A Faster Algorithm for Recognizing Edge-Weighted Interval Graphs

by

Shikha Mahajan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Abstract

Interval graphs—the intersection graphs of one dimensional intervals—are considered one of the most useful mathematical structures to model real life applications [18]. Interval graphs have been widely studied since they first appeared in the literature in 1957. In 1976, Booth and Lueker [7] introduced a data structure called PQ-trees that could recognize interval graphs in linear time; since then, several simpler linear-time algorithms have been proposed for the problem.

We investigate a lesser-studied variation of interval graphs called edge-weighted interval graphs. A graph with weights on its edges is an edge-weighted interval graph if we can assign intervals to its vertices so that the weight of an edge (u, v) is equal to the length of the intersection of the intervals assigned to u and v. In 2012, Köbler, Kuhnert, and Watanabe [28] gave an algorithm to recognize such graphs in time $O(m \cdot n)$, where m and n are the number of edges and vertices, respectively, of the given graph. In this thesis, we give an algorithm to recognize complete edge-weighted interval graphs in time $O(m \cdot \log n)$. We then observe some additional properties of PQ-trees for interval graphs, and use these properties to improve the runtime of the algorithm given by Köbler et al. [28] for recognizing general edge-weighted interval graphs to $O(m \cdot \log n)$. As the literature for finding representations of weighted intersection graphs is scarce, we hope that the techniques presented in this thesis can be used to obtain algorithms or approximation algorithms for recognition of other kinds of weighted intersection graphs.

Acknowledgements

First and foremost, I would like to thank my supervisor, Anna Lubiw, for her excellent guidance, encouragement and patience. Not only was I inspired by her expertise in research, but also by her multifaceted personality and optimistic disposition. I would also like to thank my readers Naomi Nishimura and Jonathan Buss for carefully inspecting my thesis and providing feedback.

I thank my family—Maneesh Mahajan, Nisha Mahajan, and Priyam Mahajan—for believing in me and for their abundant support. I also thank Nicole Keshav for helping me gain confidence in my writing. Last, but not the least, I thank my friends for making the journey of attaining my Master's a fun and interesting experience.

Dedication

This thesis is dedicated to my parents, for their limitless love, support, and encouragement.

Table of Contents

List of Figures

Chapter 1

Introduction

Ever since Hajós [20] introduced them in 1957, interval graphs have been widely studied due to their diverse applications. In this thesis, we study a variation of interval graphs called *edge-weighted interval graphs*. We first provide some background on the topic, and then describe the problem addressed in the thesis.

1.1 Interval graphs

A graph G is an interval graph if each vertex in the graph can be assigned to an interval such that two vertices are connected in the graph if and only if their corresponding intervals overlap. Figure 1.1 gives an example of an interval graph.

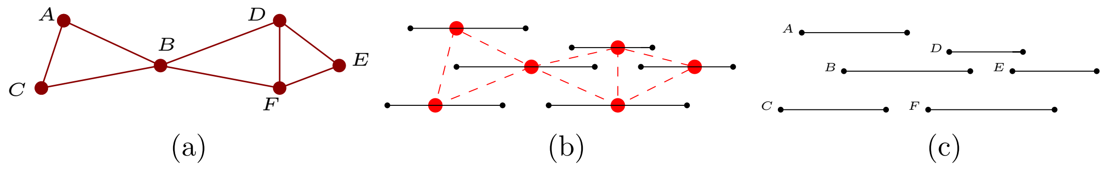


Figure 1.1: Figure 1.1a: An interval graph G; Figure 1.1b: G overlaid on its interval representation; Figure 1.1c: Interval Representation of G

Note that not all graphs can be represented as intervals as described above. Figure 1.2 gives examples of graphs that are not interval graphs.

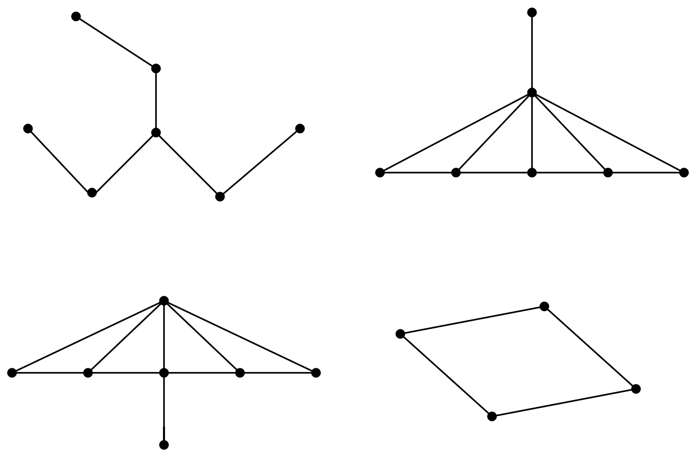


Figure 1.2: Some graphs that cannot be represented as intervals. Note: Adapted from 'Representation of a finite graph by a set of intervals on the real line' by Lekkerkerker and Boland [33]

The *interval graph recognition problem* determines if a given graph G is an interval graph, and if so, gives an interval representation of G.

In 1962, Lekkerkerker and Boland [33] gave two characterizations of interval graphs, one of which states that a graph is an interval graph if and only if it is chordal and free of asteroidal triples. Based on this characterization, they gave an $O(n^4)$ time algorithm for recognizing interval graphs. Gilmore and Hoffman [16] proved that the complement of an interval graph is a comparability graph, and used this to give an algorithm for the problem. Fulkerson and Gross [14] proved that a graph is an interval graph if and only if its maximal clique versus vertex incidence matrix observes the consecutive-ones property; they also gave an algorithm for the problem based on this characterization.

