertices V and a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V, u = v\}$. The presence of the unordered pair $\{u, v\} \in E$ denotes the existence of an edge between the vertices u and v. We say that the edge $\{u, v\}$ is *incident* on u and v, making them *adjacent*. The total number of edges incident on a vertex is called its *degree*.



(a) Map showing the bridges as they were in 1736.



(b) Graph representation of the land masses and the bridges between them. The vertices correspond to the North Bank (N), South Bank (S), Kneiphof (K) and Lomse (L).

Figure 1.1: Map and graph representation of the seven bridges of Königsberg.

This definition precludes the existence of parallel edges (multiple edges between the same pair of vertices) or loops (edges from a vertex to itself). While there exist problems for which it is useful to be able to represent parallel edges or loops, for the purpose of this work we consider only simple graphs without such features.

1.2.1 Definitions

We can make several definitions that are useful when talking about the properties and structures of various graphs.

Definition 1.2.2. Subgraph: A *subgraph* of a simple graph $G = (V, E)$ is a simple graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.

Definition 1.2.3. Complete Graph: A *complete graph* is a simple graph $G = (V, E)$ where $E = \{\{u, v\} \mid u, v \in V, u = v\}$.

Definition 1.2.4. Transitive Closure: The *transitive closure* of a graph $G = (V, E)$ is a graph $G' = (V, E')$ where $E' \supseteq E$ is the minimal superset of E such that $\forall \{u, v\}, \{v, w\} \in E' : \{u, w\} \in E'$.

Definition 1.2.5. Connected Graph: A *connected graph* is a simple graph for which its transitive closure is a complete graph.

Definition 1.2.6. Tree Graph: A *tree graph* or often simply *tree* is a connected graph $G = (V, E)$ such that $|V| = |E| + 1$. We call any vertices in the tree with degree 1 *leaves* and all other vertices *internal*.

Definition 1.2.7. Path Graph: A *path graph* is a tree with at most two leaves. We call the leaves the *ends* of the path graph.

Definition 1.2.8. Spanning Subgraph: A *spanning subgraph* of a simple graph $G = (V, E)$ is a subgraph $G' = (V, E')$ of G containing all vertices that appear in G.

Definition 1.2.9. Spanning Tree: A *spanning tree* of a connected graph G is a spanning subgraph of G that is also a tree.

1.3 Geometric Intersection Graphs

Of course, graphs do not have to directly model a physical system such as roads or bridges, and in general can represent much more complex systems. The study of graph theory therefore gives us powerful tools for working with such an abstract model. However, sometimes we may find graphs to be too general, making them able to represent structures that cannot exist in the systems we are attempting to model, and so not allowing us to solve problems or determine properties of these systems. By considering only classes of graphs that adhere to stricter rules, we can identify new properties while still being able to create meaningful models of various systems. In particular we are interested in the properties of various classes of intersection graphs.

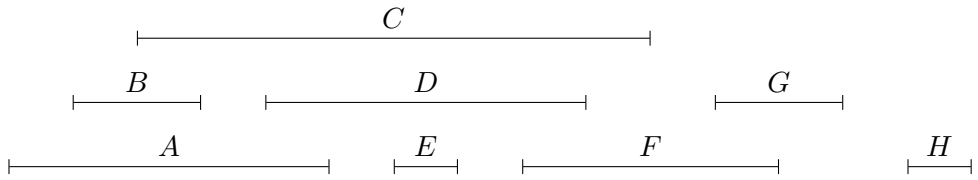
Definition 1.3.1. Graph Isomorphism: Consider some pair of simple graphs $G = (V, E)$ and $G' = (V', E')$. A *graph isomorphism* is a bijection $f : V \to V'$ such that $\{u, v\} \in E \iff \{f(u), f(v)\} \in E'$. We say G and G' are *isomorphic* if and only if there exists such an isomorphism.

Definition 1.3.2. Intersection Relation: We say two sets intersect if their intersection is nonempty. This corresponds to a binary relation $\between = \{(A, B) \mid A \cap B = \varnothing\}$. This notation is derived from the property that $A \between B \iff \exists\, x : A \ni x \in B$.

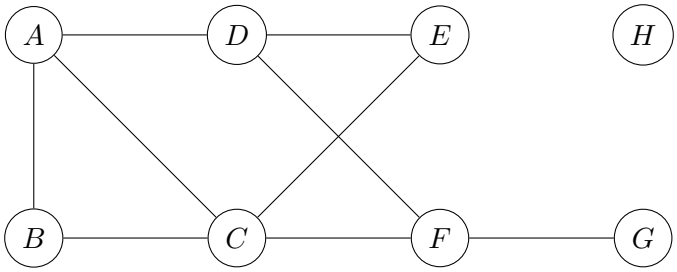
Definition 1.3.3. Intersection Model: An *intersection model* of a graph $G = (V, E)$ is a function f that maps each vertex to some set such that $\forall\, u, v \in V : f(u) \between f(v) \iff \{u, v\} \in E$. We say an intersection model is an intersection model of some class of objects if its codomain is constrained to that class. We say two intersection models are equivalent if and only if they are intersection models of the same graph.

Definition 1.3.4. Intersection Graph: An *intersection graph* is a graph G for which an intersection model exists. We say a graph is an intersection graph of some class of objects if there exists an intersection model for that graph that is an intersection model of that class.

Intersection graphs are sufficiently general to represent any graph. For example, we can provide an intersection model for any graph $G = (V, E)$ by mapping every vertex to the set of edges incident on that vertex: $f(v) = \{e \in E \mid v \in e\}$. For any pair of vertices $u, v \in V$ their intersection under this model $f(u) \cap f(v)$ is therefore the set containing just the edge $\{u, v\}$ if and only if such an edge exists in E, as is required of an intersection model. As a result, intersection graphs in general do not have any properties that meaningfully distinguish them from general graphs. Rather, it is when the intersection model is subject to some constraints that may result in the intersection graph having useful properties. Of particular interest are the properties of intersection graphs constrained to various classes of geometric objects.



(a) A set of intervals. Shown vertically separated for clarity.



(b) The intersection graph of the intervals.

Figure 1.2: An interval graph shown both as a set of intervals and their intersection graph.

Definition 1.3.5. Geometric Intersection Graph: A *geometric intersection graph* is an intersection graph of some class of geometric objects.

Geometric intersection graphs are notable for two reasons. Firstly, it is common for them to appear naturally as models in various applications depending on the geometry of the system in question. Secondly, the properties that arise from these geometric objects can lead to optimisations that make various problems tractable on geometric intersection graphs that are hard in general graphs. The following sections present a taxonomy of a number of classes of geometric intersection graphs with interesting properties relevant to this thesis.

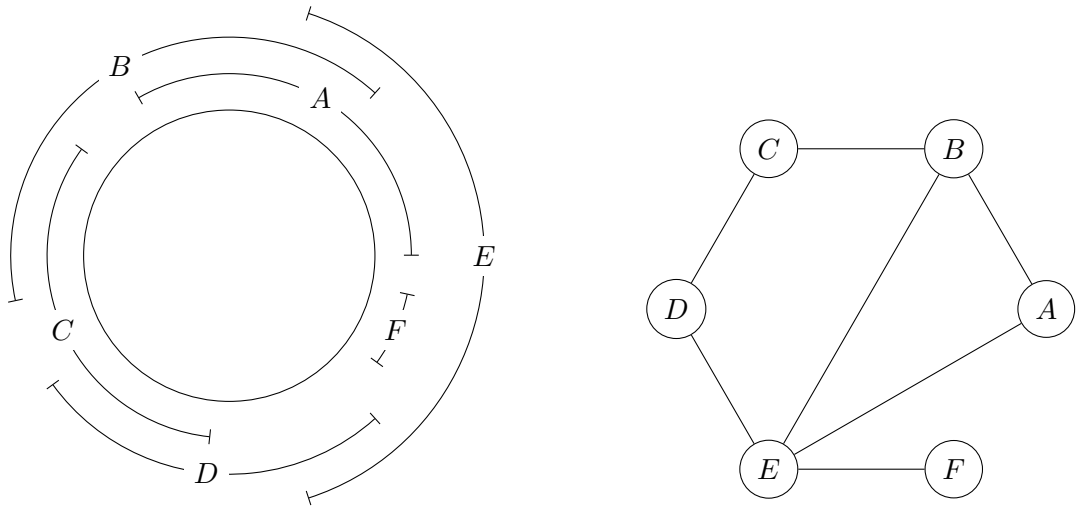
1.3.1 Interval Graphs

Definition 1.3.6. Interval Graph: An *interval graph* is an intersection graph of intervals of the real line. See Figure 1.2 for an example.

In general, the ends of an interval may be *open*, meaning the endpoint itself is not part of the interval, or *closed*, meaning it is. In the case of interval graphs, however, since all that matters is whether or not a pair of intervals intersect, there exists an equivalent interval intersection model in which all intervals have distinct endpoints. Specifically, if any two intervals share an endpoint, it is possible to offset the endpoint of one of the intervals by an infinitesimal amount without affecting whether or not they intersect, and so we can separate all endpoints while maintaining the intervals as a valid interval model of the graph. Without loss of generality we may therefore assume that no intervals in the interval model of an interval graph share endpoints.

Therefore, if we denote the left endpoint of an interval x as $l(x)$ and the right endpoint as $r(x)$, we find $x \between y \iff l(x) < r(y) \wedge l(y) < r(x)$ and the contrapositive $x \not\between y \iff r(x) < l(y) \vee r(y) < l(y)$.

Interval graphs have applications including Very Large-Scale Integration (VLSI) circuit layout [26] and fixed interval scheduling [35]. It is worth noting that we do not need to know the exact intervals



(a) A set of arcs of a circle. Shown radially separated for clarity.

(b) The intersection graph of the arcs.

Figure 1.3: A circular-arc graph shown both as a set of arcs and their intersection graph.

corresponding to each vertex in order to reason about their intersection graph. An example of this is the Berge Mystery, named after Claude Berge, to whom the original puzzle is commonly attributed [21].

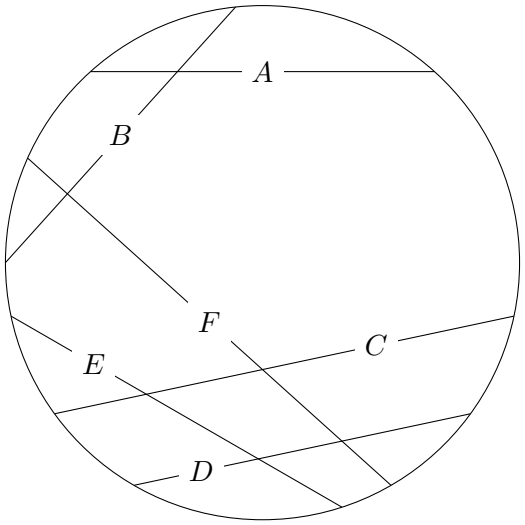
In the Berge Mystery, suspects in a theft testify as to which other suspects they saw at the scene of the crime. If it is known that each suspect visited the location exactly once, and no two suspects were present at the same time without one of them seeing the other, then their visits can be represented as intervals in time and their testimony as intersections between intervals and must therefore form an interval graph. Therefore if the graph formed by the suspects' testimony is not an interval graph, one of the suspects must have lied.

1.3.2 Circular-Arc Graphs

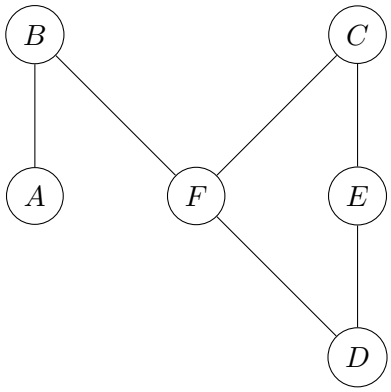
Definition 1.3.7. Circular-Arc Graph: A *circular-arc graph* is an intersection graph of arcs of a common circle. See Figure 1.3 for an example.

Of particular note is the fact that all interval graphs are also circular-arc graphs, and so interval graphs are a subclass of circular-arc graphs. As long as the relative order of their endpoints is maintained, any interval model can be transformed to an equivalent intersection model of subintervals of some finite interval. Projecting this finite interval onto the circumference of a circle allows us to transform any interval model into an equivalent circular-arc model. This means that any property that holds for all circular-arc graphs also holds for all interval graphs.

Conversely for any circular-arc graph with some point on the circle not covered by any arc, the circle could be split at that point and mapped to a segment of the real line. Under this mapping all arcs become intervals of the real line without changing their intersections, and so must also be an equivalent interval intersection model.



(a) A set of chords in a circle.



(b) The intersection graph of the chords.

Figure 1.4: A circle graph shown both as a set of chords and their intersection graph.

Like interval graphs, circular-arc graphs have applications in VLSI layout and scheduling [28]. Notably, however, they are able to represent cyclic schedules where interval graphs cannot. Circular-arc graphs also have applications in genetics and bioinformatics [28, 40].

1.3.3 Circle Graphs

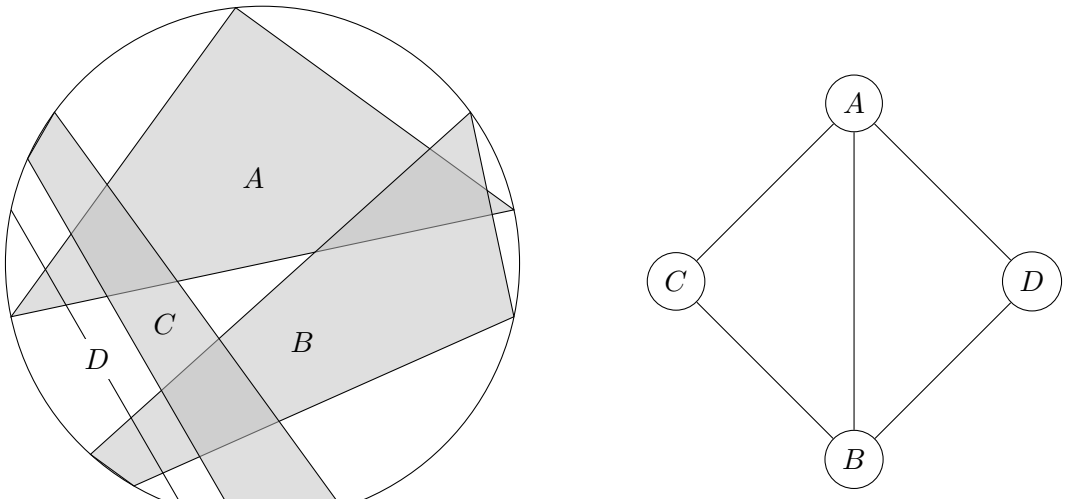
Definition 1.3.8. Circle Graph: A *circle graph* is an intersection graph of chords inscribed in a common circle. See Figure 1.4 for an example.

Circle graphs have numerous practical applications [50, 51]. They can be used to model layout problems in VLSI design, including channel routing and switch-box routing [47]. Circle graphs can also be used to solve problems relating to molecular structure in chemistry and bioinformatics. Bonsma and Breuer [3] used circle graphs to enumerate benzenoid hydrocarbons and fullerenes. The possible base pairings of nucleotides in ribonucleic acid (RNA) form a circle graph, which can be used to find maximally bonded RNA secondary structures [42].

1.3.4 Polygon-Circle Graphs

Definition 1.3.9. Polygon-Circle Graph: A *polygon-circle graph* is an intersection graph of polygons inscribed in a common circle. See Figure 1.5 for an example.

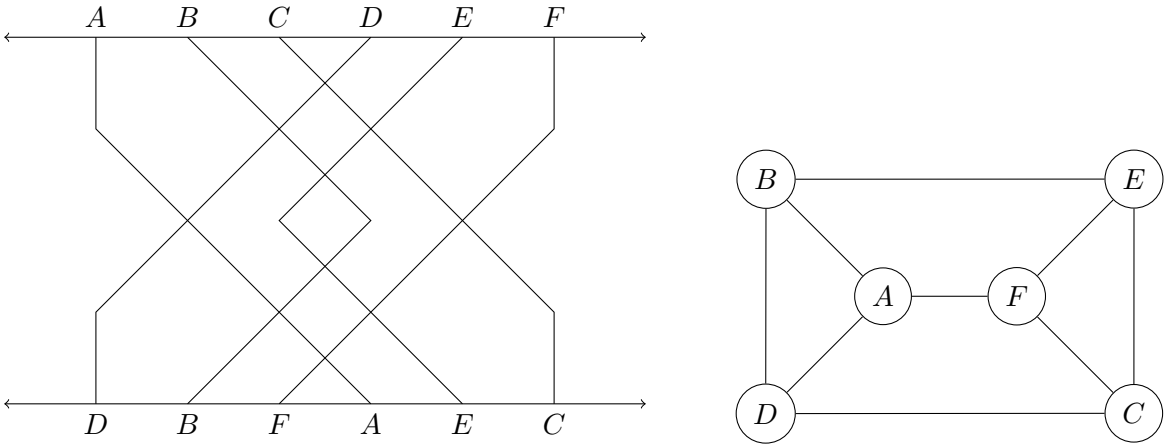
Polygon-circle graphs generalise a number of other intersection graph classes, including circle graphs and circular-arc graphs [33]. In particular, they are the minimal superclass of circle graphs that is closed under edge contraction [34], and so are useful when reasoning about connectivity between components of a circle graph.



(a) A set of polygons inscribed a circle.

(b) The intersection graph of the polygons.

Figure 1.5: A polygon-circle graph shown both as a set of inscribed polygons and their intersection graph.



(a) A set of curves between parallel lines.

(b) The intersection graph of the curves.

Figure 1.6: A co-comparability graph shown both as a set of curves between parallel lines and their intersection graph.

1.3.5 Co-comparability Graphs

Definition 1.3.10. Co-comparability Graph: A *co-comparability graph* is an intersection graph of continuous curves from a line to a parallel line. See Figure 1.6 for an example.

Due to this intersection model, co-comparability graphs are also known as *function graphs*. Co-comparability graphs are a superclass of *permutation graphs*, which are the intersection graphs of line segments spanning between a pair of parallel lines. Despite having a geometric intersection model, co-comparability graphs are more commonly known for their properties as the complements of comparability graphs. This makes them useful as an analytical tool in proofs such as that presented in Chapter 3.

Definition 1.3.11. k-subset: A k-subset of a set S is a subset of S of size k. The set of all k-subsets of S is denoted $\binom{S}{k} = \{X \subseteq S \mid |X| = k\}$.

Definition 1.3.12. Graph Complement: The *complement* of a graph $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E})$ where $\overline{E} = \binom{V}{2} \setminus E$ is the set of only those edges that do not appear in G and no others.

Definition 1.3.13. Comparability Graph: The *comparability graph* of a strict partially ordered set (V, R) is the graph $G = (V, E)$ where $E = \{\{u, v\} \in \binom{V}{2} \mid (u, v) \in R \vee (v, u) \in R\}$ is the set of all pairs in V that are comparable using the relation R.

Lemma 1.3.1. *(Golumbic, Rotem, and Urrutia [22]) The class of co-comparability graphs is exactly the class of the complements of comparability graphs.* □

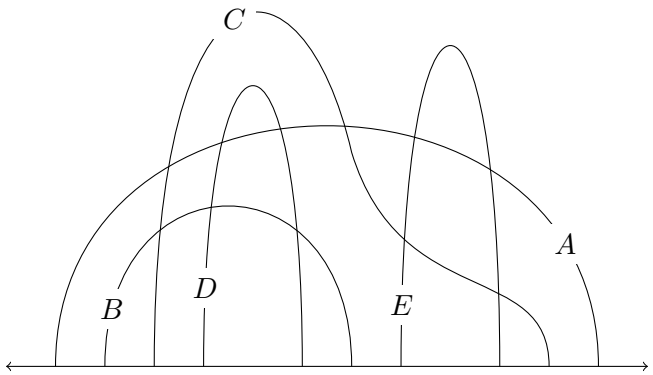
Janson and Kratochvıl [29] use a non-constructive, probabilistic method to show that neither polygon-circle graphs nor co-comparability graphs contain the other as a subclass. This method is based on analysing the evolution of various properties of the random graph $G_{n,p}$ of n vertices with connection probability p. The probability of $G_{n,p}$ having some particular property varies with n and p, but in particular, in the limit as $n \to \infty$, small changes in p can result in drastic changes in this probability. Janson and Kratochvıl combine a number of analytical results to find upper and lower threshold functions for p for a number of properties. By showing for certain classes of intersection graph that there are values of p for which the random graph must belong to one class but not the other, they are able to show that the former is not a subclass of the latter. They present several results including that neither polygon-circle graphs nor co-comparability graphs are a subclass of the other.

1.3.6 Interval Filament Graphs

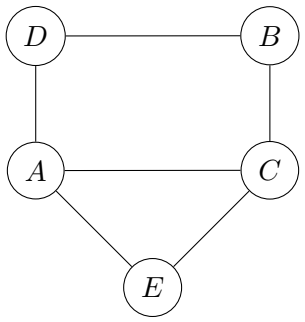
Definition 1.3.14. Interval Filament [2]: Let $l, r \in \mathbb{R}$, with $l \leq r$. A (possibly self-intersecting) curve $C \subset \mathbb{R}^2$ is an *interval filament* with endpoints l and r if:

- $y \geq 0$ for all $(x, y) \in C$,
- $(l, 0) \in C$, $(r, 0) \in C$, and
- $l \leq x \leq r$ for all $(x, y) \in C$.

See Figure 1.7a for some examples.



(a) A set of interval filaments.



(b) The intersection graph of the interval filaments.

Figure 1.7: An interval filament graph shown both as a set of interval filaments and their intersection graph.

Definition 1.3.15. Interval Filament Graph: An *interval filament graph* is an intersection graph of interval filaments. See Figure 1.7 for an example.

Gavril [19] shows that both polygon-circle graphs and co-comparability graphs are subclasses of interval filament graphs. Combining this with the result from Janson and Kratochvıl [29] mentioned in Section 1.3.5 gives a proof that polygon-circle graphs and co-comparability graphs are *proper* subclasses of interval filament graphs.

1.4 Problems of Interest

1.4.1 Recognition

As discussed in Section 1.3, graphs are interesting not only in their more general form, but also owing to the properties that arise in various classes of graphs because of the defining properties of these classes. Many other problems of interest in graphs may be easier to solve in such classes than in general graphs, but in order to take advantage of this we must know whether or not a graph belongs to the desired class. This is the recognition problem.

Definition 1.4.1. Recognition Problem: For some specified class of graphs, determine whether a given graph belongs to that class. Note that since the graph classes we are interested in are closed under vertex relabelling, the precise object used to represent any vertex cannot affect whether it belongs to that class, and so membership of these classes is based solely on the adjacency structure of the graph.

1.4.2 Intersection Representation

For some problems, it is not sufficient to know whether or not a graph belongs to a particular class in order to benefit from the properties of that class. Rather, we may need to construct an explicit representation of the graph under the constraints that define the class. Particularly in the case of intersection graphs, this means finding an intersection model (see Definition 1.3.3) for the graph.

Definition 1.4.2. Intersection Representation Problem: Given a class of intersection graphs and a graph of that class, determine an intersection model of the given graph for the specified class.

Having a known intersection model allows us to use any properties of the given class of graph, when this might not previously have been possible. In this way it can be more valuable to find an intersection representation of a graph than simply to recognise it as belonging to a particular class of intersection graphs.

In some cases we may wish to specify additional requirements for acceptable solutions to this problem. Doing so depends on the properties of the particular class in question, and may make the problem substantially more difficult. For example, the complexity of interval graph representation varies drastically based on what additional constraints are supplied. With no additional constraints, it is possible to find a representation of an interval graph in linear time $O(|V| + |E|)$. The problem becomes NP-complete if each interval in the representation is required to be a specific length. However, if we are also given the required length of the intersections of each pair of intervals, the problem can once again be solved in $O(|V| + |E|)$ [31].

1.4.3 Subclass Relation

A class of graphs is a subclass of another class if and only if all graphs that are members of the former class are also members of the latter. This property is useful to know, as any property that holds for the superclass must also hold for the subclass.

1.4.4 Shortest Path

It can be useful to associate additional information with the elements of a graph, rather than simply their presence or absence. In general we could use some function to associate this information (typically called a label) with each vertex or edge. There are many applications in which it is useful to associate some numeric value with each edge. This value is typically referred to as a *weight*, and can be used to represent quantities such as cost or length.

Definition 1.4.3. Weighted Graph: A *weighted graph* is some graph G with edges E and an associated *weight function* $w : E \to W$ where W is some numeric space.

In systems which can be modelled as graphs, it is not always the case that we are restricted to only make use of single edges in the graph, but rather we may wish to reason about the transitive connectivity of the graph. We can extend the concept of weight to transitive connectivity by combining the weights of multiple edges.

Definition 1.4.4. Path: A *path* in a simple graph $G = (V, E)$ is a subgraph of G that is a path graph. We say a path goes *from* one end of the path *to* the other. The *length* of a path in a weighted graph is the sum of the weights of all edges in the path. In an unweighted graph it is simply the number of edges in the path, as though every edge in the graph were given a weight of 1. Paths may also be specified as a sequence of vertices $\pi = \langle v_1, v_2, \ldots, v_n \rangle$ with edges assumed to exist between each adjacent pair of vertices in the sequence $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}$.

We often encounter optimisation problems that can be modelled as finding a path with minimum possible length. This gives rise to a family of shortest path optimisation problems.

Definition 1.4.5. Shortest Path: Given a graph G with vertices V and a pair of vertices $u, v \in V$ a *shortest path* from u to v is a path with minimum length out of all possible paths from u to v. Note there may be multiple such shortest paths. The shortest path problem is solved by determining one such shortest path given G, u, and v. Sometimes this problem may be stated as simply finding the length of such a shortest path and not necessarily the path itself.

Definition 1.4.6. Single-Source Shortest Paths: Given a graph G with vertices V and a vertex $u \in V$, the *single-source shortest paths* problem is solved by finding a shortest path from u to each other vertex in V, or in some cases simply the lengths of these paths.

Definition 1.4.7. All-Pairs Shortest Paths: Given a graph G with vertices V the *all-pairs shortest paths* problem is solved by finding a shortest path between every pair of vertices $u, v \in V, u = v$, or in some cases simply the lengths of these paths.

There are many famous solutions to these problems including Dijkstra's algorithm [12] and the Floyd-Warshall algorithm [14] for all-pairs shortest path lengths [10].

1.4.5 MinAPL

Since many optimisation problems can be solved using the shortest path problem, the average length of all shortest paths in a graph can be a useful metric for comparing graphs against each other in these contexts. Indeed, the average path length in a graph gives us the expected value for the length of the shortest path between a uniformly randomly sampled pair of vertices.

Definition 1.4.8. Average Path Length: The *average path length* of a graph G is the average of all the lengths in the all-pairs shortest paths of G.

In cases where we have some amount of agency to modify the system, we can use this to attempt to improve the average path length.

Definition 1.4.9. Average Path Length Minimisation: The *Average Path Length Minimisation* (MinAPL) problem is to find, given a graph $G = (V, E)$ and a set of *candidate edges* $S \subseteq \{\{u, v\} \mid u, v \in V, u = v\} \setminus E$, the edge e in S that, when inserted into G to give $G' = (V, E \cup \{e\})$, results in G' having the lowest average path length possible out of all candidates. Note that there may be multiple such edges. In the case of a weighted graph the weight function w must also be defined for all candidate edges in S.

1.4.6 Maximum Internal Spanning Tree

Spanning trees are often useful when optimising infrastructure, as they are the minimum subgraphs that span every vertex in a graph, and removing any edge from a spanning tree would cause it to become disconnected. A particular example of this is the problem of optimising communication networks by finding a spanning tree with the minimum possible number of leaves [46]. Conventionally this is referred to as the maximum internal spanning tree problem, as maximising the number of internal vertices in a spanning tree is equivalent to minimising the number of leaves.

Definition 1.4.10. Maximum Internal Spanning Tree: A *maximum internal spanning tree* (MIST) of a connected graph G is any spanning tree of G for which no other spanning tree has a greater number of internal vertices. The maximum internal spanning tree problem is to, given G, find a MIST of G.

The maximum internal spanning tree problem generalises the Hamiltonian path problem, since if a Hamiltonian path exists, it must also be the MIST, as it would not be possible to have fewer leaves than a path.

Definition 1.4.11. Hamiltonian Path: A *Hamiltonian path* in a graph G is any path that is also a spanning subgraph of G. The Hamiltonian path problem is to, given G, find a Hamiltonian path in G, if one exists.

Chapter 2

Polygon-Circle Graph Recognition and Representation

2.1 Introduction

This chapter covers the recognition (see Section 1.4.1) and representation (see Section 1.4.2) problems for polygon-circle graphs (see Section 1.3.4).

Polygon-circle graphs are useful as a generalisation of a number of other intersection graph classes, including circle graphs and circular-arc graphs, and are a subclass of interval filament graphs [19]. In particular, they are the minimal superclass of circle graphs that is closed under edge contraction. A number of optimisation problems in these graph classes have polynomial-time algorithms only if the intersection representation of the graph is already known [6, 7, 8, 50, 51], so it is valuable to be able to recognise when a graph is a polygon-circle graph and construct an intersection representation of the graph.

Several polynomial-time recognition algorithms exist for circle graphs [48, 20]. Koebe published a sketch of a polynomial-time recognition algorithm for polygon-circle graphs (using the alternative term of "spider graph"), though this sketch was later found to contain errors, and no completed algorithm was ever published [32, 33]. Pergel [43] shows that recognition of both polygon-circle and interval filament graphs is NP-complete by reduction from the Not-All-Equal Satisfiability (NAE-SAT) problem, but does not explicitly provide an algorithm for recognising polygon-circle graphs. Given how otherwise similar polygon-circle graphs are to circle graphs, it is interesting that polygon-circle graph recognition is NP-complete when circle graph recognition is possible in polynomial time. Despite this problem plainly having been a matter of some interest for some time, I am not aware of any existing polygon-circle graph recognition algorithm that is faster than brute-force enumeration of alternating sequences (see Sections 2.2 and 2.3).

This chapter will explore algorithms for polygon-circle graph recognition and representation and compare their theoretical and practical performance. In particular the novel algorithms presented in Sections 2.4 and 2.5, while still having exponential time complexities, are superexponentially faster than the brute force.

2.2 Alternating Sequence Representation

Under the purely geometric interpretation of polygon-circle graphs, there are infinitely many equivalent geometric representations, because polygon vertices can move freely on the circle and, so long as their order around the circle remains the same, the intersection relation on the polygons will not change. In order to simplify working with polygon-circle graphs, we will exploit this property to define an equivalent non-geometric representation. This is the *alternating sequence* representation first presented by Bouchet [4]. A graph is a polygon-circle graph if and only if it has an alternating sequence representation [43].

Definition 2.2.1. Alternating Sequence: An *alternating sequence* is a representation of a simple graph $G = (V, E)$ as a sequence $\mathcal{S} = S_1 \dots S_m$ of symbols taken from $V = \{v_1, \dots, v_n\}$ such that $(uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S}) \iff \{u, v\} \in E$. The notation $\mathcal{S}' \mid \mathcal{S}$ means $\mathcal{S}$ contains $\mathcal{S}'$ as a subsequence (and $\mathcal{S}' \nmid \mathcal{S}$ means it does not).

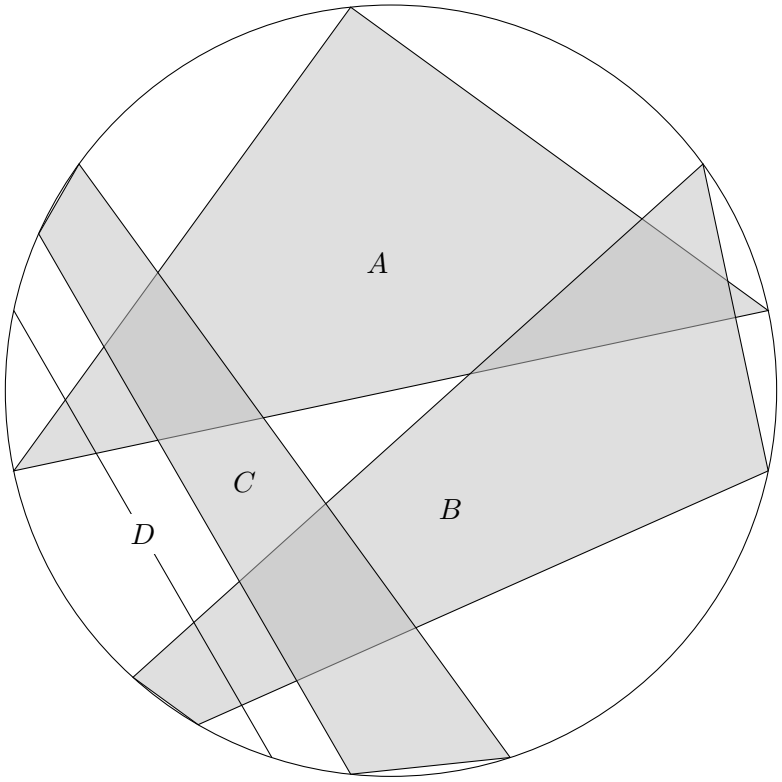
Note that any rotation or reversal of such an alternating sequence is an equivalent representation, as the subsequences $uvuv$ and $vuvu$ are each others rotations and reversals, and so the represented edges will not change under rotation or reversal. For a disconnected graph, each connected component of the graph need not intersect each other at all, and so the overall sequence can simply be the concatenation of the representation of each component. As such, and without loss of generality, we need consider only connected graphs.

As an example, the polygon-circle graph shown in Figure 2.1a can be represented by the alternating sequence $ABACCDABBDCCB$, corresponding to the counter-clockwise order in which vertices appear on the circle as demonstrated in Figure 2.1b. Rotating the sequence corresponds to changing the starting point of the sequence on the circle. Note that other alternating sequence representations for this graph exist.

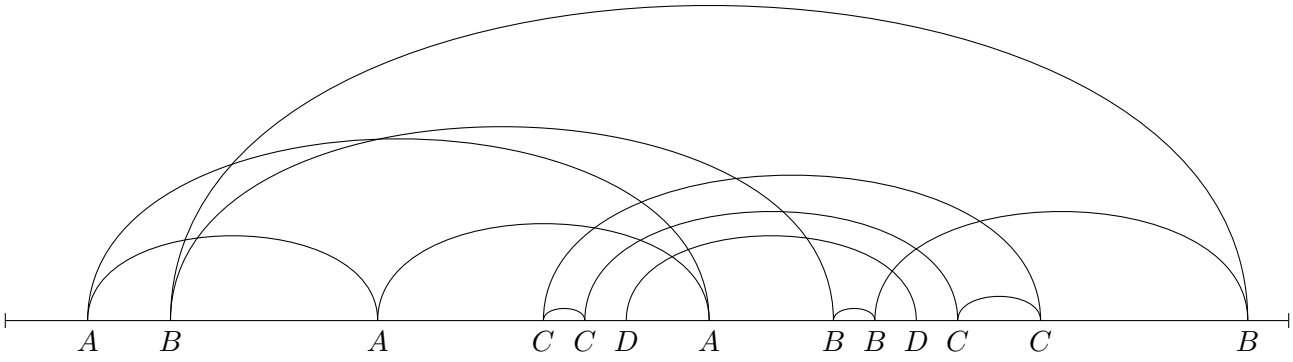
Recognising polygon-circle graphs is thereby reduced to determining whether there exists an alternating sequence that encodes the given graph. Note that the number of occurrences of a symbol in an alternating sequence corresponds to the number of vertices that polygon has, meaning circle graphs can be represented by an alternating sequence where each symbol occurs exactly twice.

Neither of the subsequence patterns corresponding to edges given in Definition 2.2.1 require the same symbol to be repeated twice in immediate succession. As a result, if the same symbol ever appears twice in succession in an alternating sequence, one of the occurrences can be removed without affecting the graph represented by the sequence. We may therefore assume without loss of generality that no alternating sequence ever has the same symbol appear twice in a row. By the same reasoning, we may also therefore assume that no alternating sequence starts and ends with the same symbol, which follows intuitively from the above observation and the fact that rotating the circle to change where we start the sequence does not affect the intersection relation of the polygons.

Lemma 2.2.1. *(Kratochvıl and Pergel [36]) For any polygon-circle graph $G = (V, E)$ and any vertex $v \in V$ with degree $d(v)$, v need only appear at least twice and at most $\max(2, d(v))$ times in the alternating sequence.* □



(a) A set of polygons inscribed in a circle — the intersection representation of a polygon-circle graph.



(b) The straightened circle showing the relationship between the inscribed polygons and their alternating sequence representation $ABACCDABBDCCB$. This example sequence is read in counter-clockwise order starting from the rightmost point of the circle, but the choice of direction and cut point are arbitrary.

Figure 2.1: A polygon-circle graph shown both as a set of inscribed polygons and their corresponding alternating sequence.

Observation 2.2.2. *For any polygon-circle graph $G = (V, E)$, a minimum-length alternating sequence representation of G need be at most $2|E| + |L|$ in length where $L = \{v \in V \mid d(v) = 1\}$ is the set of leaves of G.*

Proof. The length of the alternating sequence representation is the sum of the number of times each vertex appears in the sequence. By Lemma 2.2.1, each vertex with degree $d(v)$ need only appear at least twice and at most $\max(2, d(v))$ times in the alternating sequence. Since the graph is connected, every vertex must have $d(v) \geq 1$, and therefore the the only vertices that must appear more than $d(v)$ times will be the leaves, which will appear exactly $2 = d(v) + 1$ times, and all others will need to appear at most $d(v)$ times. Therefore the minimum required length for an alternating sequence representation of G is given by the sum $\sum_{v \in V \setminus L} d(v) + \sum_{v \in L}(d(v) + 1)$. Grouping the $d(v)$ terms into the same sum gives $\sum_{v \in V} d(v) + \sum_{v \in L} 1$. Since each edge counts towards the degree of exactly two vertices, the sum of degrees across all vertices will be exactly $2|E|$. Therefore the length of the alternating sequence needs be at most $\sum_{v \in V} d(v) + \sum_{v \in L} 1 = 2|E| + |L|$, as desired. □

2.3 Brute Force

The alternating sequence representation implies a brute-force algorithm for polygon-circle graph recognition and representation. We need simply to enumerate all possible alternating sequences for the vertices of the given graph and test each sequence to see if it represents exactly the edge set of the given graph. We can enumerate these sequences by constructing the lexicographically minimal sequence containing each symbol the appropriate number of times based on the degree of that vertex and then iterate through all distinct permutations of this sequence in lexicographic order using a classical method [30].

As per Lemma 2.2.1, let $n : V \to \mathbb{N}_{\geq 0}$ be the number of times a vertex appears in the sequence where $n(v) = 2$ when the degree $d(v)$ of v is 1 and $n(v) = d(v)$ otherwise. The total length of the sequence is then $N = \sum_{v \in V} n(v)$. The number P of distinct permutations of this sequence can be found by dividing out the number of permutations of each symbol from the number of permutations of the whole sequence $P = N!/\prod_{v \in V} n(v)!$. For any particular N, P is maximised when all vertices have n-values that differ by at most 1 from each other. If this were not the case, and a pair of vertices u and v existed such that $n(u)+1 < n(v)$, then we could increase $n(u)$ by 1 and decrease $n(v)$ by 1. This would result in a total proportional change in the denominator of P, $\prod_{v \in V} n(v)!$, of $(n(u)+1)/n(v)$, which is guaranteed to be less than 1 and hence increase P. Assuming then that in the worst case every vertex has some common degree $d > 1$, P has an upper bound given by $N!/d!^{|V|}$ where $N = |V|d = 2|E|$. Even with the factorial term in the denominator, this grows superexponentially as d increases, and is bounded by the limit $d < |V|$, giving a worst case upper bound for P of $(|V|^2 - |V|)!/(|V| - 1)!^{|V|}$ when $N = |V|(|V| - 1) = |V|^2 - |V|$.

Naively, an alternating sequence can be checked in $O(|V|^2 N)$ time by iterating over the sequence to test for the presence or absence of each edge pattern as a subsequence of the alternating sequence as required. Doing this for each candidate alternating sequence gives a time complexity of $O(|V|^2 NP)$,

which in full gives the worst case time complexity:

$$O\left(\frac{|V|^2(|V|^2-|V|)(|V|^2-|V|)!}{(|V|-1)!^{|V|}}\right)$$

To the best of my knowledge, this algorithm was the best known prior to my development of the algorithms presented in Sections 2.4 and 2.5.

2.4 Dynamic Programming

Lemma 2.4.1. *$\mathcal{S}$ is an alternating sequence representation of $G = (V, E)$ if and only if $(uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S}) \iff \{u, v\} \in E$.*

Proof. By Definition 2.2.1, $\mathcal{S}$ is an alternating sequence representation of G if and only if $(uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S}) \iff \{u, v\} \in E$. We therefore aim to show that $(uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S}) \iff (uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S})$ by considering the conditional in either direction.

To show $(uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S}) \implies (uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S})$, first consider that any $\mathcal{S}$ that satisfies the left hand side must contain at least two us as a result of $uvu \mid \mathcal{S}$ and at least two vs as a result of $vuv \mid \mathcal{S}$. We can explicitly enumerate all distinct possible orderings of these symbols as follows: $uuvv$, $uvuv$, $uvvu$, $vuuv$, $vuvu$, $vvuu$. The only two of these options that satisfy the left hand side are $uvuv$ and $vuvu$, and hence at least one of them must also be a subsequence of $\mathcal{S}$, giving us the right hand side $(uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S})$ as desired.