All above algorithms have worst case running time of at least $O(n^3)$. The first linear time algorithm for recognizing interval graphs was given by Booth and Lueker [7] in 1976. They used the characterization of interval graphs given by Fulkerson and Gross [14] and designed a data structure called PQ-trees to check if the maximal clique versus vertex incidence matrix of a graph satisfied the consecutive-ones property. As PQ-trees are difficult to implement in practice, several simpler algorithms have since been proposed for the problem. Korte et al. [29] solve the problem using an incremental algorithm based on modified PQ-trees which can be used for online recognition of graphs, i.e., when the entire graph is not known in advance and vertices are added one at a time. Simon [42] gave an algorithm that orders maximal cliques without using PQ-trees by running lexBFS four times. Habib et al. [19] solve the problem using modified lexBFS and clique trees. Hsu [22] provides a solution that does not require ordering maximal cliques. Corneil, Olariu, and Stewart [9] provide another algorithm to recognize interval graphs without ordering maximal cliques by utilizing the properties of asteroidal triples.

Interval graphs have diverse applications such as seriation of artifacts in archaeology [27], sequencing clones in DNA [17], job scheduling and resource allocation [8], modelling food webs [38] and so on. Moreover, the recognition of interval graphs is useful for solving optimization problems such as finding minimum vertex cover, maximum independent set, largest clique, minimum colouring etc. as these problems are solvable in polynomial time [15] on interval graphs.

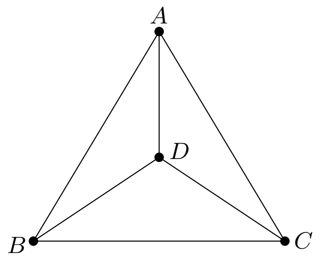
We now look at a natural variation of interval graphs, namely, the *edge-weighted interval graphs*, that are the focus of this thesis.

1.2 Edge-Weighted Interval Graphs

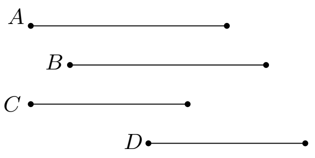
A given graph with weights on its edges is an *edge-weighted interval graph* if we can assign intervals to the vertices such that the weight of an edge (u, v) is equal to the length of the intersection of the intervals assigned to u and v. We investigate the problem of recognizing edge-weighted interval graphs, hence our problem is formally defined as: *Determine if a given edge-weighted graph G is an edge-weighted interval graph and if so, give an interval representation for G.*

Note that a given graph can be an interval graph but not an edge-weighted interval graph; one such example is described below. Graph G in Figure 1.3 is an interval graph as it can be represented as intervals, but the weighted version of G given in Figure 1.4a is not an edge-weighted interval graph as we will see next.

Consider the edge-weighted version G_w of G in Figure 1.4a. Let us try to obtain an edge-weighted interval representation of G_w. First, we obtain the smallest possible (minimal) interval representation of the subgraph ABC, and then we try to place interval D in this representation. Since the interval representation of ABC is the smallest possible, this representation must be present in all edge-weighted interval representations of G. To satisfy the intersection of D with C, we can either place D completely inside interval C (see Figure 1.4b), or place it such that it protrudes from C, either from the left or from the right (see Figure 1.4c). Observe that in all cases it is impossible for D to satisfy its intersection with both A and B without violating its intersection with C. Hence, G_w is not an interval graph.

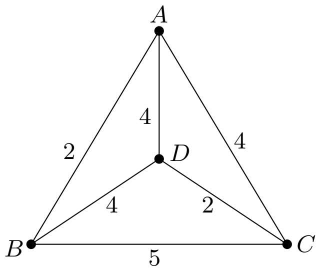


(a) Unweighted Graph G

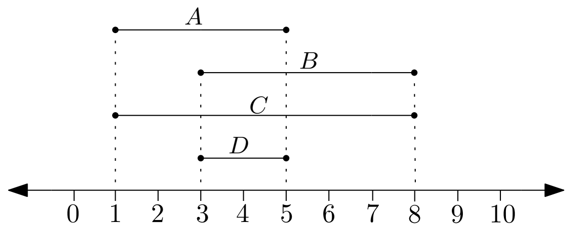


(b) Interval Representation of G

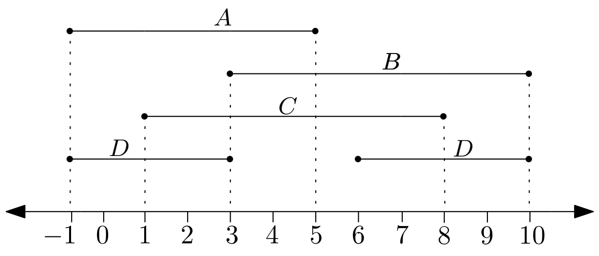
Figure 1.3: Interval Representation of unweighted graph G



(a) Weighted Graph G_w



(b) Placing D such that it is contained in C



(c) Placing D such that it protrdues from C, either on the left or on the right

Figure 1.4: G_w is not an edge weighted interval graph

The literature on recognition of edge-weighted interval graphs is not as abundant as for the recognition of interval graphs. The problem was investigated by Yamamoto [45] in 2007. In 2012, Köbler, Kuhnert, and Watanabe [28] gave an algorithm for recognizing edge-weighted interval graphs in $O(m \cdot n)$ time.