To show $(uvuv \mid \mathcal{S}) \vee (vuvu \mid \mathcal{S}) \implies (uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S})$ we can simply consider both the cases that $uvuv \mid \mathcal{S}$ and that $vuvu \mid \mathcal{S}$. If $uvuv \mid \mathcal{S}$, we see that $uvu \mid uvuv$ and that $vuv \mid uvuv$. If $vuvu \mid \mathcal{S}$, we see that $uvu \mid vuvu$ and that $vuv \mid vuvu$. Therefore in either case $uvu \mid \mathcal{S}$ and $vuv \mid \mathcal{S}$ as desired. □

By Lemma 2.4.1, in order to determine if a sequence $\mathcal{S}$ is an alternating sequence representation of a graph $G = (V, E)$, it suffices to show that $\forall\{u, v\} \in E : (uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S})$ and conversely that $\forall\{u, v\} \notin E : (uvu \nmid \mathcal{S}) \vee (vuv \nmid \mathcal{S})$. To test for the presence of any single subsequence of the form uvu we can simply scan linearly over the sequence, keeping track of how long a prefix of the subsequence we have so far encountered, starting from 0. Finding a prefix of length 3 indicates that the subsequence does appear in the sequence. Keeping track of such a prefix length for every sequence of the form uvu for all $u, v \in V$ allows us to test if this alternating sequence is a valid representation of a given graph, as for all $\{u, v\} \in E$, both uvu and vuv must have a prefix length of 3, while for all $\{u, v\} \notin E$, at least one of the two must have length less than 3. This allows us to check a candidate alternating sequence of length N in $O(|V|N)$ time, because for each symbol in the sequence we must check and potentially update the length of all $O(|V|)$ pairs in which that symbol appears.

While we could use this technique to enumerate and test all possible alternating sequences, like the brute force this is dominated by the number of sequences that need to be checked and is super-exponential. Instead, if we analyse the state used in this algorithm, we find that we only need to keep track of a prefix length from 0 to 3 for each pair and how many characters into the sequence we are. As there are $|V|(|V| - 1) = |V|^2 - |V|$ distinct ordered pairs, this gives us an upper bound on the number of possible states of $4^{|V|^2-|V|}N$ where N is the sequence length, which is bounded by

$O(|E|)$, as above. At any state there are $|V|$ possible symbols that could come next, and for each the resulting next state can be found in $O(|V|)$ by updating the prefix lengths in which that symbol appears. Memoising which states are able to reach a base case where the requirements for the graph in question are satisfied gives us a dynamic programming based solution with $O(4^{|V|^2-|V|}N)$ states and an $O(|V|^2)$ recurrence for an overall time complexity of $O(4^{|V|^2-|V|}N|V|^2)$. Given N cannot exceed $|V|^2$, this algorithm has an upper bound on time complexity of $O(4^{|V|^2-|V|}|V|^4)$ and naively a memory complexity of $O(4^{|V|^2-|V|}|V|^2)$ states. However, since part of the state is the current length of the sequence, and a state can only depend on the immediate next length, we can evaluate states in order by length and thereby discard old states that we will never again depend on. By keeping only states corresponding to the current and previous lengths, this reduces the memory complexity to $O(4^{|V|^2-|V|})$.

If this dynamic programming approach finds the given graph to be a polygon-circle graph, we can use standard techniques to trace back the choices made by the algorithm at each state to construct a valid alternating sequence representation of the given graph, and hence an intersection representation. This technique requires us to maintain all states, however, and so is incompatible with the memory reduction technique from above.

2.5 Recognition Automata

2.5.1 Deterministic Finite Automata

The alternating sequence representation also suggests another approach to the recognition and representation problems. Since we are now working with sequences, we can employ a well understood mathematical tool for working with such sequences: Deterministic Finite Automata (DFAs). We aim, therefore, to construct a DFA that accepts any sequence that is an alternating sequence for the given graph, and rejects all others.

Definition 2.5.1. Deterministic Finite Automaton: A *deterministic finite automaton* can be defined by a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet of symbols, $\delta : Q \times \Sigma \to Q$ is a transition function, q_0 is some initial state, and $F \subseteq Q$ is a set of accepting states. A deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ accepts a sequence $\mathcal{S} = S_1 \dots S_m$ if and only if there exists a sequence of states $R_0 \dots R_m$ such that $R_0 = q_0$, $R_i = \delta(R_{i-1}, S_i)$, and $R_m \in F$.

It is also possible to combine DFAs in various ways to construct a new DFA. For example (and of relevance later), we can define the conjunction of a pair of DFAs to be a new DFA that accepts only those sequences that both original DFAs would accept, and rejects all others.

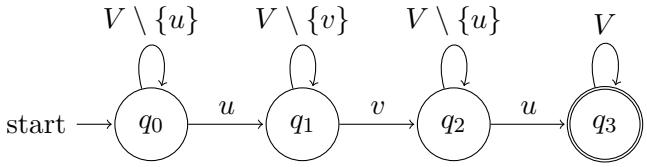


Figure 2.2: A DFA that accepts any sequence containing the subsequence uvu.

Definition 2.5.2. DFA Conjunction: Given a pair of DFAs $M_L = (Q_L, \Sigma_L, \delta_L, q_{0L}, F_L)$ and $M_R = (Q_R, \Sigma_R, \delta_R, q_{0R}, F_R)$, their *conjunction* is a DFA $M_L \wedge M_R = (Q, \Sigma, \delta, q_0, F)$ where:

$$\begin{aligned} Q &= Q_L \times Q_R \\ \Sigma &= \Sigma_L \cap \Sigma_R \\ \delta((q_L, q_R), s) &= (\delta_L(q_L, s), \delta_R(q_R, s)) \\ q_0 &= (q_{0L}, q_{0R}) \\ F &= F_L \times F_R \end{aligned}$$

This is equivalent to running both original automata in parallel, and will accept a sequence if and only if that sequence would be accepted by both M_L and M_R.

Similar constructions are possible for other logical operations. The disjunction of a pair of DFAs can be constructed in the much the same way as their conjunction, but instead we accept any state that would have been accepted by at least one of the original automata $F = F_L \times Q_R \cup Q_L \times F_R$. The negation of a DFA is simply the same automaton with the set of accepting states inverted $F' = Q \setminus F$.

2.5.2 Automaton Structure

As in Section 2.4, by Lemma 2.4.1, in order to determine if a sequence $\mathcal{S}$ is an alternating sequence representation of a graph $G = (V, E)$, it suffices to show that $\forall\{u, v\} \in E : (uvu \mid \mathcal{S}) \wedge (vuv \mid \mathcal{S})$ and conversely that $\forall\{u, v\} \notin E : (uvu \nmid \mathcal{S}) \vee (vuv \nmid \mathcal{S})$. We can construct a DFA $h(u, v)$ to accept any sequence containing the subsequence uvu as

$$h(u, v) = (\{q_0, q_1, q_2, q_3\}, V, \delta, q_0, \{q_3\}) \tag{2.1}$$

where $\delta(q_0, u) = q_1$, $\delta(q_1, v) = q_2$, $\delta(q_2, u) = q_3$, and $\delta(q, s) = q$ otherwise. This automaton is represented graphically in Figure 2.2. For a sequence to contain an alternation of u and v we therefore require that it is accepted by both $h(u, v)$ and $h(v, u)$, and so we can define the conjunction automaton $e(u, v) = h(u, v) \wedge h(v, u)$, which will accept a sequence if and only if it contains an alternation of u and v.

This is not sufficient on its own, however, as this only enforces the existence of edges, and does not disallow the existence of edges that are not in E. To this end, we can construct a DFA $\bar{h}(u, v)$ that accepts any sequence that *does not* contain the alternation $uvuv$:

$$\bar{h}(u, v) = (\{q_0, q_1, q_2, q_3, q_4\}, V, \delta, q_0, \{q_0, q_1, q_2, q_3\}) \tag{2.2}$$

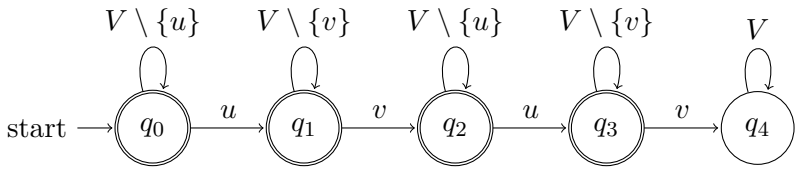


Figure 2.3: A DFA that accepts any sequence that does not contain the subsequence $uvuv$.

where $\delta(q_0, u) = q_1$, $\delta(q_1, v) = q_2$, $\delta(q_2, u) = q_3$, $\delta(q_3, u) = q_4$, and $\delta(q, s) = q$ otherwise. This automaton is represented graphically in Figure 2.3. We can then define the automaton $\bar{e}(u, v) = \bar{h}(u, v) \wedge \bar{h}(v, u)$, which will accept a sequence if and only if it *does not* contain an alternation of u and v.

Alternative constructions exist, but this formulation is convenient since it requires only conjunction to construct. Using these alternation recognition automata, we can now construct a DFA that will accept any sequence that is an alternating sequence for some graph $G = (V, E)$:

$$r(G) = \left(\bigwedge_{\{u,v\} \in E} e(u, v) \right) \wedge \left(\bigwedge_{\{u,v\} \in \overline{E}} \bar{e}(u, v) \right) \tag{2.3}$$

where $\overline{E} = \binom{V}{2} \setminus E$ is the complement of the edge set, as in Definition 1.3.12. A graph G is a polygon-circle graph if and only if there exists a sequence that is accepted by $r(G)$, and which is therefore an alternating sequence of G.

We can find an upper bound on the number of states in the final recognition automaton by multiplying together the number of states in each of its component DFAs. In the worst case, all $|V|^2 - |V|$ components have 5 states, giving an upper bound of $O(5^{|V|^2-|V|})$ states for the final recognition automaton (though we will show we can reduce this bound in Section 2.5.3).

The recognition problem therefore is to determine if there exists any sequence that is accepted by $r(G)$, and the representation problem to construct any such sequence. Given the recognition automaton, both of these can be accomplished by doing a depth-first search of the automaton to find if there is a reachable accepting state from the initial state and the corresponding sequence. Such a search can be done in $O(|V|N)$ time, where N is the number of states in the automaton, since each state has at most $|V|$ outgoing edges. The automaton can also be used to enumerate all alternating sequences of G by performing a backtracking search.

2.5.3 Construction Algorithm

We now aim to construct the recognition automaton (or equivalent) as efficiently as possible.

Definition 2.5.3. DFA Equivalence: A pair of DFAs M_L, M_R are considered *equivalent* if and only if all sequences accepted by M_L are also accepted by M_R, and all sequences accepted by M_R are also accepted by M_L.

We begin by observing that the DFAs defined in Equations (2.1) and (2.2) are *pseudocyclic*, and that pseudocyclic DFAs are closed under conjunction.

Definition 2.5.4. Pseudocyclic DFA: A DFA is *pseudocyclic* if it contains no cycles other than self-loops. More formally, a DFA is pseudocyclic if and only if there is no sequence that will cause it to return to a state it has previously left.

Lemma 2.5.1. *The conjunction of a pair of pseudocyclic DFAs is itself pseudocyclic.*

Proof. Given a pair of pseudocyclic DFAs M_L and M_R, we aim to show that their conjunction $M = M_L \wedge M_R$ is pseudocyclic. This follows intuitively from Definitions 2.5.2 and 2.5.4, since for there to be a transition loop from some state (q_L, q_R) to itself (possibly through intermediate states), there must exist a transition loop from q_L to itself in M_L, and from q_R to itself in M_R. Since both M_L and M_R are pseudocyclic, these loops can only exist as self-loops, meaning that the loop must simply be a direct transition from (q_L, q_R) to itself, as if either the M_L or M_R moved to a different state, they would be unable to return. This is by definition a self-loop, meaning that any loop in M must be a self-loop, and hence M is pseudocyclic. □

It follows that the conjunction of any set of pseudocyclic DFAs is also pseudocyclic, and hence that the recognition automaton from Equation (2.3) is pseudocyclic, as are all intermediate DFAs used in its construction.

If we construct such an automaton naively, we would expect its size to grow exponentially, as the number of states in the conjunction of two DFAs is the product of the numbers of states in each. This gives us a conservative upper bound on the number of states of $O(5^{|V|^2-|V|})$, since in the worst case we may construct a 5-state DFA as in Equation (2.2) for each ordered pair of vertices. This number of states quickly becomes intractably large, even for small graphs. To make this problem more feasible, we aim instead to construct an equivalent but smaller DFA.

To this end we exploit some properties of pseudocyclic DFAs to efficiently compute the minimal equivalent DFA. In general DFAs, this is typically done using Hopcroft's $O(n \log n)$ algorithm [27], but Revuz and Bubenzer give linear-time algorithms for acyclic DFAs [44, 5]. These algorithms can be adapted to work with pseudocyclic DFAs, if self-loops are treated as equivalent to a transition to a behaviourally equivalent state. The optimal construction for $e(u, v)$ given by this minimisation process is shown in Figure 2.4. The optimal construction for $\bar{e}(u, v)$ is simply the complement of the same, as shown in Figure 2.5. Both of these have a total of 8 states, but we only require one per unordered pair of vertices instead of one per ordered pair, giving a new upper bound on number of states in a recognition automaton of $O(8^{(|V|^2-|V|)/2}) \sim O(\sqrt{8}^{|V|^2-|V|})$ even if no further optimisations are possible. While still exponential, this is a substantial improvement over the complexity of the dynamic programming approach presented in Section 2.4.

An implementation of this recognition algorithm that performs minimisation in the same pass as it performs conjunction is available online at `https://github.com/gozzarda/pcg_dfa` [24]. It is worth noting that the order in which the conjunctions are performed can have a significant effect on the size of intermediate DFAs, and hence overall computation time, since it is possible for an intermediate DFA to have many more states than the final result.

We can find a worst-case upper bound on time complexity for this algorithm by assuming we produce an automaton with the maximum number of states ($O(\sqrt{8}^{|V|^2-|V|})$, as above) and by observing that each state may have been involved in a maximum of $|V|^2$ conjunctions, for an overall exponential

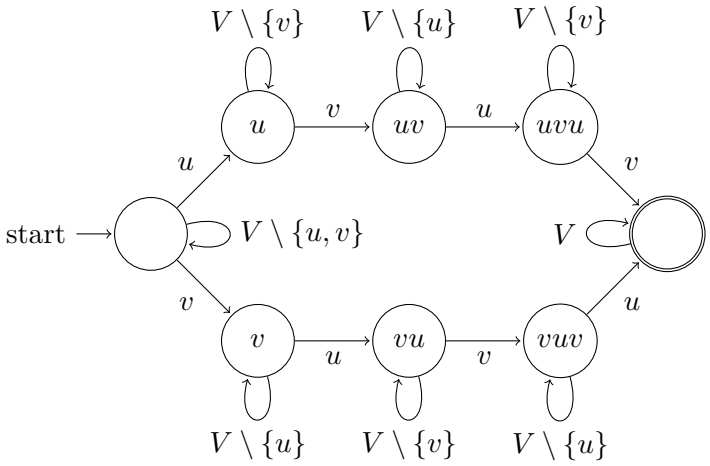


Figure 2.4: A DFA that accepts any sequence containing an alternation of u and v.

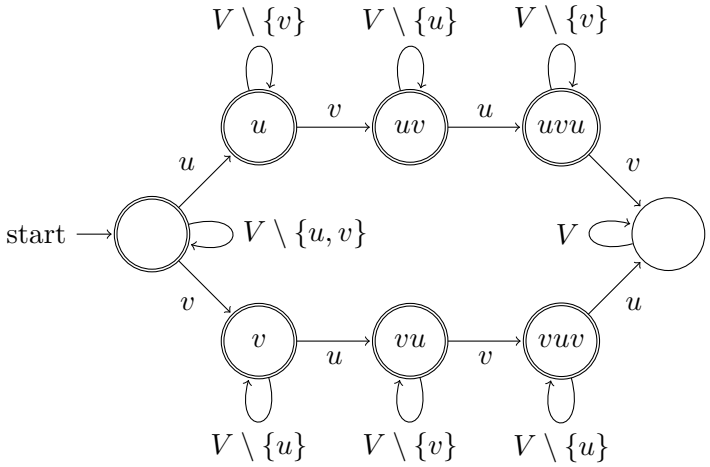


Figure 2.5: A DFA that accepts any sequence that does not contain an alternation of u and v.

upper bound of $O(\sqrt{8}^{|V|^2-|V|}|V|^2)$. Performing the conjunctions in different orders can reduce the $|V|^2$ term to $|V|$ or even $\log(|V|)$, but might come at the cost of drastically increasing the size of intermediate automata. The worst-case memory complexity of this algorithm is dominated by the automata produced, and therefore has an upper bound of $O(\sqrt{8}^{|V|^2-|V|})$. In practice, the computation time for this algorithm is dominated by intermediate automaton size, which is often much less than in the worst case, meaning this algorithm performs far better than the given upper bound on time complexity suggests.

Theorem 2.5.2. *Polygon-circle graphs can be recognised in $O(\sqrt{8}^{|V|^2-|V|}|V|^2)$ time.*

This novel algorithm provides a substantial improvement over the time complexity of the brute-force approach presented in Section 2.3, and both the time and memory complexity of the dynamic programming approach presented in Section 2.4.

2.5.4 Empirical Analysis

To demonstrate its performance in practice, we apply this recognition algorithm to the question of identifying the smallest non-polygon-circle graph.

Theorem 2.5.3. *The smallest non-polygon circle graph is the 3-prism (see Figure 3.1a).*

Proof. By exhaustively enumerating all graphs of up to seven vertices, and running this recognition algorithm on each, we found that all graphs of up to five vertices are polygon-circle graphs, and that the 3-prism is the only 6-vertex graph that is not a polygon-circle graph. □

Furthermore, we found that all 7-vertex non-polygon-circle graphs contain the 3-prism as an induced subgraph. This algorithm was able to find an alternating sequence representation (or that none exists) for all 853 pairwise nonisomorphic connected graphs on 7 vertices in under 85 seconds on an Intel Core i7-6700K processor.

The execution time of this technique is highly data-dependent, and it has proved difficult to find a tight asymptotic bound on runtime. Empirical analysis of the time taken to construct the recognition automata for a random sample of connected graphs and the number of states in these automata shows a strong exponential trend in the size of the graph (Figures 2.6 and 2.7). Note that the outliers in Figure 2.7 will be the few randomly selected graphs that happen to not be polygon-circle graphs and hence the final automaton contains a single state and simply rejects all sequences.

The memory requirements for this algorithm appear to grow rapidly with the density of the graph in question. Note that of course any complete graph can be trivially shown to be a polygon-circle graph without the use of this algorithm. This algorithm handled a number of randomly generated 50% density graphs of up to 12 vertices without exhausting the 32 GiB of RAM in my workstation. By comparison, the dynamic programming approach presented in Section 2.4 would require an estimated 4.7×10^{21} states for even a 6-vertex graph, making its memory usage infeasible for any graph that could possibly not be a polygon-circle graph.

Overall this algorithm represents a significant improvement over the alternative methods presented in Sections 2.3 and 2.4, as it is practical for sizes of graph for which other approaches would require either infeasible amounts of time or infeasible amounts of memory.

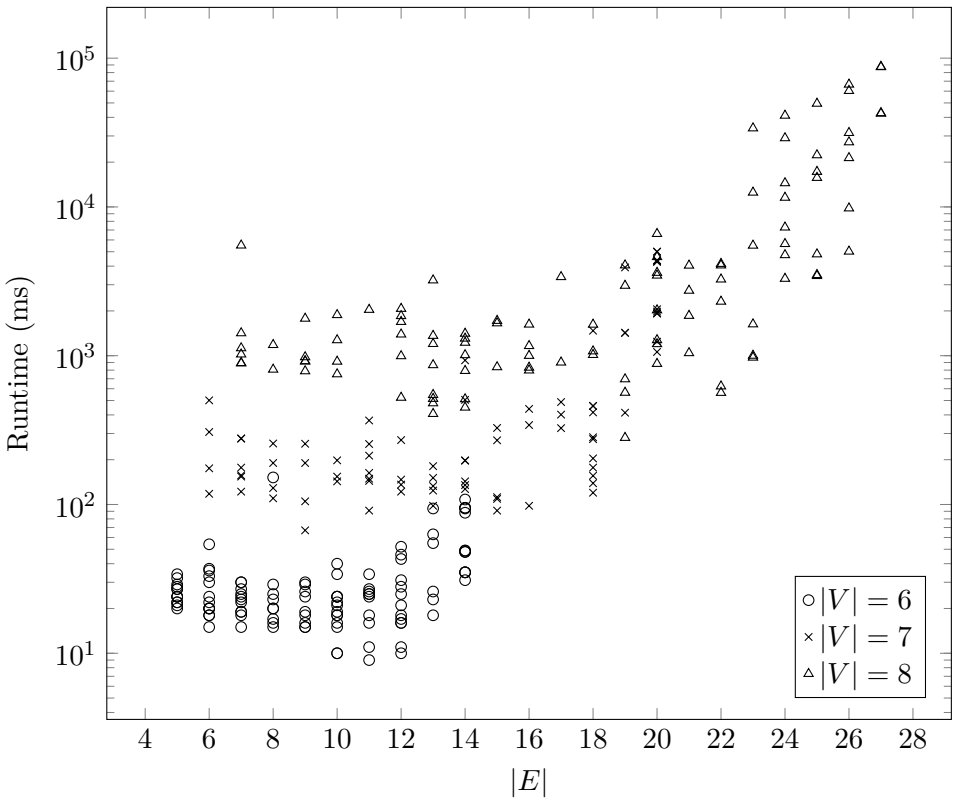


Figure 2.6: Runtime to construct recognition automata for 300 randomly generated connected graphs.

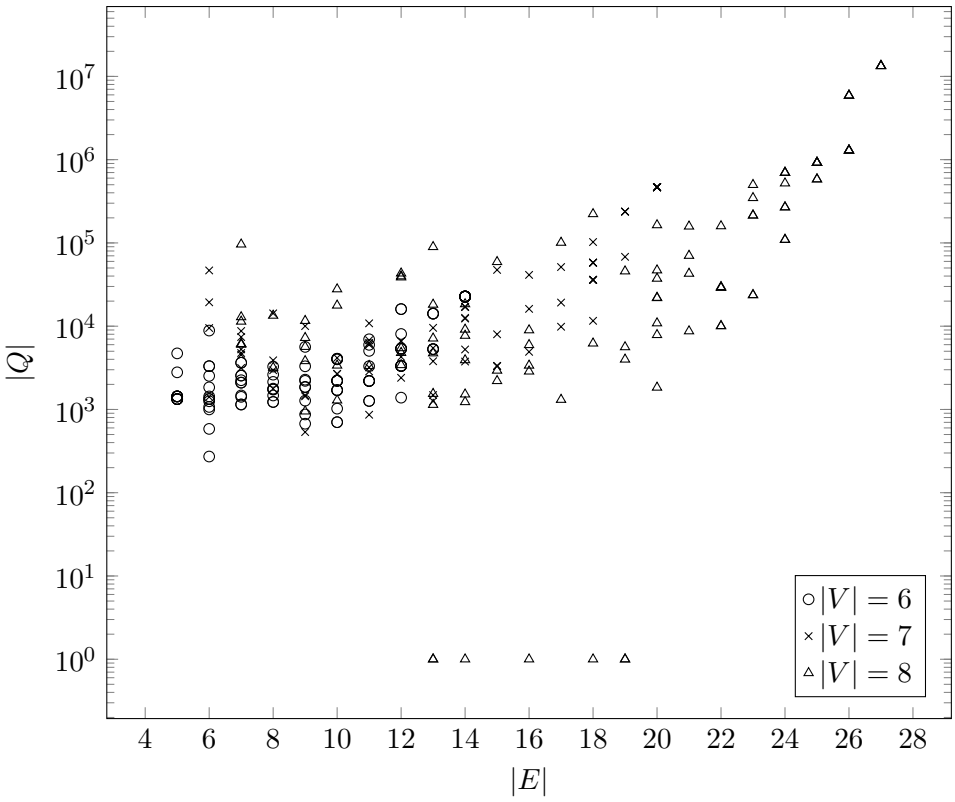


Figure 2.7: Number of states $|Q|$ in recognition automata for 300 randomly generated connected graphs.

Chapter 3

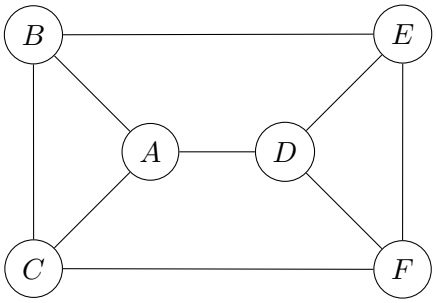
A Constructive Proof that Polygon-Circle Graphs are Not Equivalent to Interval Filament Graphs

3.1 Introduction

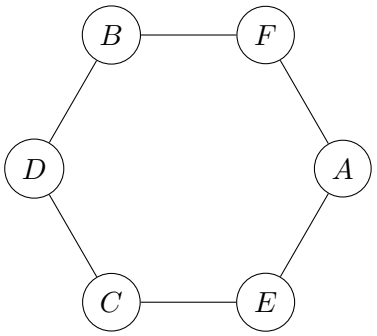
This chapter covers analysis and proof of the subclass relationship (see Section 1.4.3) between polygon-circle graphs (Section 1.3.4) and interval filament graphs (Section 1.3.6). Understanding the relationship between such graph classes is important to solving problems in the various classes and proving the correctness of these solutions.

When Gavril first introduced interval filament graphs, they also provided a straightforward proof that polygon-circle graphs are a subclass of interval filament graphs [19]. To show that polygon-circle graphs are a *proper* subclass of interval filament graphs, they first show that co-comparability graphs are also a subclass of interval filament graphs, and then cite a result from Janson and Kratochvıl [29] that polygon-circle graphs and co-comparability graphs are not equivalent. From this it follows that both polygon-circle graphs and co-comparability graphs must be proper subclasses of interval filament graphs, as there must exist a co-comparability graph that is not a polygon-circle graph, but is an interval filament graph, and there must exist a polygon-circle graph that is not a co-comparability graph, but is an interval filament graph.

However, to show that polygon-circle graphs and co-comparability graphs are not equivalent, Janson and Kratochvıl [29] use a highly involved probabilistic proof method as discussed in Section 1.3.5. This proof method is non-constructive, however, and so does not produce an explicit counterexample to prove that polygon-circle graphs are not equivalent to co-comparability graphs. Following on from the result in Section 2.5.4 that shows by complete search that the 3-prism (see Figure 3.1a) is the minimal non-polygon-circle graph we can develop a constructive proof that this graph is not a polygon-circle graph. Combined with a proof that the 3-prism is a co-comparability graph, this amounts to a proof that polygon-circle graphs and co-comparability graphs are not equivalent. This, therefore, also serves as a proof that the class of polygon-circle graphs is a proper subclass of the class of interval filament graphs.



(a) The 3-prism.



(b) The complement of the 3-prism — the 6-cycle.

Figure 3.1: The 3-prism and its complement.

Definition 3.1.1. Cycle Graph: A *cycle graph* is an undirected graph containing only a single cycle through all vertices and no other edges. More specifically, the k-cycle, denoted C_k, is a cycle graph containing k vertices.

Definition 3.1.2. Prism Graph: A *prism graph* is an undirected graph corresponding to the skeleton of a prism. More specifically, the k-prism, denoted Π_k, is a prism graph containing $2k$ vertices in the form of a pair of k-cycles where there is an edge between corresponding vertices in the two cycles. This can be more formally defined as the graph Cartesian product of the complete graph of order 2, K_2, and the k-cycle, C_k.

3.2 Proof that the 3-prism is a Co-comparability Graph

Lemma 3.2.1. *The complement of the 3-prism* Π_3 *is the 6-cycle* C_6.

Proof. A diagram of the 3-prism $\Pi_3 = (V, E)$ per Definition 3.1.2 is given in Figure 3.1a. By Definition 1.3.12, the complement of this is the graph $\overline{\Pi_3} = (V, \overline{E})$ where $\overline{E} = \binom{V}{2} \setminus E$. Thus the complement of this graph is shown in Figure 3.1b. The complement consists of a single cycle through all 6 vertices and no other edges, and so by Definition 3.1.1 is the 6-cycle $C_6 = \overline{\Pi_3}$. □

Lemma 3.2.2. *The 6-cycle* C_6 *is a comparability graph.*

Proof. Consider the strict partially ordered set (V, R) where:

$$V = \{A, B, C, D, E, F\}$$

$$R = \{(A, E), (A, F), (B, D), (B, F), (C, D), (C, E)\}$$

By Definition 1.3.13, the comparability graph of a strict partially ordered set (V, R) is the graph (V, E) where E is the set of all unordered pairs in V that are comparable using the relation R. For the given strict partially ordered set, this corresponds to the 6-cycle as shown in Figure 3.1b. □

Lemma 3.2.3. *The 3-prism* Π_3 *is a co-comparability graph.*

Proof. By Lemma 3.2.1 and Lemma 3.2.2, the complement of the 3-prism is a comparability graph, the 6-cycle. By Lemma 1.3.1, the 3-prism is therefore a co-comparability graph. □

Alternatively, we can show that the 3-prism is a co-comparability graph by constructing an appropriate intersection representation. Such an intersection representation is given in Figure 1.6.

3.3 A Constructive Proof that the 3-Prism is Not a Polygon-Circle Graph

Definition 3.3.1. Star Graph: A *star graph* is an undirected graph wherein there is a single vertex that has an edge to all other vertices, which each have degree exactly 1. More specifically, the k-star, denoted $K_{1,k-1}$, is a star graph containing k vertices.

Definition 3.3.2. Vertex Cut Set: A *vertex cut set* is a subset of the vertices of a graph which, when removed (along with any incident edges), increases the number of connected components in the graph. In a connected graph, this disconnects the graph.

Definition 3.3.3. Star Cut Set: A *star cut set* is a vertex cut set that has an induced subgraph that is a star graph.

Definition 3.3.4. Articulation Vertex: An *articulation vertex* is any vertex in a graph that forms a vertex cut set by itself.

Definition 3.3.5. Universal Vertex: A *universal vertex* of a graph is a vertex that is adjacent to every other vertex in the graph.

Lemma 3.3.1. *No cycle graph contains an articulation vertex.*

Proof. Removing any single vertex from a cycle graph results in a path graph. As a path graph is still connected, that vertex cannot have been a vertex cut set, and so cannot have been an articulation vertex. □

Lemma 3.3.2. *No prism graph contains a star cut set.*

Proof. First, note that prism graphs have enough symmetry that all vertices are equivalent, so we can consider some arbitrary vertex v and hence need only to show that it cannot be the centre of a star cut set. Each vertex in a prism graph can be considered as belonging to one cycle and having exactly one neighbour in the other cycle. Let us refer to the cycle to which v belongs as the *inner* cycle, and to the other as the *outer* cycle. It follows that a star centred on v may contain at most one vertex in the outer cycle. Since no cycle graph contains an articulation vertex (Lemma 3.3.1), regardless of whether or not the outer vertex is part of the star, the outer cycle will remain connected. Every remaining vertex in the inner cycle will therefore be adjacent to a vertex in the outer cycle, and so the whole graph will remain connected, no matter what other vertices are removed from the inner cycle. Therefore no prism graph contains a star cut set. □

Lemma 3.3.3. *For $k \geq 3$, a prism graph Π_k contains no universal vertices.*

Proof. By Definition 3.1.2, each vertex only has one neighbour in the opposite cycle, and so for $k > 1$ every vertex must have at least one vertex in the opposite cycle to which it is not adjacent, and therefore cannot be a universal vertex. □

Lemma 3.3.4. *(Durán, Grippo, and Safe [13]) The 3-prism $\Pi_3 = \overline{C_6}$ is not a circular-arc graph.* □

Lemma 3.3.5. *(Cameron and Hoàng [7]) A connected polygon-circle graph contains a star cut set or a universal vertex, or else it is a circular-arc graph.* □

Lemma 3.3.6. *The 3-prism Π_3 is not a polygon-circle graph.*

Proof. By the contrapositive of Lemma 3.3.5, any graph that contains neither a star cut set nor a universal vertex and is not a circular-arc graph cannot be a connected polygon-circle graph.

- By Lemma 3.3.2, all prism graphs contain no star cut set.
- By Lemma 3.3.3, all prism graphs $\Pi_{k\geq 3}$ contain no universal vertex.
- By Lemma 3.3.4, the 3-prism Π_3 is not a circular-arc graph.

Since the 3-prism Π_3 is connected, it follows that it must not be a polygon-circle graph. □

3.4 Conclusion

Lemma 3.4.1. *(Gavril [19]) The class of polygon-circle graphs is a subclass of the class of interval filament graphs.* □

Lemma 3.4.2. *(Gavril [19]) The class of co-comparability graphs is a subclass of the class of interval filament graphs.* □

Theorem 3.4.3. *The class of polygon-circle graphs is a proper subclass of the class of interval filament graphs.*

Proof. Consider the 3-prism Π_3. By Lemma 3.2.3, this graph is a co-comparability graph. By Lemma 3.4.2, it is also therefore an interval filament graph. However, by Lemma 3.3.6, it is *not* a polygon-circle graph. As such we have an explicit example of a graph that is an interval filament graph but not a polygon circle graph, and so these classes cannot be equivalent. By Lemma 3.4.1, the class of polygon-circle graphs is a subclass of the class of interval filament graphs. Therefore the class of polygon-circle graphs must be a proper subclass of the class of interval filament graphs. □

To my knowledge, this marks the first constructive proof that polygon-circle graphs are a proper subclass of interval filament graphs. With further work it may be possible to generalise this result to a larger class of graphs that are interval filament graphs but not polygon circle graphs. This result could have applications in finding excluded subgraphs of polygon-circle graphs and other classes.

Chapter 4

Average Path Length Minimisation by Shortcut Edge Addition in Circular-Arc Graphs

4.1 Introduction

This chapter covers the average path length minimisation through shortcut edge addition problem (MinAPL, see Section 1.4.5) in weighted circular-arc graphs (see Section 1.3.2 and Definition 1.4.3). Any solution to this problem generalises to interval graphs. Achieving low average path lengths using a limited number of edges is closely related to a property of graphs called the *small-world* phenomenon, first observed by Milgram [41]. Small-world networks are characterised by having an average path length that grows logarithmically with the size of the graph while also having a high clustering coefficient [52]. This is a property that occurs frequently in nature but is uncommon in structured graphs. Selecting additional edges to minimise the average path length while keeping the graph small can convert a given graph into a small-world network. We will first discuss solutions for general graphs and then specialise to circular-arc graphs to achieve better performance in some cases.

4.2 Distance Matrix Update

A straightforward solution for general graphs proposed by Ward and Datta [49] computes the all-pairs shortest path lengths and hence the total and average path length, and then for each edge in the set of candidate edges S computes the change in these values as a result of inserting that edge and simply keeps track of the candidate edge that gave the smallest average path length. We can initialise the distance matrix with any appropriate all-pairs shortest paths algorithm, for example the classic $O(|V|^3)$ Floyd-Warshall algorithm [14]. For any single candidate edge $s = \{u, v\} \in S$ with weight $w(s)$ the distance matrix D can be updated using:

$$D'_{i,j} = \min(D_{i,j}, D_{i,u} + w(s) + D_{v,j}, D_{i,v} + w(s) + D_{u,j}) \tag{4.1}$$

Applying this update for every pair of vertices takes $O(|V|^2)$ time. As we know the change in each element of the distance matrix, we can compute the change in the sum of all shortest path lengths, and hence the change in average path length. The greatest total decrease will correspond to the minimum possible average path length. Doing this for each candidate edge takes $O(|S||V|^2)$ time to find the best single candidate edge, giving an overall complexity of $O(|V|^3 + |S||V|^2)$ if we use Floyd-Warshall to find the original distance matrix. This approach is presented in Algorithm 1. This algorithm is also applicable to directed graphs with minor modifications [25].

Algorithm 1 (Ward and Datta [49]) Finding the candidate edge that minimises average path length in a general graph by updating distance matrix.

```
1: function DistanceMatrixUpdateMinAPL(G, w, S)
2:     (V, E) ← G
3:     D ← APSP(G, w)
4:     Δ ← ∞
5:     e ← ∅
6:     for s ∈ S | s ∉ E do
7:         {u, v} ← s
8:         δ ← 0
9:         for i ∈ V do
10:            for j ∈ V do
11:                D'_{i,j} ← min(D_{i,j}, D_{i,u} + w(s) + D_{v,j}, D_{i,v} + w(s) + D_{u,j})
12:                δ ← δ + D'_{i,j} − D_{i,j}
13:            end for
14:        end for
15:        if δ < Δ then
16:            Δ ← δ
17:            e ← {s}
18:        end if
19:    end for
20:    return e
21: end function
```

4.3 Threshold Algorithm

The threshold algorithm is based on the same underlying logic as the distance matrix update algorithm described in Section 4.2, and affords significant improvements in performance for large $|S|$. The threshold algorithm is based on the observation that as the weight of the inserted edge decreases, for each pair of vertices there is some threshold value at which that edge will become part of the shortest path, and after that point for every incremental decrease in the weight of the inserted edge, the shortest path between those vertices will decrease by the same amount. We will first present the directed variant of the threshold algorithm and then adapt it to support undirected graphs.

In a directed graph, for any single candidate edge $s = (u, v)$ with weight $w(s)$, the distance matrix D can be updated using:

$$D'_{i,j} = \min(D_{i,j}, D_{i,u} + w(s) + D_{v,j}) \tag{4.2}$$

Therefore, the change in path length $\delta_{i,j}$ as a result of adding e is:

$$\begin{aligned}\delta_{i,j} &= D'_{i,j} - D_{i,j} && (4.3)\\ &= \min(D_{i,j}, D_{i,u} + w(s) + D_{v,j}) - D_{i,j} && (4.4)\\ &= \min(0, D_{i,u} + w(s) + D_{v,j} - D_{i,j}) && (4.5)\end{aligned}$$

which has piecewise definition:

$$\delta_{i,j} = \begin{cases} D_{i,u} + w(s) + D_{v,j} - D_{i,j} & \text{if } D_{i,u} + w(s) + D_{v,j} - D_{i,j} < 0, \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

Note that for this formulation to be well defined, $D_{i,u} + w(s) + D_{v,j}$ and $D_{i,j}$ must not both be infinite. This precludes an infinite disconnection cost, but the same results are achieved by selecting a sufficiently large finite disconnection cost to be greater than all other values that will be compared to it. If we define a threshold value $T_{i,v,j}$, Equation (4.6) can be rearranged as follows:

$$T_{i,v,j} = D_{i,j} - D_{v,j} \tag{4.7}$$

$$\delta_{i,j} = \begin{cases} D_{i,u} + w(s) - T_{i,v,j} & \text{if } D_{i,u} + w(s) < T_{i,v,j}, \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Figure 4.1 gives a graphical representation of the threshold value. The change in the total sum of shortest path lengths as a result of inserting s is:

$$\delta = \sum_{i \in V} \sum_{j \in V} \delta_{i,j} \tag{4.9}$$

Observe that only values of i and j such that $D_{i,u} + w(s) < T_{i,v,j}$ contribute to the sum in Equation (4.9), and that the left-hand side of this condition is independent of j. Therefore the subset $V'_i \subseteq V$ can be defined and substituted into Equation (4.9) as follows:

$$V'_i = \{j \in V \mid D_{i,u} + w(s) < T_{i,v,j}\} \tag{4.10}$$

$$\begin{aligned}\delta &= \sum_{i \in V} \sum_{j \in V'_i} \delta_{i,j} \\ &= \sum_{i \in V} \sum_{j \in V'_i} (D_{i,u} + w(s) - T_{i,v,j}) \\ &= \sum_{i \in V} \left(|V'_i| \times (D_{i,u} + w(s)) - \sum_{j \in V'_i} T_{i,v,j} \right)\end{aligned} \tag{4.11}$$

If the elements of $\{T_{i,v,j} \mid j \in V\}$ are sorted by ascending value, the elements of $\{T_{i,v,j} \mid j \in V'_i\}$ form a contiguous suffix of the sorted list. A binary search on the sorted elements gives the bounds of this suffix. A pair of cumulative sum arrays can then be used to compute the two sums over V'_i in Equation (4.11). A cumulative sum array is a simple data structure that allows the sum of a

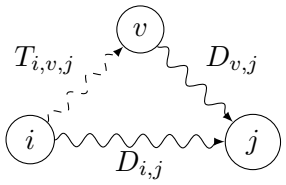


Figure 4.1: Shown are the shortest paths from i to j and from v to j with lengths $D_{i,j}$ and $D_{v,j}$ respectively. The threshold value $T_{i,v,j} = D_{i,j} - D_{v,j}$ is the required length of the hypothetical shortest path from i to v such that the shortest path from i to j through v has length $D_{i,j}$.

contiguous range of elements in a list to be computed in constant time. Let C be a sum array for some given numeric list. The n-indexed element C_n of the sum array is the sum of the first n elements of the original list. The sum of all elements in the index range $[i, j)$ is then $C_j - C_i$. The cumulative sum array for any given list can be computed in linear time in the size of the input list. Algorithm 2 shows a method for solving MinAPL using this result.