We first give a simple algorithm that solves the edge-weighted interval representation problem for complete graphs in $O(m \cdot \log n)$ time. Then we use this algorithm along with properties of PQ-trees to improve the running time of Köbler's algorithm for general graphs to $O(m \cdot \log n)$.

Outline of the Thesis

The remainder of this chapter discusses related work. Chapter 2 introduces the notation and describes the concepts used in the thesis. In Chapter 3, we give an intuition of how different components of our algorithm work together. Chapter 4 presents an algorithm for finding an edge-weighted interval representation of a complete graph. In Chapter 5 we describe how properties of PQ-trees with the algorithm from Chapter 4 improves the running time of the algorithm for general graphs proposed by Köbler et al. [28]. Chapter 6 concludes the thesis and suggests directions for future work.

1.3 Related Work

In this section we mention other problems in the literature that place constraints on interval representation of graphs and we describe some generalizations of interval graphs.

1.3.1 Interval Graph Representation with Constraints

Several variations of the interval graph representation problem with constraints can be found in the literature. In 1965, Fulkerson and Gross [14] gave an algorithm for finding the interval representation of graphs with given interval lengths and pairwise intersection lengths in $O(n^2)$ time. Pe'er and Shamir [37] have proven the NP-completeness of finding the interval representation of a graph when interval lengths are specified. Skrien [43] gives a $O(n^3)$ algorithm for finding the interval representation when the endpoints of the intervals must be placed according to a given partial order. The *seriation with side constraints* problem, where the relative position of two disconnected intervals is specified, is solvable in linear time [37]. The *interval graph with distance constraints* problem, to find an interval representation respecting the specified distance between the end points of the intervals, is solved in quadratic time by Shamir et al. [37]. Jampani and Lubiw in [24] give a $O(m \cdot \log n)$ algorithm to find the simultaneous interval representations of two graphs, where each vertex present in both graphs must be represented by the same interval.

1.3.2 Intersection Graphs

An intersection graph representing a set of objects has a vertex corresponding to an object with the rule that two vertices are adjacent if their corresponding objects intersect. Interval graphs are therefore intersection graphs of intervals. In this section we discuss known results on some other types of intersection graphs that are related to interval graphs.

Boxicity Graphs

Boxicity graphs are the intersection graphs of d-dimensional intervals, i.e. axis aligned d-dimensional boxes. Interval graphs are therefore boxicity-1 graphs. The recognition of boxicity-d graphs with $d \geq 2$ is NP-complete [31]. Figure 1.5 shows the intersection of two-dimensional intervals.

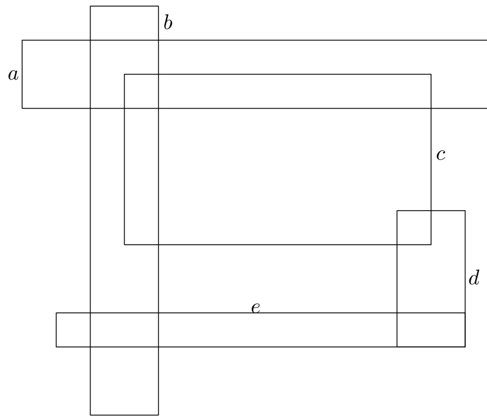


Figure 1.5: Intersection of two-dimensional intervals

Circular-Arc Graphs

Circular-arc graphs are the intersection graphs of arcs on a circle as shown in Figure 1.6. The recognition problem was conjectured to be NP-complete by Booth [6], but was later proven polynomial by Tucker [44] with an $O(n^3)$ algorithm. The algorithm was then improved to $O(m \cdot n)$ by Hsu [23], where m, n represent the number of edges and vertices respectively. This was later improved to $O(n^2)$ by Eschen and Spinrad [13]. The first linear time algorithm was given by McConnell [34]. In [26], Kaplan gives a simpler algorithm for recognizing circular-arc graphs in linear time.

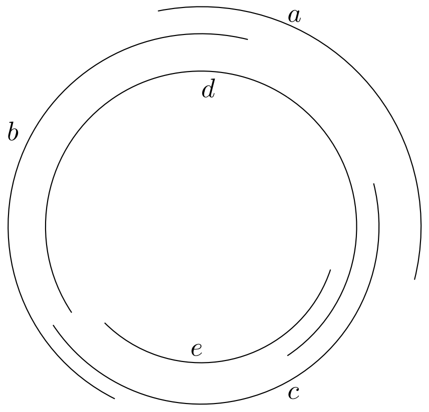


Figure 1.6: Intersection of arcs on a circle

Contact Graphs

Each vertex of a contact graph G corresponds to an object such as a line segment or polygon in space, and an edge between two vertices in G signifies that the objects corresponding to the two vertices touch, but do not cross, in a predefined manner. Alam et al. in [2, 1] study proportional contact graphs of polygons where the area of the object is specified as weights on the vertices. In [3], Alam et al. study proportional contact representation of $3D$ objects composed of boxes. De Fraysseix and de Mendez [10] study the contact graphs of segments and pseudo-segments and give partial results on representing all planar graphs by the intersection of pseudo-segments. Recognition of contact graphs of line segments with contact degrees of three or more was proven to be NP-complete by Hliněnỳ [21].

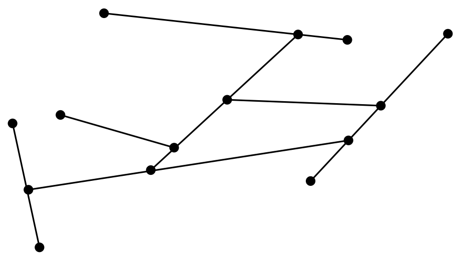


Figure 1.7: Contact of Segments

String Graphs

A string graph is the intersection graph of a set of curves in the plane. Each vertex of a string graph corresponds to a curve in the plane, and two vertices are connected if their corresponding curves intersect. In 1991, Kratochvíl [30] proved that the recognition of string graphs is *NP*-hard. Schaefer and Štefankovič [41] show that string graphs can be recognized in non-deterministic exponential time. Later, in 2003, Schaefer [40] et al. showed the recognition of string graphs is *NP*-complete by showing that the problem is in *NP*.

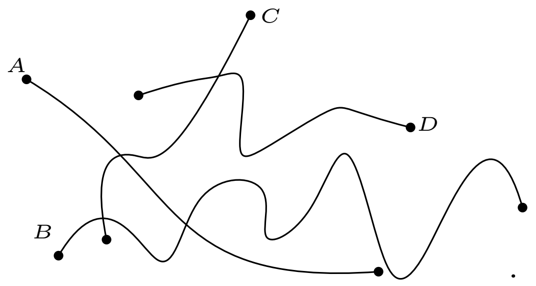


Figure 1.8: Intersection of Curves

Segment Graphs

A segment graph is the intersection graph of straight line segments in a plane. Segment graphs are a proper subset of string graphs [12]. Kratochvíl and Matoušek [32] have shown that the recognition of segment graphs is in PSPACE. They also show that the problem is *NP*-complete when the segments are forced to lie in k directions for every $k \geq 2$.

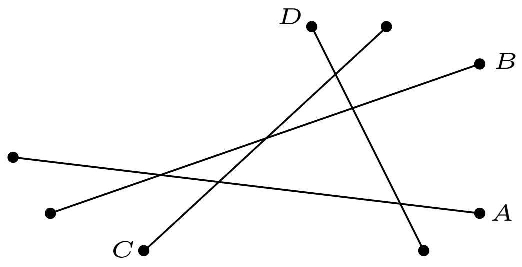


Figure 1.9: Intersection of Segments

Chapter 2

Preliminaries

This chapter introduces the notation used in the thesis and provides the reader with background on PQ-trees.

2.1 Notation Used

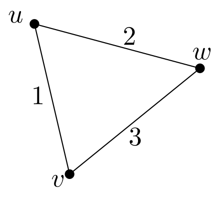
For a given graph G, we denote the set of vertices of G by $V(G)$, and the set of edges of G by $E(G)$. The size of $V(G)$ and $E(G)$ is denoted by n and m, respectively. We use I_v to represent the interval corresponding to a vertex v. The left and right points of I_v are denoted by $l(I_v)$ and $r(I_v)$, respectively.

Definition 2.1. A *minimal* interval representation of an edge-weighted graph is one in which no interval can decrease in size and still represent the given graph (see Figure 2.1).

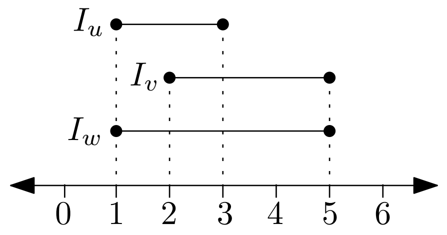
In the following sections we describe some concepts that are fundamental for recognizing interval graphs.

2.2 Maximal Cliques and Consecutive-Ones Property

A maximal clique of a graph is a clique that is not a subset of any larger clique of the graph. For a graph to be an interval graph, there must exist a linear ordering of its maximal cliques such that all cliques containing a vertex are consecutive [7]. A binary matrix is



(a) Graph G



(b) Minimal Interval Representation M of G

Figure 2.1: A minimal interval representation M of G. Notice that it is impossible to reduce the size of any interval in M and still satisfy all edge-weights of G

said to have the *consecutive-ones property* when we can permute its columns such that the ones in every row are consecutive. The clique matrix M of a graph is a matrix where the columns correspond to the maximal cliques of the graph, the rows correspond to its vertices, and M_{ij} is defined as:

$$M_{ij} = \begin{cases} 1, & \text{if clique } C_j \text{ contains the vertex } v_i \\ 0, & \text{if clique } C_j \text{ does not contain vertex } v_i \end{cases} \tag{2.1}$$

A graph is an interval graph if and only if its clique matrix M has the consecutive-ones property [14].

A chordal graph is a graph that does not contain induced cycles of length four or more; such a graph can be recognized in linear time by using a technique called lexBFS [39]. Since we cannot represent cycles of length more than three as intersection of intervals, every interval graph must be a chordal graph. Therefore, we first check if G is a chordal graph, and if it is not, then we know that G is not an interval graph. Otherwise, we proceed to find its maximal cliques.

An arbitrary graph may have an exponential number of maximal cliques; however, since G is a chordal graph, it has a linear number of maximal cliques, that can be listed in $O(m)$ time [15]. Therefore, we obtain the clique matrix M of G, and test if M satisfies the consecutive-ones property.

Booth and Lueker [7] in 1976 introduced a data structure called PQ-trees to check if a $m \times n$ matrix satisfies the consecutive-ones property. Their algorithm runs in $O(m+n+f)$ time, where f is the number of non zero entries in the matrix. Hence we can determine if a graph is an interval graph in linear time using PQ-trees. The following section describes some more properties of PQ-trees.