The function $\textsc{Sort}(V, T_{i,j})$ on Line 12 of Algorithm 2 sorts the elements $j \in V$ in ascending order according to the corresponding value of $T_{i,v,j}$. This gives an ordering $J_{i,v}$ of V for which $T_{i,v,j}$ increases monotonically, which can be used to compute the cumulative sum array W_v, as well as a monotonically increasing ordering $T'_{i,v}$ of $T_{i,v}$. The function $\textsc{BinarySearchUpperBound}(T'_{i,v}, L)$ on Line 24 performs a binary search over the ordering $T'_{i,v}$ and returns an index l, the minimum 0-based index for which $T'_{i,v,l} > L$, or $|T'_{i,v}| = |V|$ if no such l exists. This index is then used in conjunction with the aforementioned cumulative sum array to compute a partial sum from Equation (4.11) for a particular i and s. By looping over all $i \in V$ and all candidate edges $s \in S$ it is possible to compute δ_s — the change in sum of shortest path lengths as a result of adding edge s — and s can therefore be selected to minimise the sum of, and hence the average path length in $G' = (V, E \cup \{s\})$.

Algorithm 2 has a time complexity of $O(apsp(G, w) + |V|^3 \log|V| + |V||S| \log|V|)$, where $apsp(G, w)$ is the run time of the all-pairs shortest path algorithm used. Even when using the classic Floyd-Warhsall algorithm, the complexity is dominated by the other terms, giving an overall complexity of $O(|V|^3 \log|V| + |V||S| \log|V|)$. For large $|S|$ (i.e., $|S| \in \Omega(|V|^2)$), this is a significant improvement in performance over the distance matrix update algorithm (see Algorithm 1 and Section 4.2), giving run time $O(|V|^3 \log|V|)$ as opposed to $O(|V|^4)$. For small $|S|$, however, the distance matrix update algorithm will outperform this algorithm. Memory complexity remains $O(|V|^2)$ in all cases, which is the same as the distance matrix update algorithm.

This approach can be adapted to undirected graphs by simply combining the contributions of the edge in each direction, because we can know that no shortest path will go through the edge in both directions. The modified version of this algorithm is presented in Algorithm 3.

Algorithm 2 Finding the candidate edge that minimises average path length in a general graph using the threshold algorithm.

function ThresholdMinAPL(G, w, S)
  $(V, E) \leftarrow G$
  $D \leftarrow \text{APSP}(G, w)$ ▷ Compute the all-pairs shortest path lengths
  for $s \in S$ **do**
    $\delta_s \leftarrow 0$ ▷ Initialise change in sum of path lengths for each edge
  end for
  for $i \in V$ **do**
    for $v \in V$ **do**
      for $j \in V$ **do**
        $T_{i,v,j} \leftarrow D_{i,j} - D_{v,j}$ ▷ Compute thresholds as per Equation (4.7)
      end for
      $J_{i,v} \leftarrow \text{Sort}(V, T_{i,j})$ ▷ Sort $j \in V$ by $T_{i,v,j}$
      $W_{v,0} \leftarrow 0$ ▷ Cumulative sum of $T_{i,v}$ in sorted order
      for $l \in [0, |V|)$ **do**
        $j \leftarrow J_{i,v,l}$
        $T'_{i,v,l} \leftarrow T_{i,v,j}$ ▷ Sort T
        $W_{v,l+1} \leftarrow W_{v,l} + T_{i,v,j}$
      end for
    end for
    for $s \in S$ **do**
      $(u, v) \leftarrow s$
      if $w(s) + D_{v,u} \geq 0$ **then** ▷ Disregard negative-weight cycles
        $L \leftarrow D_{i,u} + w(s)$ ▷ Length of path from i to v through s
        $l \leftarrow \text{BinarySearchUpperBound}(T'_{i,v}, L)$ ▷ Find lowest threshold that is met
        $\delta_s \leftarrow \delta_s + (|V| - l) \times L - (W_{v,|V|} - W_{v,l})$ ▷ As per Equation (4.11)
      end if
    end for
  end for
  $\Delta \leftarrow 0$ ▷ Trivial solution
  $e \leftarrow \varnothing$
  for $s \in S$ **do**
    if $\delta_s < \Delta$ **then**
      $\Delta \leftarrow \delta_s$
      $e \leftarrow \{s\}$
    end if
  end for
  return e
end function

Algorithm 3 Finding the candidate edge that minimises average path length in a general graph using the threshold algorithm.

function UndirectedThresholdMinAPL(G, w, S)
  $(V, E) \leftarrow G$
  $D \leftarrow \text{APSP}(G, w)$ ▷ Compute the all-pairs shortest path lengths
  for $s \in S$ **do**
    $\delta_s \leftarrow 0$ ▷ Initialise change in sum of path lengths for each edge
  end for
  for $i \in V$ **do**
    for $v \in V$ **do**
      for $j \in V$ **do**
        $T_{i,v,j} \leftarrow D_{i,j} - D_{v,j}$ ▷ Compute thresholds as per Equation (4.7)
      end for
      $J_{i,v} \leftarrow \text{Sort}(V, T_{i,j})$ ▷ Sort $j \in V$ by $T_{i,v,j}$
      $W_{v,0} \leftarrow 0$ ▷ Cumulative sum of $T_{i,v}$ in sorted order
      for $l \in [0, |V|)$ **do**
        $j \leftarrow J_{i,v,l}$
        $T'_{i,v,l} \leftarrow T_{i,v,j}$ ▷ Sort T
        $W_{v,l+1} \leftarrow W_{v,l} + T_{i,v,j}$
      end for
    end for
    for $s \in S$ **do**
      $\{u, v\} \leftarrow s$
      if $w(s) \geq 0$ **then** ▷ Disregard negative-weight cycles
        $L \leftarrow D_{i,u} + w(s)$ ▷ Length of path from i to v through s
        $l \leftarrow \text{BinarySearchUpperBound}(T'_{i,v}, L)$
        $\delta_s \leftarrow \delta_s + (|V| - l) \times L - (W_{v,|V|} - W_{v,l})$
        $L' \leftarrow D_{i,v} + w(s)$ ▷ Length of path from i to u through s
        $l' \leftarrow \text{BinarySearchUpperBound}(T'_{i,u}, L')$
        $\delta_s \leftarrow \delta_s + (|V| - l') \times L' - (W_{u,|V|} - W_{u,l'})$
      end if
    end for
  end for
  $\Delta \leftarrow 0$ ▷ Trivial solution
  $e \leftarrow \varnothing$
  for $s \in S$ **do**
    if $\delta_s < \Delta$ **then**
      $\Delta \leftarrow \delta_s$
      $e \leftarrow \{s\}$
    end if
  end for
  return e
end function

4.4 Repeated All-Pairs Shortest Paths in Circular-Arc Graphs

Circular-arc graphs have various properties that do not hold for general graphs, but the algorithms presented in Sections 4.2 and 4.3 do not make use of these properties. Of particular relevance to the MinAPL problem is that there exists an $O(|V|^2)$ all-pairs shortest path algorithm for circular-arc graphs due to Saha, Pal, and Pal [45]. Given the all-pairs shortest path lengths for a graph, it is simple to compute the average of these lengths in time proportional to the number of paths being averaged, $O(|V|^2)$. This means that we can compute the average path length in the graph after adding a particular candidate edge in $O(|V|^2)$ time. We then need simply perform this process for each candidate edge, keeping track of the best average path length for any candidate edge (see Algorithm 4).

Algorithm 4 Finding the candidate edge that minimises average path length in a graph by repeated all-pairs shortest path.

```
1:  function RepeatedShortestPathsMinAPL(G, w, S)
2:      (V, E) ← G
3:      D ← APSP(G, w)
4:      Δ ← ∞
5:      e ← ∅
6:      for s ∈ S | s ∉ E do
7:          δ ← 0
8:          E′ ← E ∪ {s}
9:          G′ ← (V, E′)
10:         D′ ← APSP(G′, w)
11:         for i ∈ V do
12:             for j ∈ V do
13:                 δ ← δ + D′_{i,j} − D_{i,j}
14:             end for
15:         end for
16:         if δ < Δ then
17:             Δ ← δ
18:             e ← {s}
19:         end if
20:     end for
21:     return e
22: end function
```

This algorithm performs this $O(|V|^2)$ process once for each of the $O(|S|)$ candidate edges, giving an $O(|S||V|^2)$ MinAPL algorithm for circular-arc graphs. In a general graph, this method not be competitive with the algorithms presented in Sections 4.2 and 4.3, but for classes of graphs for which these exist more efficient all-pairs shortest paths algorithms, such as circular-arc graphs, this gives a specialised MinAPL algorithm that may be more efficient in some cases.

4.5 Comparison

While it does not work in general graphs, the $O(|S||V|^2)$ repeated all-pairs shortest path algorithm presented in Section 4.4 strictly outperforms the $O(|V|^3 + |S||V|^2)$ distance matrix update algorithm

presented in Section 4.2 in circular-arc graphs, where we can exploit more efficient specialised algorithms for finding the all-pairs shortest paths lengths. The repeated all-pairs shortest path algorithm also outperforms the threshold algorithm presented in Section 4.3 for small $|S| \in O(|V| \log|V|)$, but is worse for larger $|S|$.

Theorem 4.5.1. *The MinAPL problem in circular-arc graphs can be solved either in* $O(|V|^3 \log|V| + |V||S| \log|V|)$ *or in* $O(|S||V|^2)$ *time.*

Thus, depending on the density of the candidate edge set, the repeated all-pairs shortest paths approach and the threshold algorithm jointly represent the best solutions to MinAPL in circular-arc graphs of which I am aware.

Chapter 5

A Greedy Algorithm for Maximum Internal Spanning Tree in Interval Graphs

5.1 Introduction

This chapter examines the MIST problem (see Section 1.4.6) in interval graphs (see Section 1.3.1). This problem is the subject of a recent work by Li, Feng, Jiang, and Zhu [38]. Section 5.2 discusses their solution, and presents a counterexample for which their algorithm does not work. Section 5.3 describes the closely related Hamiltonian path problem (see Definition 1.4.11), and a known greedy algorithm for finding Hamiltonian paths in interval graphs. Section 5.4 describes a novel $O(|I|^2 + |E| \log |I|)$ algorithm for maximum internal spanning tree in interval graphs based on extending this greedy approach. The greedy algorithms take as input an interval intersection representation of the interval graph, but this can be constructed for any interval graph in linear time using an algorithm by Corneil, Olariu, and Stewart [11], and so is not a dominating factor in the complexity of these algorithms.

A reviewer has pointed out that the algorithm presented in Section 5.4 is very similar to one presented by Li, Shang, and Shi [37]. I developed this algorithm independently in early 2021, prior to the publication of their result in July of 2022, but had not planned to publish it until after this thesis. Though this is no longer a unique result, it is still my own original work, and so I have left this chapter as I originally intended to publish it.

5.2 Li *et al.* Interval MIST Algorithm

Li, Feng, Jiang, and Zhu [38] present a novel algorithm (see Algorithm 5) for solving the interval MIST problem for a set of n intervals in $O(n^2)$ time. This algorithm is based on computing a maximum path cover P^* of the intersection graph of the intervals and connecting these paths to construct a spanning tree. A maximum path cover is a set of paths that cover all vertices in the graph using the maximum number of edges (or equivalently the minimum number of paths). The total number of edges in all

the paths of the path cover is given by $E(P^*)$. Their proof of correctness for this algorithm depends on the following propositions:

> [38, Lemma 5]: The number of internal vertices of a maximum internal spanning tree is less than the number of edges of a maximum path cover in a graph.
>
> [38, Lemma 8]: Let P^* be a maximum path cover of an interval graph G. Then [Li *et al.*'s algorithm] returns a spanning tree with the number of internal vertices equal to $|E(P^*)|-1$.
>
> [38, Theorem 1]: [Li *et al.*'s algorithm] can find a maximum internal spanning tree for an interval graph with time complexity $O(n^2)$, where n is the number of vertices in the interval graph.

Li *et al.*'s proof for Theorem 1 begins:

> By [38, Lemma 8] and [38, Lemma 5], [Li *et al.*'s algorithm] returns a spanning tree with the number of internal vertices equal to the upper bound of the number of internal vertices in a spanning tree. So [Li *et al.*'s algorithm] can find a maximum internal spanning tree on an interval graph.

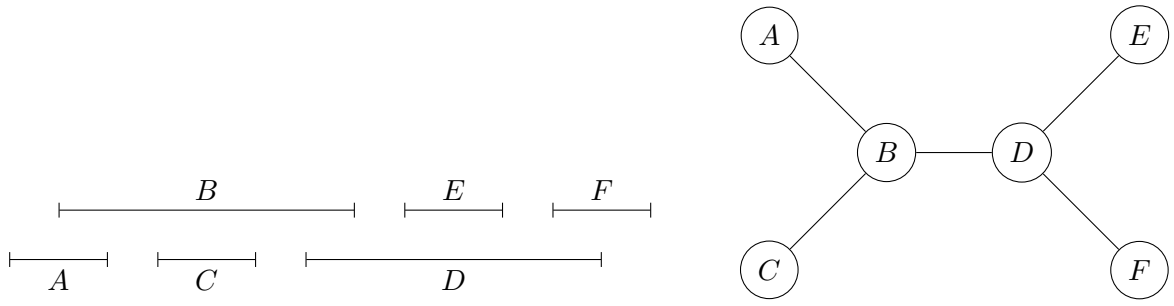
We can show that at least one of these results must be incorrect by constructing a counterexample for which these results do not hold. Such a counterexample is given in Figure 5.1.

Theorem 5.2.1. *[38, Theorem 1] does not hold for all interval graphs.*

Proof. Consider the interval graph G presented in Figure 5.1. G is already a tree, and so has a unique spanning tree, G itself. G contains 2 internal vertices: B and D. G has a maximum path cover P^* of 2 paths with a total of $E(P^*) = 4$ edges: $\langle A, B, C\rangle$ and $\langle E, D, F\rangle$. By [38, Lemma 8], running Li *et al.*'s algorithm on G would return a spanning tree with $|E(P^*)| - 1 = 3$ internal vertices. However, we previously showed that G is its own spanning tree and contains just 2 internal vertices, so [38, Lemma 8] cannot possibly have produced a spanning tree with 3 internal vertices, and so must not hold for this interval graph. Therefore Li *et al.*'s algorithm was not able to find a MIST for this interval graph, and so [38, Theorem 1] cannot hold for all interval graphs. □

An implementation of Algorithm 5 is available online at `https://github.com/gozzarda/interval_mist` [23]. To the best of my knowledge Li *et al.* did not provide an implementation of this algorithm, and my implementation is the first. This result was discovered by testing this implementation against a sample of randomly generated interval graphs. The algorithm failed on the graph shown in Figure 5.1 and similar structures, as it reached a state in which it assumes the existence of a vertex with a particular property based on the following lemma:

> [38, Lemma 3]: Let G_1, G_2 be two connected subgraphs of an interval graph G. If G_1 and G_2 are intersecting and [the leftmost right endpoint in G_1] < [the leftmost right endpoint in G_2], then there exists another vertex $w \in V(G_1)$, such that w and [the vertex with leftmost right endpoint in G_2] are connected by an edge of G.



(a) Interval set. (b) Intersection graph of the intervals.

Figure 5.1: An interval graph with maximum path cover of four edges and maximum internal spanning tree of two interval vertices.

Algorithm 5 (Li, Feng, Jiang, and Zhu [38]) Finding a maximum internal spanning tree on an interval graph.

Require: An interval graph G which has already been right-end ordered.
Ensure: A maximum internal spanning tree of G

1: Find a maximum path cover P^* of G, which can be done in linear time using an algorithm by Arikati and Rangan [1].
2: $T_c \leftarrow \{p \mid p \text{ is a path component of } P^*\}$, $P_C \leftarrow P^* \setminus \{p\}$
3: **while** P_C is not empty **do**
4: Choose a path component q from P_C, where q is intersecting T_C.
5: **if** $leftMost(q) < leftMost(T_C)$ **then**
6: By [**Lemma 3**], choose a vertex $w \in V(q)$ which is adjacent to $v_{leftMost(T_C)}$. Let T_C be the resultant new tree by adding the edge between w and $v_{leftMost(T_C)}$.
7: **end if**
8: **if** $leftMost(q) > leftMost(T_C)$ **then**
9: By [**Lemma 3**], choose a vertex $w \in V(T_C)$ which is adjacent to $v_{leftMost(q)}$. Let T_C be the resultant new tree by adding the edge between w and $v_{leftMost(q)}$.
10: **end if**
11: $P_C \leftarrow P_C \setminus \{q\}$
12: **end while**
13: **return** T_C.

The presented proof of this lemma depends on partitioning G_1 into two vertex subsets based on which side of the leftmost right endpoint in G_2 each vertex's right endpoint lands, and arguing that since the two subsets must be connected there must be a vertex that intersects with the leftmost right endpoint interval in G_2. This proof fails to consider the possibility that all vertices in G_1 are wholly to the left of the leftmost right endpoint interval in G_2. The algorithm uses this result to justify connecting each path to the spanning tree via the leftmost right endpoint interval of one component, but due to this oversight such an edge does not always exist. In the case of the given counterexample, the algorithm finds the maximum path cover given above and attempts to join them, assuming the existence of an edge between E (the leftmost right endpoint interval of the $\langle E, D, F\rangle$ path) to some vertex in the other path, yet no such edge exists. The algorithm does not handle this contradiction and so halts with an error.

This counterexample shows the existence of an error in the proof of correctness presented in Li, Feng, Jiang, and Zhu [38]. As a result their algorithm cannot be assumed to be correct for any given interval graph.

5.3 Hamiltonian Path in Interval Graphs

As mentioned in Section 1.4.6, the MIST problem is a generalisation of the Hamiltonian path problem, in that if a Hamiltonian path exists, it is also a valid MIST, but for any graph where the MIST is not a path, no Hamiltonian path can exist. It follows that the MIST problem must be NP-hard in general graphs, as is the Hamiltonian path problem. In interval graphs, however, there are a number of problems that can be solved in polynomial time despite being NP-complete in general graphs. For example, there exists an optimal $O(n)$ algorithm by Chang, Peng, and Liaw [9] for Hamiltonian path in the intersection graph of a set of n intervals for which the sorted order of interval endpoints is known. We aim to generalise one of these algorithms to construct a greedy algorithm for the MIST problem in interval graphs. For this and later sections in this chapter and unless otherwise specified, intervals are considered to have the same ordering as the ordering of their right endpoints.

Of particular interest for our purposes is a greedy algorithm that can find a Hamiltonian path for the intersection graph of a set of n intervals in $O(n \log n)$ [39]. Note that this algorithm as originally published orders intervals by their left endpoints, not right, but the algorithm is presented mirrored here for consistency (see Algorithm 6). This algorithm operates by greedily building a path through the intersection graph of the intervals until it has either found a Hamiltonian path or is unable to continue, in which case no Hamiltonian path can exist. It begins with a trivial path consisting of just the single interval with leftmost right endpoint. It then incrementally extends this path by greedily picking the leftmost of the remaining intervals that is adjacent to the previous interval, and connecting the chosen interval to the previous interval. If this algorithm ever encounters a state in which there are remaining intervals that are not yet part of the path, but none of them are adjacent to the previous interval, this means that no Hamiltonian path exists for the intersection graph of the given intervals. If the algorithm successfully constructs a path that covers all intervals, this is of course a Hamiltonian path. In either case, the algorithm returns the path P it constructed and a set of any remaining intervals S.

Algorithm 6 Attempt to greedily build a Hamiltonian path for a set of intervals I [39]

```
1: function FindPath(I)
2:     p ← min(I)                          ▷ p holds the last vertex added to the path
3:     S ← I \ {p}                         ▷ The set of vertices not yet in the path
4:     V_P ← {p}
5:     E_P ← ∅
6:     while ∃ s ∈ S : s ⋣ p. do           ▷ There exists a vertex that can be appended to the path
7:         t ← min(s ∈ S | s ⋣ p)          ▷ Greedily choose the leftmost neighbour to append
8:         S ← S \ {t}
9:         V_P ← V_P ∪ {t}
10:        E_P ← E_P ∪ {p, t}
11:        p ← t
12:    end while
13:    P ← (V_P, E_P)
14:    return (P, S)
15: end function
```

Lemma 5.3.1. *(Manacher, Mankus, and Smith [39, Theorem (Path HP)]) For any set of intervals I with intersection graph G, $(P, S) = \textsc{FindPath}(I)$ will return a Hamiltonian path P in G such that $S = \varnothing$ if and only if G has a Hamiltonian path.* □

The path constructed by Algorithm 6 can be shown to have some interesting properties. These results will prove useful in Section 5.4.

To show that the leftmost interval in I is guaranteed to be a leaf in the greedy path P, we need simply observe that after selecting its successor in the path, it will never again have an edge connected to it.

Lemma 5.3.2. *Let I be a set of intervals. Let $(P, S) = \textsc{FindPath}(I)$. The leftmost interval* $\min(I)$ *is a leaf in P.*

Proof. $\textsc{FindPath}(I)$ begins by picking $\min(I)$ as the first vertex in P, and continues by strictly appending other vertices to the path, only ever adding an edge between the immediate previous vertex, p, and the next chosen vertex, t. Therefore at most one edge may ever have been connected to $\min(I)$, and so $\min(I)$ must be a leaf in P. □

As the path is constructed, Algorithm 6 maintains a set S of the remaining intervals, which is simply all intervals that have not yet been added to the path.