2.3 PQ-Trees

The columns of the matrix correspond to the leaves of the PQ-tree.

The immediate descendants of an internal node of a PQ-tree can be reordered by some rules (detailed below). For any such reorderings, the ordering of the leaves of the PQ-tree is called its *frontier*. Booth and Lueker [7] prove that the possible frontiers are in one-to-one correspondence with the orderings of the columns of a matrix that satisfy the consecutive-ones property.

PQ-trees are built of three types of nodes: P-nodes, Q-nodes and leaves. P-nodes are denoted by a circle and Q-nodes are denoted by a rectangle. The order of the children of a Q-node must be preserved up to reflection, i.e., we are only allowed to reverse the order of the children of a Q-node to maintain the consecutive-ones property of the frontier of the tree. On the other hand, the children of a P-node can be arranged in any order without violating the consecutive-ones property of the frontier.

(a) P Node, represented by a circle (b) Q-node, represented by a rectangle

Figure 2.2: Two types of non-leaf nodes of PQ-trees.

Two PQ-trees T_1 and T_2 are equivalent if we can obtain the frontier of T_2 from T_1 by:

- Arbitrarily permuting the children of its P-nodes
- Reversing the order of the children of its Q-nodes

As mentioned earlier, Booth and Lueker give a linear-time algorithm to construct a PQ-tree corresponding to 0-1 matrix M, if such a PQ-tree exists. Their algorithm works by starting with one P-node with all columns as leaves, and then considering the rows one by one, modifying the PQ-tree to ensure that the columns with the 1s in that row appear consecutively in any frontier.

In this thesis, we will need some properties of PQ-trees of interval graphs. Let G be an interval graph and consider a PQ-tree of G's clique matrix M. The leaves of the PQ-tree correspond to the maximal cliques of G. In particular, for any row r of the matrix, the columns with a 1 in row r must appear consecutively in any frontier of the PQ-tree.

We say that a vertex v *appears* in a node of a PQ-tree if it is in at least one maximal clique corresponding to a leaf in the subtree rooted at the node. A vertex w *covers* a node N if it is in all maximal cliques corresponding to the leaves descended from N. We claim that any P-node in the PQ-tree of a connected graph is covered by at least one vertex.

Claim 2.1. For each P-node N_p in a PQ-tree of a connected graph G, there exists a vertex $v \in V(G)$ that covers N_p.

Proof. We know that the children of a P-node N_p can be arranged in any order. This implies that any vertex either appears in only one child of N_p, or it covers N_p. Since we are dealing with connected components, there must be at least one vertex v that covers N_p. □

The corollary below follows from the above claim:

Corollary 2.1. Any vertex v that appears in a clique corresponding to a leaf of a P-node N_p either covers N_p or appears in only one child of N_p.

For a Q-node N_q, we prove that if a vertex v appears in more than one child of N_q, then v covers each child of N_q that it appears in. To prove this claim, consider a node N_q. Let the ordering of the children of N_q be denoted by O_{N_q}. Recall that the ordering $O_{N_q} = N_1, \cdots, N_t$ is unique up to reflection.

Definition 2.2. A vertex v is *long* for a Q-node N_q if it appears in multiple children of N_q.

Claim 2.2. If a vertex w is long for a Q-node N_q, then w covers each $N_i \in O_{N_q}$ that it appears in.

Proof. The proof is trivial when N_i is a leaf of the PQ-tree.

Suppose w appears in N_i—assume w is in clique C_1 corresponding to a leaf of N_i. Since w is long, it is in a clique C_2 corresponding to a leaf of some N_j, $j \neq i$.

Let C_3 be a clique corresponding to a leaf of N_i, $C_3 \neq C_1$. We must show that w is in C_3. Take one ordering of the PQ-tree, and suppose that C_1 appears before C_2 in the frontier. Then all the descendants of N_i appear before C_2 in the frontier. Irrespective of whether N_i is a P-node or a Q-node, we may reverse the order of its descendants using valid PQ-node reorderings, resulting in a new frontier. In one of these two frontiers, our three cliques appear in the order C_1, C_3, C_2. By the consecutive-ones property, since w is in C_1 and C_2, it is also in C_3. □

Corollary 2.2 below follows from the above proof:

Corollary 2.2. The long vertices of a Q-node form an interval system on its immediate children $N_1, \cdots, N_t$.

2.4 Weighted Helly Theorem For Intervals

Helly's theorem [4] states that if we have a collection of n convex subsets of R^d with $n > d$ such that the intersection of every $d+1$ of these subsets is non-empty, then the entire collection has a nonempty intersection. As we are dealing with weighted graphs, our solution uses a quantitative version of Helly's theorem which gives us information about the size of the common intersection between all intersecting subsets. Bárány, Katchalsky, and Pach [4] give a lower bound on the size of the common intersection in any dimension d. For interval graphs we are concerned with the simple case of $d = 1$, for which an exact result holds. Since we have not found this result in the literature, we give a proof for it in Theorem 2.1.

Theorem 2.1. *For a set $S = \{I_1, I_2, .., I_n\}$ of n intervals with the property that $|I_i \cap I_j| \geq t, \forall i, j \in [1, n]$, there exists a common intersection of length at least t between all the intervals in S.*

Proof. We prove the theorem by induction on n intervals.

Base case: When $n = 2$, the proof is trivial.

Let us assume that the theorem holds for $n-1$ intervals. Let C be the common intersection of $n-1$ intervals, $I_1, \cdots, I_{n-1}$ in S. Then $|C| \geq t$ by the induction hypothesis. We must prove that $|I_n \cap C| \geq t$.