Lemma 5.3.3. *Let I be a set of intervals. In the execution of $\textsc{FindPath}(I)$, it is a loop invariant that $S = I \setminus V_P$.*

Proof. S is initially defined to be exactly this prior to the start of the loop. For all subsequent iterations of the loop, the same vertex, t, is added to V_P and deleted from S, and no other changes are made to either. Thus by induction $S = I \setminus V_P$ at the beginning and end of every iteration of the loop in $\textsc{FindPath}(I)$. □

Every edge that appears in the path is the result of the former interval selecting the latter as its leftmost remaining neighbour (that is, that does not appear in the path before it).

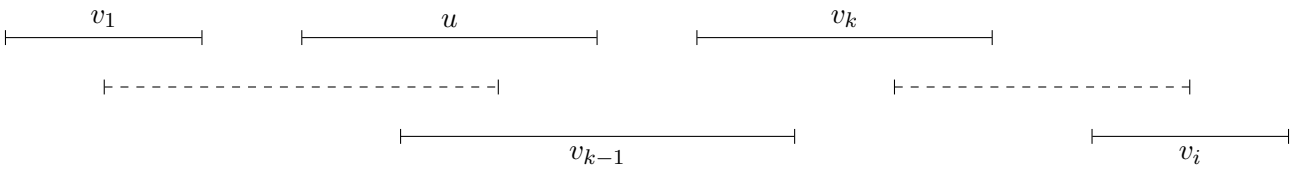


Figure 5.2: An example arrangement of intervals showing that if v_i is wholly right of u there must be some pair of intervals v_{k-1}, v_k such that v_k is wholly right of u and $v_{k-1} \not\ni u$. Dashed intervals represent the existence of a connected path of intervals.

Lemma 5.3.4. *Let I be a set of intervals. Let $(P', S') = \textsc{FindPath}(I)$. Let $P' = \langle v_1, \ldots, v_n \rangle$ such that $v_1 = \min(I)$. For any edge $\{v_k, v_{k+1}\}$ in P', it follows that $v_{k+1} = \min(s \in I \setminus \{v_1, \ldots, v_k\} \mid s \not\ni v_k)$, the leftmost remaining interval that intersects v_k.*

Proof. Consider the execution of $\textsc{FindPath}(I)$ up to the point where it has added v_k to V_P. At the beginning of the next iteration of the loop, $p = v_k$. By Lemma 5.3.3, $S = I \setminus V_P = I \setminus \{v_1, \ldots, v_k\}$. Therefore v_{k+1} must have been selected by $t \leftarrow \min(s \in S \mid s \not\ni p)$, and so $v_{k+1} = \min(s \in I \setminus \{v_1, \ldots, v_k\} \mid s \not\ni v_k)$, as desired. □

This means that any interval that is selected to be added to the path may not be wholly to the right of any remaining intervals, as for any other interval wholly to its left, there should have been an earlier interval in the path that would have selected that as its successor.

Lemma 5.3.5. *Let I be a set of intervals. Let $(P, S) = \textsc{FindPath}(I)$. Let $P = \langle v_1, \ldots, v_n \rangle$. For any index i, no interval in $I \setminus \{v_1, \ldots, v_i\}$ may be wholly left of v_i.*

Proof. We will show this by contradiction. Let i be some index and $u \in I \setminus \{v_1, \ldots, v_i\}$ some interval such that u is wholly left of v_i. By Lemma 5.3.2, we know $v_1 = \min(I)$, and hence that v_i may not be wholly left of v_1. Therefore $v_1 < u < v_i$. Since v_i is known to be wholly right of u, there must be some $k \leq i$ such that v_k is the first interval in P that is wholly right of u. Therefore v_{k-1} may not be wholly right of u, but it intersects v_k, which we know to be wholly right of u, and so $v_{k-1} \not\ni u$ (see Figure 5.2). By Lemma 5.3.4, the presence of the $\{v_{k-1}, v_k\}$ edge in P means that $v_k = \min(s \in I \setminus \{v_1, \ldots, v_{k-1}\} \mid s \not\ni v_{k-1})$. Yet since $k \leq i$ and $u \in I \setminus \{v_1, \ldots, v_i\}$, we know that $u \in I \setminus \{v_1, \ldots, v_{k-1}\}$, and we also know that $u < v_k$, which means that v_k cannot have been the leftmost remaining interval. Thus we have arrived at a contradiction, and so our original assumption must have been false, as desired. □

Manacher, Mankus, and Smith [39] show that this algorithm can be implemented in $O(|I| \log |I|)$ time and $O(|I|)$ space using a data structure they describe as a heap but which is more commonly called a segment tree. Segment trees are binary trees in which the leaves represent individual items in an ordered collection and each internal node stores the result of some associative aggregation operation on the leaves of the subtree rooted at that node. Updating a leaf in a range tree requires updating the values of its parents, but maintaining this structure allows us to efficiently find the result of aggregating any contiguous range by combining the values of internal nodes that are entirely spanned by this range. For a balanced tree, updates take logarithmic time, as we must update every ancestor of the modified leaf. In this case, they use range trees to select the leftmost remaining interval that

intersects p. Each interval populates a leaf of the range tree ordered by their left endpoints. Each leaf initially contains the right endpoint of the corresponding interval, and internal range nodes contain the minimum of all right endpoints in the range. This initial tree can be constructed in linear time. To find $\min(s \in S \mid s \not\ni p)$, we query the range tree to find the minimum right endpoint for intervals with left endpoints less than $r(p)$, which can be done in logarithmic time. They show (as does Lemma 5.3.5) that no interval in S can be wholly to the left of p, so this query will always find the interval that intersects p with leftmost right endpoint. To remove an interval from S we can simply replace it with a sentinel value that is considered greater than every other endpoint and so will never be selected as the minimum. This means that we can select the leftmost neighbour of p and remove it from S in logarithmic time, giving an overall complexity of $O(|I| \log |I|)$.

5.4 Greedy Interval MIST Algorithm

The logic of the greedy Hamiltonian path algorithm for interval graphs (see Algorithm 6 can be extended to give a greedy algorithm for finding MISTs of connected interval intersection graphs. Section 5.4.1 presents a recursive construction of such an algorithm, and proves its correctness. Section 5.4.2 adapts this to an iterative version of the same algorithm, and Section 5.4.3 shows that this can be implemented in $O(|I| \log |I|)$.

5.4.1 Recursive Algorithm

As detailed in Section 5.3, if Algorithm 6 fails to find a Hamiltonian path P where $(P, S) = \textsc{FindPath}(I)$ such that $S = \varnothing$, then no Hamiltonian path exists. In the case that S is nonempty, any MIST must have more than two leaves. If we collapse the tail (all but the first vertex) of P into a single vertex, this should remove a leaf from the MIST, and so gives a smaller subproblem to solve. If we can then restore the collapsed vertices, this suggests a recursive algorithm based on repeated application of FindPath. This algorithm is presented in Algorithm 7.

Any connected graph must have a MIST, and any MIST of this graph must have the same, minimal, number of leaves. Therefore any set of intervals with a connected intersection graph has a specific number of leaves that appear in any MIST of that graph. We aim to show the correctness of Algorithm 7 by induction on the number of leaves in a MIST of the intersection graph of the input.

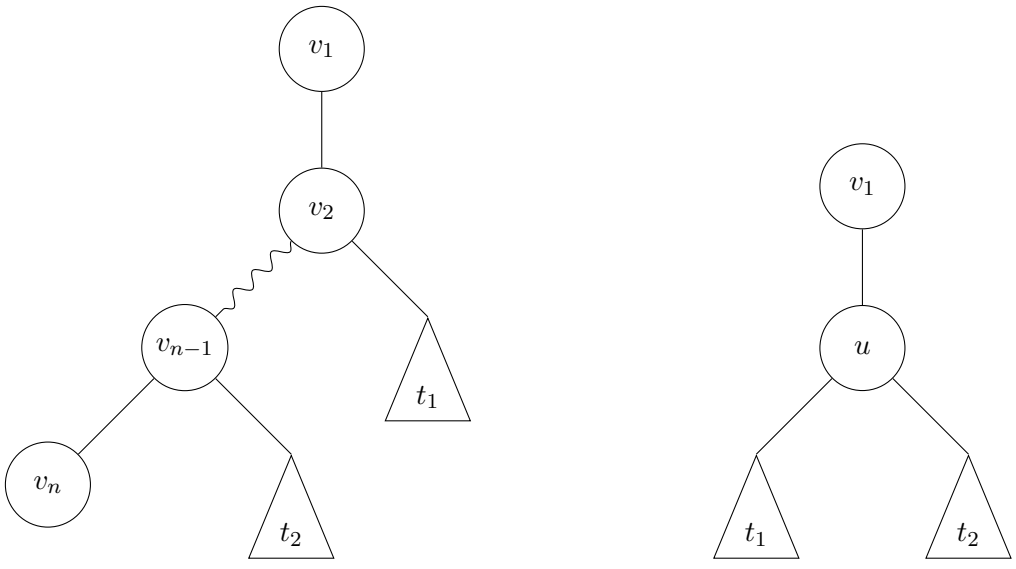
Definition 5.4.1. Leaf-Leaf Path: A *leaf-leaf path* in a tree is a path in which both ends of the path are leaves in the tree.

If we may assume that the greedy path found by FindPath is a leaf-leaf path in some MIST, we can show the inductive step holds. We do this by showing that contracting the tail of the greedy path as in Algorithm 6 will remove a leaf from the tree, giving a smaller subproblem that we can solve recursively. Then re-expanding the tail will reintroduce at most one leaf, which must therefore give a MIST.

Lemma 5.4.1. *Let I be a set of intervals with connected intersection graph G. Let $(P, S) = \textsc{FindPath}(I)$. Let T be a MIST of G with $l > 2$ leaves in which P is a leaf-leaf path (such as in Figure 5.3a). Let I' be any set of intervals with connected intersection graph G' which contains an*

Algorithm 7 Build a MIST for a set of intervals I with connected intersection graph

function FINDMIST(I)
  $(P, S) \leftarrow$ FINDPATH(I) ▷ Attempt to find a Hamiltonian path
  if $S = \varnothing$ **then**
    return P ▷ Any Hamiltonian path is a MIST
  end if
  $V_t \leftarrow V(P) \setminus \{\min(I)\}$ ▷ The tail of the path
  $u \leftarrow \bigcup V_t$ ▷ The union of the tail, another interval
  $I \leftarrow (I \setminus V_t) \cup \{u\}$ ▷ Replace the tail with its union
  $(V_T, E_T) \leftarrow$ FINDMIST(I) ▷ Recursively solve subproblem
  $V_T \leftarrow (V_T \setminus \{u\}) \cup V_t$ ▷ Re-expand u
  $E_T \leftarrow E_T \cup E(P)$
  for $e \in E_T \mid u \in e$ **do** ▷ Replace each edge incident on u with an edge to some element of V_t
    $v \leftarrow e \setminus u$ ▷ v was a neighbour of u
    Let $u' \in \{x \in V_t \mid x \not\simeq v\}$ ▷ Select an arbitrary neighbour of v from V_t
    $E_T \leftarrow (E_T \setminus \{e\}) \cup \{u', v\}$ ▷ Replace the edge
  end for
  return (V_T, E_T)
end function



(a) A tree in which P is a leaf-leaf path.

(b) The result of replacing the tail of P with its contraction, u.

Figure 5.3: An example of converting a tree in which $P = \langle v_1, \ldots v_n \rangle$ is a leaf-leaf path into a tree with one fewer leaf where u is the contraction of $v_2, \ldots, v_n$.

$l-1$ *leaf MIST. If* FindMIST(I') *will return a MIST of* G'*, then* FindMIST(I) *will return a MIST of* G*.*

Proof. Let $P = \langle v_1, \ldots, v_n \rangle$ such that $v_1 = \min(I)$, as per Lemma 5.3.2. Let $V_t = V(P) \setminus \{\min(I)\} = \{v_2, \ldots, v_n\}$ and $u = \bigcup V_t$, as in Algorithm 7. Since we know the elements of V_t form a connected path, their union u will be an interval itself, and any interval that intersects an element of V_t must also intersect u. Therefore we may construct a spanning tree T' by taking the vertex contraction of the elements of V_t and replacing them with u and all edges incident on one of these elements with an edge to u instead. An example of this transformation is given in Figure 5.3.

The degrees of any vertices that are not elements of V_t will not have changed, and since T' is a spanning tree but v_1 is a leaf connected only to u, u must be internal in T'. Since V_t contained a leaf of T, T' therefore must have $l-1$ leaves.

We can therefore choose $I' = V(T') = (I \setminus V_t) \cup \{u\}$. Since we know T' is an $l-1$ leaf spanning tree of G', therefore any MIST of G' may have at most $l-1$ leaves. Hence we may assume $T'_g = \text{FindMIST}(I')$ to be a MIST of G' and to contain at most $l-1$ leaves.

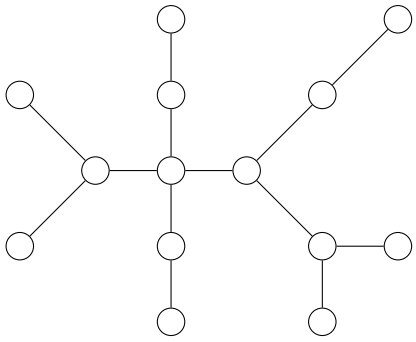
We can now re-expand u in T'_g, replacing it with the elements of V_t and reintroducing the edges from P to give a new tree T_g. Any other intervals that intersected u must also therefore intersect some element of V_t, and so all edges incident on u can be replaced with an edge to some element of V_t. Since all other vertices in V_t are already internal in P, only v_n may be a new leaf, meaning T_g can have at most l leaves, and so must be an l leaf MIST of G, as desired. □

We therefore wish to show that for any connected interval intersection graph there exists some MIST in which the greedy path is a leaf-leaf path. This will require us to show that the existence of a MIST is sufficient to show the existence of a MIST with this property. To this end we will explore several results on prefixes of the desired path and MISTs which contain them. Since the path in question is constructed greedily, it stands to reason that even if other options that were not chosen by the greedy algorithm are removed, it will still make the same choices, as they will continue to be the locally best option.

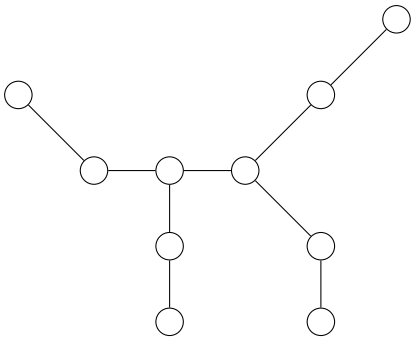
First, if we know that FindPath would chose a particular interval to extend the greedy path, then removing any of the other remaining intervals would not cause it to choose a different interval.

Lemma 5.4.2. *Let* I *be a set of intervals with connected intersection graph* G*. Let* $(P, S) =$ FindPath(I)*. Let* $P = \langle v_1, v_2, \ldots, v_n \rangle$*. Let* $I' \subseteq I$ *be a subset of intervals and* $(P', S') =$ FindPath(I') *such that* P' *has a common prefix* $\langle v_1, v_2, \ldots, v_k \rangle$ *with* P*. If* $v_{k+1} \in I'$*, then* $\langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$ *is a prefix of both* P *and* P'*.*

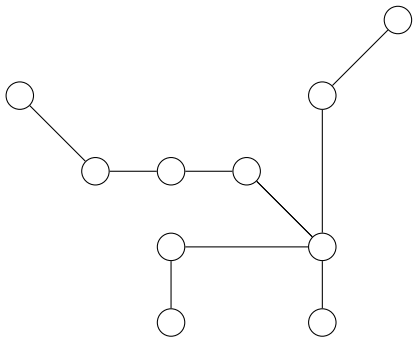
Proof. Consider the execution of FindPath(I) and FindPath(I'). We know that both will reach a point at which $P = \langle v_1, v_2, \ldots, v_k \rangle$. Let us assume $v_{k+1} \in I'$. By Lemma 5.3.3, therefore $v_{k+1} \in S$ in both, and S in FindPath(I) is a superset of that in FindPath(I'). We know that FindPath(I) will then find $\min(s \in S \mid s \not\ni P_{|P|}) = v_{k+1}$. Since it is the minimum in FindPath(I), and FindPath(I') does not have any other intervals from which to select, it must also be the minimum interval in FindPath(I'). Therefore both FindPath(I) and FindPath(I') will append v_{k+1} to P, and so $\langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$ will be a prefix of both. □



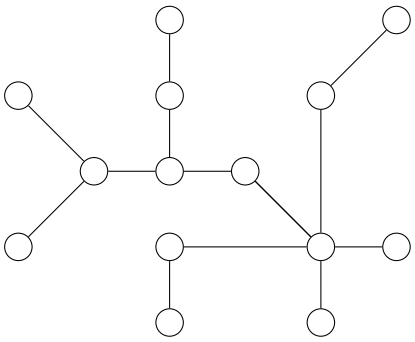
(a) A 7-leaf tree T.



(b) A leaf-induced subtree t of T with 4 leaves.



(c) An alternative 4-leaf spanning tree t' of $V(t)$.



(d) T', the result of replacing the edges of t in T with those from t', another 7-leaf tree.

Figure 5.4: An example of replacing a leaf-induced subtree of a tree.

Next, we use this to show that for any prefix of the greedy path, removing intervals other than those in the path will not cause FindPath to choose a different prefix.

Lemma 5.4.3. *Let I be a set of intervals with connected intersection graph G. Let $(P, S) =$ FindPath(I). Let $P = \langle v_1, v_2, \ldots, v_n \rangle$. Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be some prefix of P. Let $I' \subseteq I$ be some subset of intervals such that $V(p) \subseteq I'$. Let $(P', S') =$ FindPath(I'). Then p is also a prefix of P'.*

Proof. We will show this by induction on the length of the prefix, k.

- Base case $k = 1$: This follows from Lemma 5.3.2, as if $\min(I) \in I'$, then $\min(I') = \min(I)$ must be a prefix of both P and P'.
- Inductive step: By the inductive hypothesis we may assume this holds for prefixes of length k. Therefore we know $\langle v_1, v_2, \ldots v_k \rangle$ is a prefix of both P and P'. By Lemma 5.4.2, if $v_{k+1} \in I'$, then $\langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$ is a prefix of P'. Therefore this holds for any prefix of length $k + 1$.

By induction it follows that for any prefix p of P, p is also a prefix of P'. □

Using this we can establish some transformations on MISTs that will allow us to construct desirable substructures while remaining a MIST. We will use these to show that we can transform any MIST into a MIST that contains the greedy path as a leaf-leaf path, as desired by Lemma 5.4.1.

Definition 5.4.2. Leaf-Induced Subtree: A *leaf-induced subtree* of a tree for a given set of leaves of that tree is the minimal subtree that contains those leaves. See Figure 5.4b for an example. Note that any leaf-leaf path is a leaf-induced subtree.

Any leaf-induced subtree of a MIST can be replaced with an alternative spanning tree of the same vertices, and so long as the alternative doesn't have more leaves than the original subtree, the result must still be a MIST. We show this by considering that the degree of any vertices not in the subtree does not change, and merging the alternative tree back into the MIST cannot decrease the degree of any vertices in the alternative tree, and so cannot possibly increase the number of leaves.

Lemma 5.4.4. *Let T be a MIST of some graph G. Let t be a leaf-induced subtree of T with l leaves. Let t' be any spanning tree of $V(t)$ with at most l leaves. Let T' be the result of removing all edges in t from T and replacing them with those from t'. T' is a MIST of G, and the leaves of t' are leaves in T'.*

Proof. See Figure 5.4 for an example of the replacement process. Since t' is still connected, so must be T', and the number of vertices and edges in T' is the same as in T, and so T' must still be a tree. Since we only remove edges that appear in t, any vertex that does not appear in t cannot decrease in degree, and so may not become a leaf. Furthermore, any vertices in t' cannot have lesser degree in T' than in t', as all edges that appear in t' also appear in T', and so no internal vertex in t' can be a leaf in T'. Therefore T' can have at most as many leaves as T. But T was defined as being a MIST, so no other spanning tree may have fewer leaves than T, and so T' must have the same number of leaves as T, meaning all leaves in t' are leaves in T' and T' is a MIST of G. □

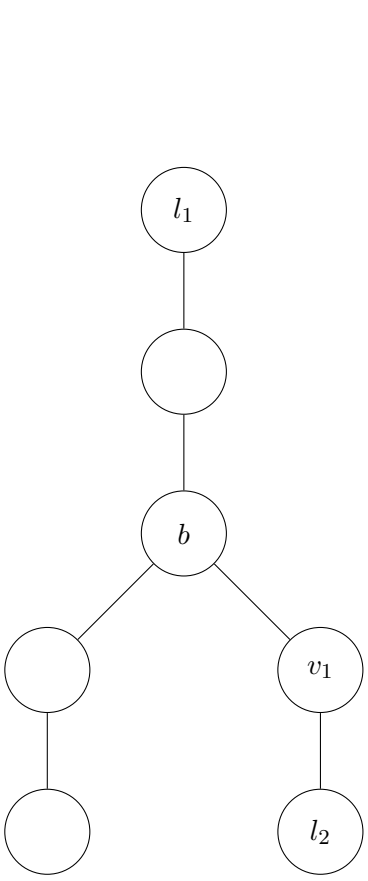
Using this, we can replace any leaf-leaf path in a MIST with any other path through the same vertices while remaining a MIST. Specifically, we can replace it with the greedy Hamiltonian path given by FindPath for those intervals. Since we already have a MIST, the number of leaves cannot possibly decrease, meaning that the leaves of the greedy path must also be leaves in the new MIST.

Lemma 5.4.5. *Let I be a set of intervals with connected intersection graph G. Let T be a MIST of G. Let p be a leaf-leaf path in T. Let $(P, S) = \textsc{FindPath}(V(p))$. There exists a MIST of G in which P is a leaf-leaf path.*

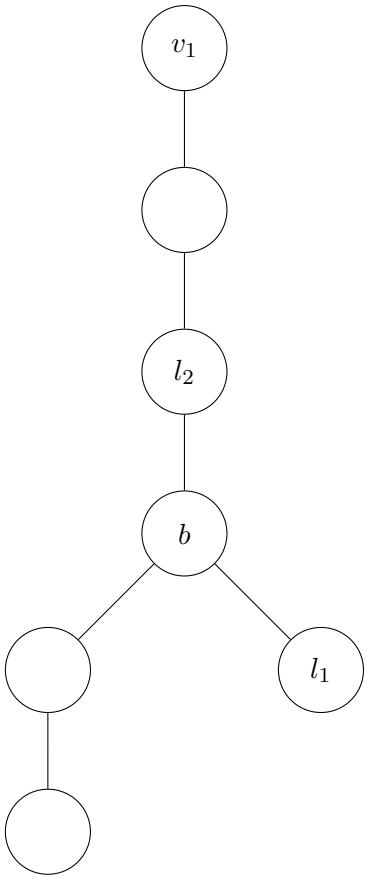
Proof. Note that p is a leaf-induced subtree of T. Since p is a path, we know the intersection graph of $V(p)$ must have a Hamiltonian path. Therefore, by Lemma 5.3.1, P will be a Hamiltonian path of $V(p)$. By Lemma 5.4.4 we can replace the edges from p with those from P to give a MIST which contains P as a subgraph and in which the leaves of P are leaves of the MIST. Therefore there exists a MIST of G in which P is a leaf-leaf path. □

We can now use these transformations to construct MISTs which contain prefixes of the greedy path.

Definition 5.4.3. Leaf Path: A *leaf path* in a tree is a path in which at least one end of the path is a leaf in the tree.



(a) A tree in which v_1 is not a leaf.



(b) A possible result of reordering $\langle l_1, \ldots, l_2 \rangle$ using FindPath. Note that since $v_1 = \min(I)$ it will now be a leaf.

Figure 5.5: An example of converting a tree in which $v_1 = \min(I)$ is not a leaf into one where it is by replacing the $\langle l_1, \ldots, l_2 \rangle$ path with the path given by FindPath.

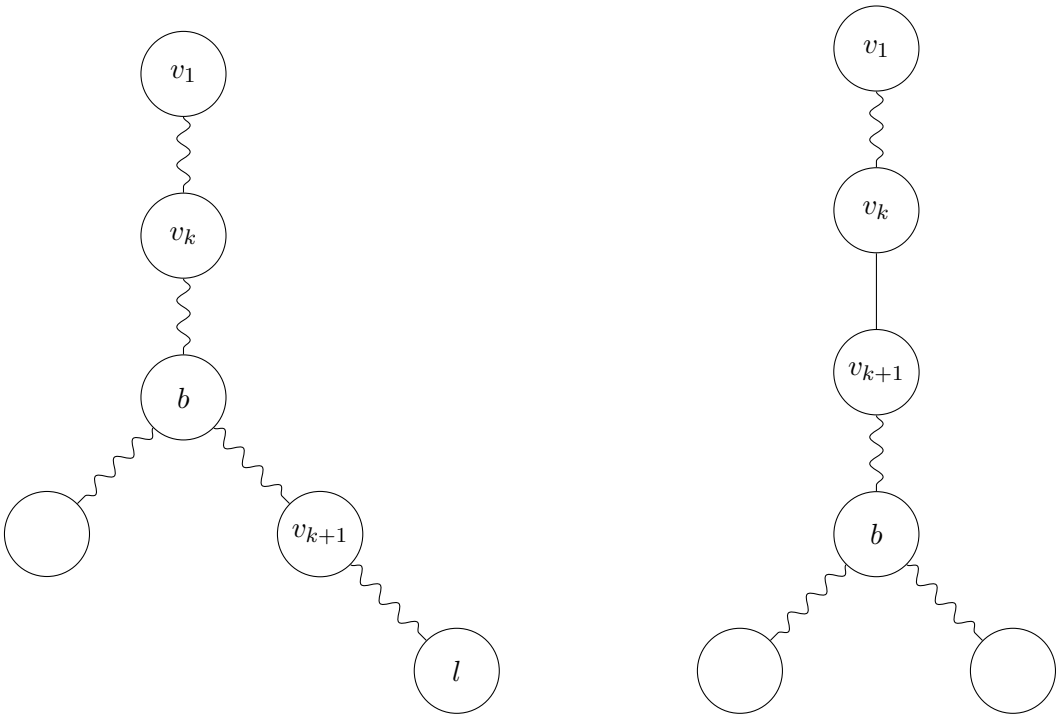
Definition 5.4.4. Prefix Path: Let I be a set of intervals with connected intersection graph G. Let T be a spanning tree of G. Let $(P, S) = \text{FindPath}(I)$. A *prefix path* is a leaf path of T which is a prefix of P.

Since we know the leftmost interval, $\min(I)$, will always be a leaf in the greedy path, we can reorder any leaf-leaf path that contains it to construct a MIST in which $\min(I)$ is a leaf, as it must be in order for the MIST to contain the greedy path as a leaf-leaf path as desired by Lemma 5.4.1.

Lemma 5.4.6. *Let I be a set of intervals with connected intersection graph G. Let T be a MIST of G. There exists a MIST of G in which the leftmost interval* $\min(I)$ *is a leaf.*

Proof. Select any leaf-leaf path p through $\min(I)$ in T. Let $(P, S) = \text{FindPath}(V(p))$. By Lemma 5.3.2, $\min(I)$ is a leaf in P. By Lemma 5.4.5, there exists a MIST of G in which P is a leaf-leaf path, and hence in which $\min(I)$ is a leaf. A simple example of this transformation is given in Figure 5.5. □

Definition 5.4.5. Branch Vertex: A *branch vertex* or simply *branch* in a tree is any vertex with degree greater than 2.



(a) A tree in which the longest prefix path $\langle v_1, \ldots, v_k \rangle$ does not contain a branch.

(b) The tree after reordering $Q = \langle v_1, \ldots, l \rangle$ using FindPath to construct a longer prefix path.

Figure 5.6: An example of reordering a tree to extend the longest prefix path if it does not contain a branch. After repeated application, eventually the longest prefix path in the tree must contain a branch.

For any prefix path that does not contain a branch vertex, there must be a leaf-leaf path that starts with this prefix and goes through any other vertex we desire. By selecting such a path that contains the next interval that would appear in the greedy path, we can reorder this path to construct a MIST which contains a longer prefix path than the original.

Lemma 5.4.7. *Let I be a set of intervals with connected intersection graph G. Let T be a MIST of G. Let $(P, S) = \textsc{FindPath}(I)$. Let p be the longest prefix path in T. If no vertex in p is a branch, then there exists a MIST of G which contains a longer prefix path than p.*

Proof. An example is given in Figure 5.6. Let $p = \langle v_1, \ldots, v_k \rangle$ and $P = \langle v_1, \ldots, v_k, v_{k+1}, \ldots v_n \rangle$ such that $v_1 = \min(I)$. We know therefore that the edge $\{v_k, v_{k+1}\}$ exists in G. Let $p' = \langle v_1, \ldots, v_k, v_{k+1} \rangle$ be a longer prefix of P than p. It suffices to show that there exists a MIST of G which contains p' as a leaf path. Let Q be any leaf-leaf path through v_1 and v_{k+1}. Since no vertex in p is a branch, any path from v_1 to another leaf must first pass through all vertices in p, and so Q must therefore contain p as a subgraph. Let $I' = V(Q)$. Therefore $V(p') \subseteq I' \subseteq I$. Let $(P', S') = \textsc{FindPath}(I')$. By Lemma 5.4.3, any prefix of P must also therefore be a prefix of P', and so p' is a prefix of P'. By Lemma 5.4.5, there exists a MIST of G in which P' is a leaf-leaf path. In any such MIST p', being a prefix of P', must be a leaf path. Hence there exists a MIST of G which contains p' as leaf path, as desired. □

Repeating this process will extend the prefix path until it reaches a branch. Otherwise it would eventually grow so long that it must contain all vertices, and so must be a Hamiltonian path, which means any tree with a branch could not have been a MIST.

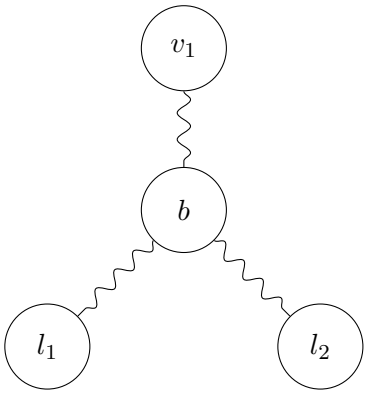
Lemma 5.4.8. *Let I be a set of intervals with connected intersection graph G. Let there exist a MIST of G with $l > 2$ leaves. Let $(P, S) = \textsc{FindPath}(I)$. There exists a MIST of G which contains a prefix path which contains a branch.*

Proof. By Lemma 5.4.6, there exists a MIST of G in which the leftmost interval $\min(I)$ is a leaf. By Lemma 5.3.2, $\min(I)$ is also a leaf of P, and hence a prefix of length 1. Therefore there exists a MIST of G which contains a length $k = 1$ prefix path. This prefix path contains a single vertex, a leaf, and so contains no branch.

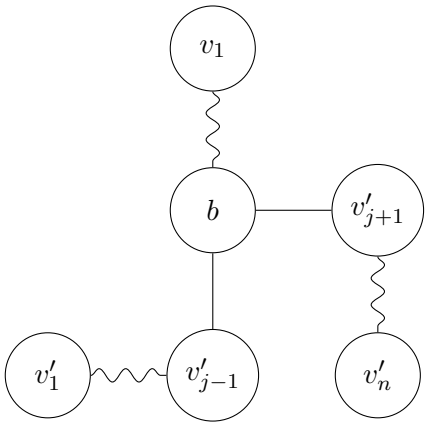
By Lemma 5.4.7, if there exists a MIST of G in which the longest prefix path contains no branch, then there exists a MIST of G which contains a longer prefix path. Through repeated application of this result we must either find a MIST which contains a prefix path with a branch or a prefix path through all vertices. Since $l > 2$, we know that G does not contain a Hamiltonian path, and hence no prefix path may contain all vertices. Therefore there must exist a MIST of G which contains a prefix path which contains a branch. □

Combining these results, we can show first that for any connected interval intersection graph for which there exists a three-leaf MIST, there must also exists a MIST which contains the greedy path as a leaf-leaf path. We will show this using the following steps:

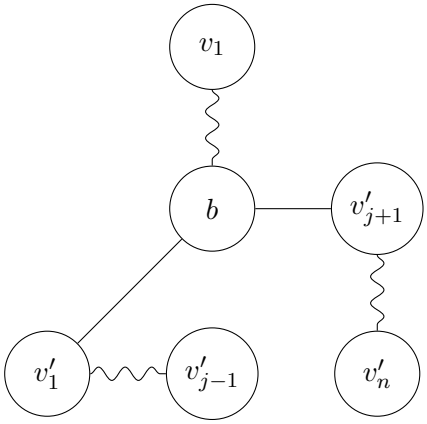
- As above, extend the prefix path until it contains a branch (the unique branch in a three-leaf tree)



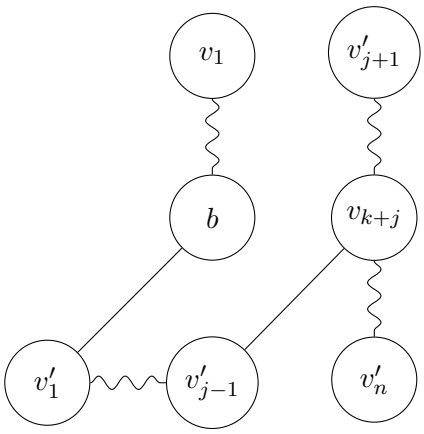
(a) A 3-leaf tree with unique branch $b = v_k$ such that $\langle v_1, \ldots, v_k \rangle$ is a prefix path.



(b) Reorder the leaf-leaf path $\langle l_1, \ldots, l_2 \rangle$ using FindPath such that $v'_j = b$.



(c) Use $v'_1 \ni\!\!\!\in b$ to reattach b to the other end of $\langle v'_1, \ldots, v'_{j-1} \rangle$. Note this forms a prefix path $\langle v_1, \ldots, v_{k+j-1} \rangle$ such that $v_{k+j-1} = v'_{j-1}$.



(d) Use $v'_{j-1} \ni\!\!\!\in v_{k+j}$ to connect $\langle v'_{j+1}, \ldots, v'_n \rangle$ to v'_{j-1} such that b is no longer the branch and so the branch-free prefix path is now longer.

Figure 5.7: The process of extending a branch-free prefix path in a 3-leaf MIST. As shown in Lemma 5.4.9, repeated application of this transformation must eventually construct a MIST in which the greedy path is a leaf-leaf path.

- The resulting MIST must have a leaf-leaf path through the branch that contains every vertex not in the prefix path (Figure 5.7a)
- Reorder this leaf-leaf path using FindPath (Figure 5.7b)
- Any interval that appears before the branch in this reordering must be less than the branch, but cannot intersect the branch's predecessor, and so must be wholly contained within the branch interval
- Therefore the leftmost interval in this reordering must be the leftmost remaining neighbour of the branch
- Rearrange the MIST to connect the branch to its leftmost remaining neighbour, thereby extending the prefix path (Figure 5.7c)
- If the end of the new prefix path is not adjacent to any remaining interval, then it must be the end of the greedy path, and so we have found a MIST that contains the greedy path as a leaf-leaf path, as desired
- If not, then by connecting it to one of the vertices in the remaining path segment, we can disconnect this segment from the branch, making it no longer the branch (Figure 5.7d)
- Therefore we have constructed a longer branch-free prefix path, which cannot continue forever, and so must eventually construct a MIST which contains the greedy path as a leaf-leaf path

Lemma 5.4.9. *Let I be a set of intervals with connected intersection graph G. Let T be a MIST of G with 3 leaves. Let $(P, S) = \textsc{FindPath}(I)$. There exists a MIST of G in which P is a leaf-leaf path.*

Proof. By Lemma 5.4.8, we may assume without loss of generality that T contains a prefix path that contains a branch. Any tree with exactly 3 leaves contains a single branch vertex. Let b be the unique branch vertex of T. Let $v_k = b$ such that $p = \langle v_1, \ldots, v_{k-1} \rangle$ is the longest prefix path that does not contain a branch. This construction is shown in Figure 5.7a.

Let $I' = I \setminus V(p)$. We know that there exists a leaf-leaf path in T through every interval in I', and so the intersection graph of these intervals must have a Hamiltonian path. Therefore let $(P', \varnothing) = \textsc{FindPath}(I')$.

By Lemma 5.3.5, no interval in I' may be wholly to the left of any in p. Therefore they must either intersect v_{k-1} or be wholly to its right. However, by Lemma 5.3.4, the presence of the $\{v_{k-1}, b\}$ edge in P shows that any remaining interval that intersects v_{k-1} may not be left of b. Conversely, any interval in I' less than b may not intersect v_{k-1}, and so must be wholly to its right. Therefore, since we know $v_{k-1} \not\ni b$, the left endpoint of b must be before the right endpoint of v_{k-1}, and hence before the left endpoint of any interval in I', which we know to be wholly to the right of v_{k-1}. It follows that any interval in I' less than b must be wholly contained within b, since we know its right endpoint must be left of the right endpoint of b, and its left endpoint must be right of the left endpoint of b.

We may assume b is not a leaf in P', as otherwise we could attach p to P' using the $\{v_{k-1}, v\}$ edge to construct a Hamiltonian path of G, which contradicts with the assumption that the MIST of

G has 3 leaves. Let $P' = \langle v'_1, \ldots, v'_j, \ldots, v'_n \rangle$ such that $v'_j = b$ (see Figure 5.7b). By Lemma 5.3.2, $v'_1 = \min(I') < b$. Therefore, as above, v'_1 must be wholly contained within b, and so $v'_1 \ni\!\!\!\not\, b$.

Consider v_{k+1}, the next interval after b in the original path P. By Lemma 5.3.4, the presence of the $\{b, v_{k+1}\}$ edge shows that v_{k+1} is the leftmost of the remaining intervals that intersects b. But, since $v'_1 = \min(I')$ and $v'_1 \ni\!\!\!\not\, b$, it must also be the case that v'_1 is the leftmost of the remaining intervals that intersects b. Hence $v_{k+1} = v'_1$. Furthermore, we know from P' that for all v'_i before b in P', $i < j - 1$, the leftmost remaining interval that intersects v'_i is v'_{i+1}, so it follows that for all $i < j$, $v_{k+i} = v'_i$.

We can therefore construct a path $p' = \langle v_1, \ldots, v_{k+j-1} \rangle$ that is a prefix of P (see Figure 5.7c). We also know from P' that it is possible to construct a path $p'' = \langle v'_{j+1}, \ldots, v'_n \rangle$ through every interval in $I \setminus V(p')$.

If $p' = P$, then we can construct a spanning tree of G by connecting p' and p'' using the $\{b, v'_{j+1}\}$ edge from P'. Since v'_{j+1} is a leaf in p'', and b is internal in p', this spanning tree will have exactly three leaves: v_1, v_{k+j-1}, and v'_n, and so must be a MIST of G. Therefore we have shown there exists a MIST of G in which P is a leaf-leaf path, as desired.

On the other hand, if $p' = P$, then since p' is a prefix of P we know the edge $\{v_{k+j-1}, v_{k+j}\}$ from P must connect the leaf v_{k+j-1} of p' to some vertex in p''. The vertex v_{k+j} may not be a leaf in p'', as otherwise we could use this edge to construct a Hamiltonian path in G, which contradicts the assumption that the MIST of G has 3 leaves. Therefore v_{k+j} must be internal within p'', and so connecting p' and p'' using this edge gives a spanning tree of G with exactly three leaves (see Figure 5.7d), which is therefore a MIST of G.

While this MIST does not necessarily contain P as a leaf-leaf path, it does contain a longer branch-free prefix path than p, as v_{k+j-1} is the unique branch vertex in this MIST, and so $\langle v_1, \ldots, v_{k+j-2} \rangle$ may not contain any branches. Therefore through repeated application of the above process, the length of the longest branch-free prefix path must strictly increase. This process cannot continue forever, as the prefix path cannot grow longer than P itself, and so must eventually give a MIST in which P is a leaf-leaf path, as desired. □

We can now generalise this property from three-leaf MISTs to MISTs with any number of leaves so as to show that for any connected interval intersection graph, there exists a MIST which contains the greedy path as a leaf-leaf path. For any prefix path in any MIST, we can select a three-leaf leaf-induced subtree that contains that prefix and the next vertex in the greedy path. We can therefore rearrange this subtree, and hence the whole MIST, to extend the prefix path. This process can be repeated indefinitely until the prefix path is the whole greedy path.

Lemma 5.4.10. *Let I be a set of intervals with connected intersection graph G. Let $(P, S) =$ FindPath(I). There exists a MIST of G in which P is a leaf-leaf path.*

Proof. Let l be the number of leaves in a MIST of G.

If $l = 2$, then there exists a Hamiltonian path in G. By Lemma 5.3.1, if there exists a Hamiltonian path, then P will be a Hamiltonian path, and will therefore be a MIST in which P is a leaf-leaf path, as desired.

If $l \geq 3$, then we will proceed by induction on the greatest length k for which there exists a MIST of G which contains a prefix path of length k.

- Base case $k = 1$: By Lemma 5.4.6, there exists a MIST of G in which $\min(I)$ is a leaf. By Lemma 5.3.2, $\min(I)$ is also the first element in P. Therefore there exists a MIST of G which contains a length $k = 1$ prefix path.