Two intervals A and B have $|A \cap B| \geq t$ if the following conditions are satisfied:

1. $r(A) - l(B) \geq t$

2. $r(B) - l(A) \geq t$
3. $r(A) - l(A) \geq t$
4. $r(B) - l(B) \geq t$

The conditions follow from the observation that two intervals overlap by at least t if and only if each of the left end points of the two intervals are at least distance t before each of the right endpoints.

We will prove the above conditions for I_n and C.

As C represents the common intersection of $n-1$ intervals, there exists an interval I_j that starts at $l(C)$ and an interval I_k that ends at $r(C)$, i.e. $l(C) = l(I_j)$ and $r(C) = r(I_k)$. Since $|I_k \cap I_n| \geq t$, therefore $r(I_k) - l(I_n) \geq t$, and hence $r(C) - l(I_n) \geq t$. Similarly, since $|I_j \cap I_n| \geq t$, therefore $r(I_n) - l(I_j) \geq t$, and hence $r(I_n) - l(C) \geq t$.

Note that $|C| \geq t$ by induction hypothesis, and hence $r(C) - l(C) \geq t$. Also note that $|I_n| \geq t$, otherwise the intersection of I_n with any interval cannot be t, and so $r(I_n) - l(I_n) \geq t$. We have shown that all four conditions are satisfied by I_n and C, hence $|I_n \cap C| \geq t$. $\square$

Chapter 3

Overview of the Algorithm

As mentioned in Chapter 1, Chapter 4 presents Algorithm 4.2 for obtaining the edge-weighted interval representation of complete graphs, which is used as a subroutine to improve the running time of Köbler's algorithm. In this chapter we explain the bottleneck of Köbler's algorithm, and then give an intuition of how Algorithm 4.2 together with PQ-trees can help in removing this bottleneck.

3.1 Bottleneck of Köbler's algorithm

In their paper, Köbler et al. [28] solve two variations of the interval graph representation problem: one where the edges of the given graph are weighted, and one where both the edges and the vertices of the given graph are weighted. The weight on a vertex, the *l-weight*, indicates the length of its corresponding interval, and the weight on an edge, the *s-weight*, represents the length of intersection of intervals of its adjacent vertices. They refer to a set of intervals respecting the weights on the edges as the *s-respecting interval representation* and a set of intervals respecting both the weights on the vertex and the edges as the (l, s)-respecting interval representation. Their algorithm finds an *(l, s)-respecting interval representation*, or determines if none exists, in $O(n + m)$ time. They use the same algorithm for finding the s-respecting intervals by reducing it to the (l, s)-respecting interval representation problem by assigning lengths to the intervals (and therefore l weights to the vertices of the graph) based on their s-weights. The method for computing the lengths of the intervals uses a relation called R_s, which we refer to as the *protrusion relation*. Their paper employs a $O(m \cdot n)$ time brute force algorithm to find the protrusion relation, which is the bottleneck of their algorithm.

To explain how the protrusion relation R_s is computed, we first define R_s and some related terms.

Definition 3.1. An edge $(u, v) \in R_s$ if $(u, v) \in E(G)$ and $\exists w \in V(G)$ such that $s(u, w) > min\{s(u, v), s(v, w)\}$.

If a vertex w satisfies the definition for $(u, v) \in R_s$, we say the w *witnesses* $(u, v) \in R_s$.

Next, we define the term *protrudes* for any two intervals I_u and I_v.

Definition 3.2. An interval I_u *protrudes* from an interval I_v if $I_u \cap I_v \neq \phi$ and $I_u \setminus I_v \neq \phi$. In other words, I_u protrudes from I_v if I_u and I_v intersect, and I_u contains a point p not in I_v (see Figure 3.1).

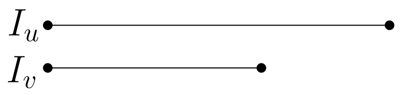


Figure 3.1: Interval I_u protrudes from interval I_v

For vertices u, v, we say that u *protrudes* from v if I_u protrudes from I_v in every minimal s-respecting interval representation of G.

In any minimal s-respecting representation, we say that vertex u protrudes from vertex v *on the left* if $l(I_u) < l(I_v)$. Likewise, u protrudes from v *on the right* if $r(I_u) > r(I_v)$.

In Lemma 3.1 of [28], Köbler et al. state and prove that an interval I_u protrudes from an interval I_v in all minimal s-respecting interval representations of G if and only if $(u, v) \in R_s$. For the convenience of the reader we prove the same in Lemma 3.1 below:

Lemma 3.1. *Let R be a minimal s-respecting interval representation of a graph G. Then edge $(u, v) \in R_s$ if and only if I_u protrudes from I_v.*

Proof. Suppose edge $(u, v) \in R_s$. Then, there exists a vertex $w \in V(G)$ such that $s(u, w) > s(u, v)$ or $s(u, w) > s(v, w)$; either way, in any s-respecting interval representation, there must exist at least one point outside I_v common to I_u and I_w, and hence I_u protrudes from I_v.