- Inductive step: By the inductive hypothesis we may assume there exists a MIST T of G with a prefix path p of length k such that $p = \langle v_1, \ldots, v_k \rangle$. Let u be any leaf in T such that the leaf-leaf path from v_1 to u contains p. Let w be any leaf in T such that the leaf-leaf path from v_1 to w contains v_{k+1}. The union of these two leaf-leaf paths is then a leaf-induced subtree t with three leaves: v_1, u, and w. Let $I' = V(t)$, and G' be the intersection graph of I'. Let $(P', S') = \textsc{FindPath}(I')$. Let $p = \langle v_1, \ldots, v_{k+1} \rangle$ be the length $k+1$ prefix of P. Since $V(p) \subseteq I'$, by Lemma 5.4.3, p is also a prefix of P'. By Lemma 5.4.9, there exists a MIST t' of G' that contains P' as a leaf-leaf path. Therefore, by Lemma 5.4.4, we can replace t with t' in T to give a MIST T' of G in which P' is a leaf-leaf path. Since p is a prefix of both P and P', it must therefore be a prefix path in T' Therefore there exists a MIST of G which contains a length $k+1$ prefix path.

Therefore, by induction, there exists a MIST of G which contains a prefix path of any length, up to and including the length of P itself. Therefore there exists a MIST of G in which P is a leaf-leaf path. □

Finally, we combine this with our original result to give an inductive proof of correctness for Algorithm 7.

Theorem 5.4.11. *Let I be a set of intervals with connected intersection graph G. $\textsc{FindMIST}(I)$ will return a MIST of G.*

Proof. Let $(P, S) = \textsc{FindPath}(I)$. We will proceed by induction on the number of leaves l in a MIST of G.

- Base case $l = 2$: By Lemma 5.3.1, if G contains a Hamiltonian path, then P will be a Hamiltonian path and $S = \varnothing$. Therefore $\textsc{FindMIST}(I)$ will call $\textsc{FindPath}(I)$, find S to be empty, and will immediately return P, which as a Hamiltonian path must be a MIST of G.

- Inductive step: Let I' be a set of intervals with connected intersection graph G' such that there exists a MIST of G' with $l - 1 \geq 2$ leaves. By the inductive hypothesis we may assume that $\textsc{FindMIST}(I')$ will return a MIST of G'. By Lemma 5.4.10, there exists a MIST T' of G' in which P is a leaf-leaf path. Therefore, by Lemma 5.4.1, $\textsc{FindMIST}(I)$ will return a MIST of G.

Therefore, by induction, $\textsc{FindMIST}(I)$ will return a MIST of G. □

5.4.2 Iterative Algorithm

It is possible to convert the recursive Algorithm 7 presented in Section 5.4.1 to an equivalent iterative form using the following observations:

If FindPath does not find a Hamiltonian path, the remaining vertices must not intersect the ends of the greedy path, but rather must be wholly to their right.

Lemma 5.4.12. *Let I be a set of intervals with connected intersection graph G. Let $(P, S) =$ FindPath(I). Every interval in S must be wholly right of the leaves of P.*

Proof. FindPath(I) would only return when it had found a vertex with no remaining neighbours in S. Therefore the final vertex t in the path must be wholly left of everything in S. Furthermore, since by Lemma 5.3.2 $\min(I)$ is a leaf in P, it must be left of t, and so anything wholly right of t must also by wholly right of $\min(I)$. Therefore every interval in S must be wholly right of both $\min(I)$ and t, the leaves of P. □

The subproblem constructed by Algorithm 7 will have the same leftmost interval as the original problem, and the only neighbour of this interval will be u, the union of the greedy path tail vertices V_t, and so these must be the first two vertices of the greedy path for the subproblem.

Lemma 5.4.13. *Let I be a set of intervals with connected intersection graph G. Let $I' = (I \setminus V_t) \cup \{u\}$ be the subproblem constructed in the execution of FindMIST(I). The first two vertices selected by FindPath(I') will be $\min(I)$ and u.*

Proof. Since $\min(I) \in I'$, by Lemma 5.3.2, it will be the first vertex chosen by FindPath(I'). Furthermore, by Lemma 5.4.12, any vertex not in V_t must be wholly right of $\min(I)$. Therefore u must be the only vertex that intersects $\min(I)$, and so will be chosen next. □

Therefore, after selecting the first two vertices $\min(I)$ and u, the invocation of FindPath(I') will be left with the same set of remaining intervals S as was returned by FindPath(I). The difference then is that while FindPath(I) was unable to find a remaining neighbour of p, and so had to return, FindPath(I') is looking for an interval that intersects $u = \bigcup V_t$. Furthermore, $V_t = V_P \setminus \{\min(I)\}$, but we know that nothing in S can intersect $\min(I)$, so equivalently we may select the leftmost interval that intersects $\bigcup V_P$.

Therefore we can construct an iterative version of Algorithm 7 by modifying Algorithm 6 such that, when no remaining interval intersects p, it instead selects $t \leftarrow \min(s \in S \mid s \not\ni \bigcup V_P)$. Of course this vertex will not be adjacent to p, so we must then select a new parent to which we can connect t. In the recursive version, we simply select any adjacent vertex in V_t, and similarly here we can select any adjacent vertex in V_P. This iterative construction of FindMIST is presented in Algorithm 8. An implementation of this algorithm is available from `https://github.com/gozzarda/interval_mist` [23].

Naively, this algorithm can be implemented in $O(|I|^2)$ time by simply iterating through all remaining intervals in order to find the leftmost one that intersects our desired interval. However, we can improve on this complexity using the techniques described in Section 5.3.

5.4.3 Efficient Implementation

As in Section 5.3, we can use segment trees to find $\min(s \in S \mid s \not\ni p)$ in logarithmic time. Notably, this result depends on the result from Lemma 5.3.5 that no interval in S may be wholly left of p, and therefore any remaining interval with a left endpoint less than the right endpoint of p must intersect p, allowing us to simply query a prefix of intervals ordered by left endpoint. The same segment tree

Algorithm 8 Build a MIST for a set of intervals I

```
1: function FindMIST(I)
2:     p ← min(I)
3:     S ← I \ {p}
4:     V_P ← {p}
5:     E_P ← ∅
6:     while S = ∅ do
7:         if ∃ s ∈ S : s ⋟ p. then
8:             t ← min(s ∈ S | s ⋟ p)              ▷ Same criteria as Algorithm 6 by default
9:         else                                    ▷ If the normal criteria cannot select anything
10:            t ← min(s ∈ S | s ⋟ ⋃V_P)            ▷ Allow branching from any vertex in the tree
11:            Let p ∈ {v ∈ V_P | v ⋟ t}           ▷ Connect t to an arbitrary neighbour in V_P
12:        end if
13:        S ← S \ {t}
14:        V_P ← V_P ∪ {t}
15:        E_P ← E_P ∪ {p, t}
16:        p ← t
17:    end while
18:    T ← (V_P, E_P)
19:    return T
20: end function
```

can be used to find $\min(s \in S \mid s \not\ni \bigcup V_P)$, as for any interval to be wholly left of $\bigcup V_P$, it must be wholly left of everything in V_P, which again cannot be the case.

Even if we can select the leftmost neighbour of $\bigcup V_P$ efficiently, this is still dominated by then selecting a neighbour of that vertex from V_P, for which we cannot immediately adapt the same segment tree. We can, however, use the same techniques. Since we only wish to select any arbitrary neighbour from $\{v \in V_P \mid v \not\ni t\}$, all we require is that if there exists some interval in V_P that intersects t, we are able to find some such interval. We know that for some interval v to intersect t, then $l(v) < r(t) \wedge l(t) < r(v)$. Using a similar segment tree to above, in which intervals are ordered by left endpoint, we can easily query the range of intervals for which $l(v) < r(t)$. For any of these intervals to intersect t there must therefore be one such that $l(t) < r(v)$. The existence of any such interval means that the rightmost of these intervals must also intersect t, and therefore that if the rightmost interval does not intersect t, then none of them may. We can find the rightmost interval in this range by taking the maximum right endpoint. Therefore if we instead use a segment tree which aggregates elements by taking the maximum of its children, we can select some $p \in \{v \in V_P \mid v \not\ni t\}$ using a single segment tree query. This means we can both find an neighbour of t in V_P and insert t into V_P in logarithmic time.

If we represent each of S and V_P with a segment tree, we can therefore add each vertex to the tree in logarithmic time. Using a minimum segment tree we can select the leftmost remaining neighbour of either p or $\bigcup V_P$ from S in logarithmic time. Using a maximum segment tree we can select an arbitrary neighbour of t from V_P in logarithmic time. The selected vertex t can then be removed from S and inserted into V_P both in logarithmic time. Since Algorithm 8 will repeat this process to add one interval at a time to the tree until none remain, this gives us an overall time complexity of $O(|I| \log |I|)$.

Given the counterexample to Li *et al.*'s algorithm presented in Section 5.2, this is the only polynomial-time interval MIST algorithm of which I am aware.

Chapter 6

Conclusion

Geometric intersection graphs have applications ranging from bioinformatics and chemistry to automated circuit layout, scheduling, and infrastructure optimisation. I examined several problems relating to geometric intersection graphs, and have presented solutions to these problems.

Polygon-circle graph recognition has been a long standing problem of interest [33], despite having been shown to be NP-complete [43]. Though they are typically described by their geometric interpretation, polygon-circle graphs also have an equivalent alternating sequence representation [4]. This alternating sequence representation suggests a simple brute-force enumeration method for finding a representation of a given polygon-circle graph. I presented this method in Chapter 2, along with two more efficient methods. By analysing the properties of alternating sequences, we see that there are many prefixes that are behaviourally equivalent and interchangeable with each other, such that exploring a prefix that is equivalent to one we have already seen is a waste of time. I exploit this property in the design of a pair of more efficient algorithms for this problem. The first is a dynamic programming solution based on memoising these states to shortcut exploring states that we already know do not lead to a representation of the graph. This method is substantially faster than brute force, but requires so much memory for the memoisation table as to be intractable for any interesting graph. Instead I use pseudocyclic automata to construct a recognition automaton for alternating sequence representations of the desired graph. By optimising the automaton to combine behaviourally equivalent states at each point in the construction, the final recognition automaton uses the minimum possible number of states. Tracing the states of the resulting automaton allows us to construct an alternating sequence representation, and hence a polygon-circle intersection representation of the desired graph. While in the worst case this algorithm requires $O(\sqrt{8}^{|V|^2-|V|}|V|^2)$ time and $O(\sqrt{8}^{|V|^2-|V|})$ memory, this optimisation means that in practice the algorithm requires much less time and memory than these upper bounds would suggest. To my knowledge this is the first practical algorithm for polygon-circle graph recognition. With further development this method may be generalisable to other classes of graphs that permit sequence representations, expanding its practical applications.

The above polygon-circle graph recognition algorithm is fast enough to enable an exhaustive search to find the smallest non-polygon-circle graph, the 3-prism. In Chapter 3 I developed a constructive proof that the 3-prism is not a polygon-circle graph, but that it is an interval filament graph. In combination with the known proof [19] that polygon-circle graphs are a subclass of interval filament graphs, this serves as a constructive proof that they are specifically a proper subclass. While this

result was already known, the previous proof relied on analyses of the structure of random graphs in the limit to infinity, and so was not constructive [29]. This is the first constructive proof of this property, and with further development its intermediate results may have applications in proofs of other graph subclass relationships.

In Chapter 4 I examined the problem of minimising the average shortest path length in circular arc graphs by inserting a new edge from a set of candidates S. This problem can be solved naively by simply recomputing the average path length for each new edge. I presented a pair of solutions that work in general graphs, including an algorithm of my own design based on precomputing weight thresholds at which the new edge becomes better than an alternative path. This algorithm is a more natural fit for directed graphs, but was able to be generalised to undirected graphs while remaining $O(|V|^3 \log|V| + |V||S| \log|V|)$. In circular-arc graphs specifically, using an $O(|V|^2)$ all-pairs shortest path algorithm by Saha, Pal, and Pal [45] makes the naive approach just $O(|S||V|^2)$, but this is still slower than the threshold algorithm for large numbers of candidate edges. This method is also applicable to interval graphs, as well as any other subclasses of circular-arc graphs, and of course the threshold algorithm is applicable in general directed and undirected graphs.

Several optimisation problems that are NP-hard in generals graphs are known to have polynomial-time solutions in interval graphs, such as the Hamiltonian path problem [39]. The maximum internal spanning tree (MIST) problem is a generalisation of the Hamiltonian path problem, as if a Hamiltonian path exists it is also a spanning tree with the maximum possible number of internal vertices. Li, Feng, Jiang, and Zhu [38] present what they claim to be a polynomial-time algorithm for MIST in interval graphs. In Chapter 5, I presented a counterexample that shows that Li *et al.*'s algorithm and proof of correctness for that algorithm are incorrect. I then used the problem's relationship to the Hamiltonian path problem to extend a polynomial-time greedy Hamiltonian path algorithm for interval graphs by Manacher, Mankus, and Smith [39] to find MISTs in interval graphs. I proved the correctness of this algorithm and showed that it can be implemented in $O(|I| \log |I|)$ time, making it the only polynomial-time interval MIST algorithm of which I am aware.

Geometric intersection graphs have proved to be a powerful tool for modelling a wide range of systems and solving valuable problems in informatics, design, and optimisation. I have presented a number of problems in this field, and developed novel solutions to these problems that extend the state of the art. With further work these results may not only find practical applications, but may serve as a basis for other new developments in this field.

Bibliography

[1] S. R. Arikati and C. P. Rangan. Linear algorithm for optimal path cover problem on interval graphs. *Information Processing Letters*, 35(3):149–153, 1990.

[2] D. Best and M. Ward. A faster algorithm for maximum independent set on interval filament graphs. *arXiv preprint arXiv:2110.04933*, 2021.

[3] P. Bonsma and F. Breuer. Counting hexagonal patches and independent sets in circle graphs. *Algorithmica*, 63:645–671, 2012.

[4] A. Bouchet. Circle graph obstructions. *J. Comb. Theory, Ser. B*, 60(1):107–144, 1994.

[5] J. Bubenzer. Cycle-aware minimization of acyclic deterministic finite-state automata. *Discrete Applied Mathematics*, 163:238–246, 2014.

[6] K. Cameron. Induced matchings in intersection graphs. *Discrete Mathematics*, 278(1-3):1–9, 2004.

[7] K. Cameron and C. T. Hoàng. On the structure of certain intersection graphs. *Information processing letters*, 99(2):59–63, 2006.

[8] K. Cameron, R. Sritharan, and Y. Tang. Finding a maximum induced matching in weakly chordal graphs. *Discrete Mathematics*, 266(1-3):133–142, 2003.

[9] M.-S. Chang, S.-L. Peng, and J.-L. Liaw. Deferred-query: an efficient approach for some problems on interval graphs. *Networks: An International Journal*, 34(1):1–10, 1999.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[11] D. G. Corneil, S. Olariu, and L. Stewart. The lbfs structure and recognition of interval graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1905–1953, 2010.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] G. Durán, L. N. Grippo, and M. D. Safe. Structural results on circular-arc graphs and circle graphs: a survey and the main open problems. *Discrete Applied Mathematics*, 164:427–443, 2014.

[14] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[15] T. French, A. Gozzard, and M. Reynolds. A modal aleatoric calculus for probabilistic reasoning. In *Indian Conference on Logic and Its Applications*, pages 52–63. Springer, 2019.

[16] T. French, A. Gozzard, and M. Reynolds. A modal aleatoric calculus for probabilistic reasoning: extended version. *arXiv preprint arXiv:1812.11741*, 2018.

[17] T. French, A. Gozzard, and M. Reynolds. Aleatoric dynamic epistemic logic for learning agents. In *Pacific Rim International Conference on Artificial Intelligence*, pages 433–445. Springer, 2019.

[18] T. French, A. Gozzard, and M. Reynolds. Dynamic aleatoric reasoning in games of bluffing and chance. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1964–1966, 2019.

[19] F. Gavril. Maximum weight independent sets and cliques in intersection graphs of filaments. *Information Processing Letters*, 73(5-6):181–188, 2000.

[20] E. Gioan, C. Paul, M. Tedder, and D. Corneil. Practical and efficient circle graph recognition. *Algorithmica*, 69(4):759–788, 2014.

[21] M. C. Golumbic. Graph theoretic models for reasoning about time. In *Annual Asian Computing Science Conference*, pages 352–362. Springer, 2004.

[22] M. C. Golumbic, D. Rotem, and J. Urrutia. Comparability graphs and intersection graphs. *Discrete Mathematics*, 43(1):37–46, 1983.

[23] A. Gozzard. gozzarda/interval_mist: v1.0.0, version v1.0.0, Apr. 2023. DOI: 10.5281/zenodo.7797218. URL: https://doi.org/10.5281/zenodo.7797218.

[24] A. Gozzard. pcg_dfa, version v0.1.2, Apr. 2023. DOI: 10.5281/zenodo.7797177. URL: https://doi.org/10.5281/zenodo.7797177.

[25] A. Gozzard, M. Ward, and A. Datta. Converting a network into a small-world network: fast algorithms for minimizing average path length through link addition. *Information Sciences*, 422:282–289, 2018.

[26] K. C. Ho and S. B. Vrudhula. Interval graph algorithms for two-dimensional multiple folding of array-based vlsi layouts. *IEEE transactions on computer-aided design of integrated circuits and systems*, 13(10):1201–1222, 1994.

[27] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[28] R.-W. Hung, M.-S. Chang, and C.-H. Laio. The hamiltonian cycle problem on circular-arc graphs. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 1, pages 18–20, 2009.

[29] S. Janson and J. Kratochvıl. Thresholds for classes of intersection graphs. *Discrete Mathematics*, 108(1-3):307–326, 1992.

[30] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005.

[31] J. Köbler, S. Kuhnert, and O. Watanabe. Interval graph representation with given interval and intersection lengths. *Journal of Discrete Algorithms*, 34:108–117, 2015.

[32] M. Koebe. Colouring of spider graphs. In *Topics in Combinatorics and Graph Theory*, pages 435–441. Springer, 1990.

[33] M. Koebe. On a new class of intersection graphs. In *Annals of Discrete Mathematics*. Volume 51, pages 141–143. Elsevier, 1992.

[34] A. Kostochka and J. Kratochvıl. Covering and coloring polygon-circle graphs. *Discrete Mathematics*, 163(1-3):299–305, 1997.

[35] M. Y. Kovalyov, C. T. Ng, and T. E. Cheng. Fixed interval scheduling: models, applications, computational complexity and algorithms. *European journal of operational research*, 178(2):331–342, 2007.

[36] J. Kratochvıl and M. Pergel. Two results on intersection graphs of polygons. In *International Symposium on Graph Drawing*, pages 59–70. Springer, 2003.

[37] P. Li, J. Shang, and Y. Shi. A simple linear time algorithm to solve the mist problem on interval graphs. *Theoretical Computer Science*, 930:77–85, 2022.

[38] X. Li, H. Feng, H. Jiang, and B. Zhu. Solving the maximum internal spanning tree problem on interval graphs in polynomial time. *Theoretical Computer Science*, 734:32–37, 2018.

[39] G. K. Manacher, T. A. Mankus, and C. J. Smith. An optimum $\Theta(n \log n)$ algorithm for finding a canonical hamiltonian path and a canonical hamiltonian circuit in a set of intervals. *Information Processing Letters*, 35(4):205–211, 1990.

[40] S. Mandal, A. Pal, and M. Pal. An optimal algorithm to find centres and diameter of a circular-arc graph. *Advanced Modeling and Optimization*, 9(1):155–170, 2007.

[41] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[42] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied mathematics*, 35(1):68–82, 1978.

[43] M. Pergel. Recognition of polygon-circle graphs and graphs of interval filaments is np-complete. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 238–247. Springer, 2007.

[44] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[45] A. Saha, M. Pal, and T. K. Pal. An optimal parallel algorithm for solving all-pairs shortest paths problem on circular-arc graphs. *Journal of Applied Mathematics and Computing*, 17:1–23, 2005.

[46] G. Salamon and G. Wiener. On finding spanning trees with few leaves. *Information Processing Letters*, 105(5):164–169, 2008.

[47] N. A. Sherwani. *Algorithms for VLSI physical design automation*. Springer Science & Business Media, 2012.

[48] J. Spinrad. Recognition of circle graphs. *Journal of Algorithms*, 16(2):264–282, 1994.

[49] M. Ward and A. Datta. Personal Communication, 2016.

[50] M. Ward, A. Gozzard, and A. Datta. A maximum weight clique algorithm for dense circle graphs with many shared endpoints. *Journal of Graph Algorithms and Applications*, 21(4):547–554, 2017.

[51] M. Ward, A. Gozzard, M. J. Wise, and A. Datta. A faster algorithm for maximum induced matchings on circle graphs. *J. Graph Algorithms Appl.*, 22(2):389–396, 2018.

[52] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.