For the backward implication, suppose without loss of generality that I_u protrudes from I_v on the left, implying $l(I_u) < l(I_v)$. Since R is a minimal representation, there must be at least one interval I_w that shares a point with I_u to the left of $l(I_v)$. If $r(I_u) < r(I_w)$,

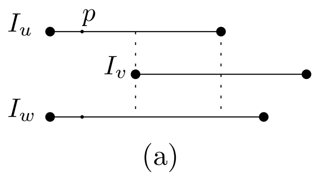


Figure 3.2: If u protrudes from v on the left, Figure 3.2a shows that if $r(I_u) < r(I_w)$, then $s(u, w)$ is as large as $s(u, v)$ plus one additional point p outside I_v; Figure 3.2b shows that if $r(I_u) \geq r(I_w)$ then $s(u, w)$ is as large as $s(w, v)$ plus one additional point p outside I_v; in both cases w satisfies the condition for $(u, v) \in R_s$

then $s(u, w)$ is as large as $s(u, v)$ plus one additional point outside I_v (see Figure 3.2a). Otherwise, if $r(I_u) \geq r(I_w)$ then $s(u, w)$ is as large as $s(w, v)$ plus one additional point (see Figure 3.2b). Either way it satisfies the condition for $(u, v) \in R_s$.

□

Note that the above proof implies the following corollaries:

Corollary 3.1. If I_u protrudes from I_v in some minimal edge-weighted interval representation of G, then $(u, v) \in R_s$. If $(u, v) \in R_s$, then I_u protrudes from I_v in any edge-weighted interval representation of G.

Corollary 3.2. If vertex w witnesses the protrusion of u from v by satisfying the condition for $(u, v) \in R_s$, then I_u protrudes from I_v in all s-respecting interval representations of any subgraph of G containing vertices u, v and w.

Corollary 3.3. Intervals I_u and I_w protrude from interval I_v on the same side in any minimal s-respecting interval representation if w witnesses the protrusion of u from v.

Proof. Note that if w witnesses the protrusion of u from v, then $s(u, w) > s(u, v)$ or $s(u, w) > s(v, w)$, and hence there must exist one point p common to I_u and I_w outside I_v in any minimal s-respecting representation of a subgraph of G containing u, v, w. If p is to the left of $l(I_v)$ then both u and w protrude from v on the left, and if p is to the right of $r(I_v)$ then both u and w protrude from v on the right; hence in either case both u and w must protrude from v on the same side in any s-respecting interval representation of G. □

Köbler et al. employ a brute force approach to compute R_s by searching for a witness for every edge by testing all vertices of G. The running time of their algorithm for determining R_s is therefore $O(m \cdot n)$.

3.2 Our Approach to Computing R_s

Algorithm 4.2 from Chapter 4 gives a minimal s-respecting representation of a given complete graph (or determines that none exists) in time $O(m \cdot \log n)$. Note that once we obtain a minimal s-respecting representation M of a graph G, we can obtain the protrusions of G by simply comparing the left and right end points of the intervals in M as justified by Corollary 3.1. Therefore, R_s can be found in $O(m \cdot \log n)$ for complete graphs. In Chapter 5, we find R_s for general graphs by using PQ-trees to find a subset of intervals that witness most of the protrusions, and find the remaining protrusions by running Algorithm 4.2 a constant number of times.

Chapter 4

Obtaining the Edge-Weighted Interval Representation of a Complete Graph

In this chapter we give an algorithm for finding a minimal edge-weighted interval representation for a given complete graph. In a later chapter we will rely on this algorithm to find the protrusion relation R_s for a complete graph. We begin by defining some terms in Section 4.1, and then describe the algorithm in Section 4.2. Subsection 4.2.1 gives the pseudocode, the proof of correctness can be found in Subsection 4.2.2, and finally the runtime analysis is provided in Subsection 4.2.3.

4.1 Definitions

In this section we define some terms that will be used by our algorithm.

Definition 4.1. Tentative Interval Representation: Intuitively, a *tentative interval representation* of a graph G is one that can be 'expanded' to a valid edge-weighted interval representation of G, if such exists. More precisely, a set of intervals $T = \{I_v^T : v \in V(G)\}$ is a *tentative interval representation* if the following condition holds: if G has an interval representation, then it has an interval representation $M = \{I_v^M : v \in V(G)\}$ such that $\forall v \in V(G), I_v^T \subseteq I_v^M$.

Definition 4.2. Free and fixed endpoints: At any time we have a tentative interval representation T of G which we modify by pulling the left points of some intervals further left and right points of some intervals further right. We may also *fix* some endpoints which means that they have reached their final position and cannot be changed. The endpoints that are not yet fixed are called *free*. All figures in this thesis use solid ends to show fixed endpoints and hollow ends to show free endpoints.

Definition 4.3. Free and fixed intervals: An interval is *fixed* when both its endpoints are fixed. An interval that is not fixed is *free*.

Definition 4.4. Free Subgraph: The *free subgraph* F is the subgraph of G induced by vertices with free intervals in T.

Definition 4.5. Uniform vertex: A vertex u in a graph G is a *uniform vertex* if the weight of all edges incident to u is the same. We will also apply this definition to the free subgraph F and will speak of u being uniform in F.

As G is a clique with positive edge-weights, all intervals in any edge-weighted interval representation of G must have a point p in common. In our algorithm, we give a minimal edge-weighted interval representation of G where $p = 1$. We define below the *rightmost representation* of G with $p = 1$.

Definition 4.6. Rightmost representation: In a *rightmost representation*, each interval is placed as far right as possible while still containing the point 1. Moving any interval towards the right in a rightmost representation will either violate its intersection with other intervals, or it will no longer contain the point 1. The algorithm described in this section will always give us the minimal rightmost representation of the graph. To further illustrate, Figure 4.1 shows several ways that three intervals can be arranged to satisfy the same intersections; our algorithm gives the minimal rightmost representation as shown in Figure 4.1d.

4.2 The Algorithm

The algorithm works by modifying and fixing the endpoints of the intervals of a tentative interval representation T to satisfy as many constraints as possible, until there are no free intervals remaining. At all times during the algorithm, the following invariants are maintained:

1. T is a tentative interval representation of the rightmost representation M of G.

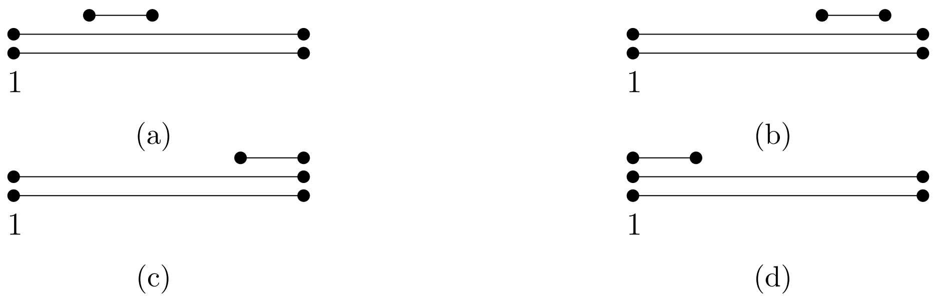


Figure 4.1: Several ways in which three intervals can be arranged; our algorithm computes the representation shown in 4.1d

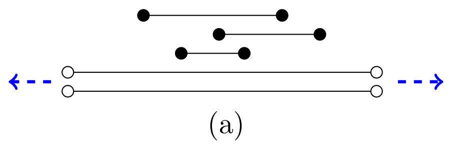
2. (a) All free left points $\leq$ fixed left points
   (b) All free right points $\geq$ fixed right points
3. All free left points are equal. All free right points are equal.

As mentioned earlier, since G is a clique with positive edge-weights, any interval representation M of G will contain one point which is common to all intervals. We arbitrarily choose 1 as the common point, thus initially T consists of n zero length intervals whose endpoints are free and set to 1.

The algorithm proceeds by applying the following cases:

Case 1: Both ends of all free intervals are free:
Pick an edge (i, j) having the smallest weight from the free subgraph F of G. By Theorem 2.1 (the weighted Helly theorem for intervals), any interval representation M of G must have a common subinterval of length $s(i, j)$ in the free intervals of T. Therefore, if the length of the free intervals is less than $s(i, j)$, we increase their lengths to $s(i, j)$ by pulling their right points. Notice that due to Invariant 1, the free endpoints are always outside the fixed endpoints, hence the choice of pulling either the left or the right points of free intervals will not impact the intersection of fixed intervals as shown in Figure 4.2a. However, in order to obtain the rightmost representation, we must pull the right points of the intervals (we will justify this in the proof of correctness).

In the selected edge (i, j), either at least one of i, j is a uniform vertex, or both i, j are non-uniform vertices. If the selected intersection is due to a *uniform* vertex u, fix the endpoints of I_u (see Figure 4.3b). We have now ensured that



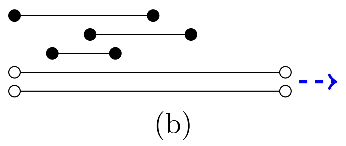
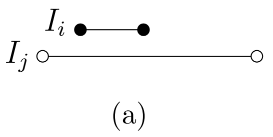


Figure 4.2: Observe in Figure 4.2a that pulling free intervals towards left or right has no impact on the intersection of fixed intervals; however, to obtain a rightmost representation, we pull the right endpoints to increase the length as shown by Figure 4.2b

I_u satisfies the intersections specified in G with its neighbours, and hence its length never needs to exceed $s(i,j)$. Otherwise, if the smallest weight is not due to a uniform vertex, we arbitrarily fix the left point of one of the intervals and the right point of the other interval. This ensures that the intersection of I_i and I_j is equal to $s(i,j)$, and cannot change later in the algorithm (see Figure 4.3).



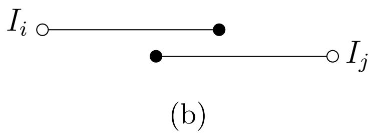


Figure 4.3: If i is a uniform vertex in (i,j), we fix its end points as shown in 4.3a. If both i,j are not uniform, then we arbitrarily pick one of I_i, I_j and fix its right point, and fix the left point of the other interval as shown in 4.3b.

Case 2: There exists an interval I_f in T with a fixed right point and a free left point:
Pull the left point of all intervals with free left points and $l(I_f)$ towards the left until all intervals satisfy the specified intersection with I_f. We will later show that if the left point of an interval needs to move right instead of left to satisfy the intersection, then G is not an edge-weighted interval graph. This process is demonstrated by Figure 4.5 for graph in Figure 4.4. After pulling all the intervals as required, we fix the left point of I_f as all the intersections with I_f have been satisfied, and hence $l(I_f)$ does not need to change.

We then run a routine called *fix_endpoints* to fix the left points of all intervals that start after any other interval (the pseudocode for *fix_endpoints* also fixes the right endpoints of all intervals that stop before any other interval, which will apply only for Case 3). The necessity of fixing the endpoints can be understood from Figure 4.6. Notice that moving $l(I_i)$ changes the intersection between I_i

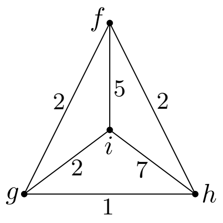


Figure 4.4: Graph G to be represented

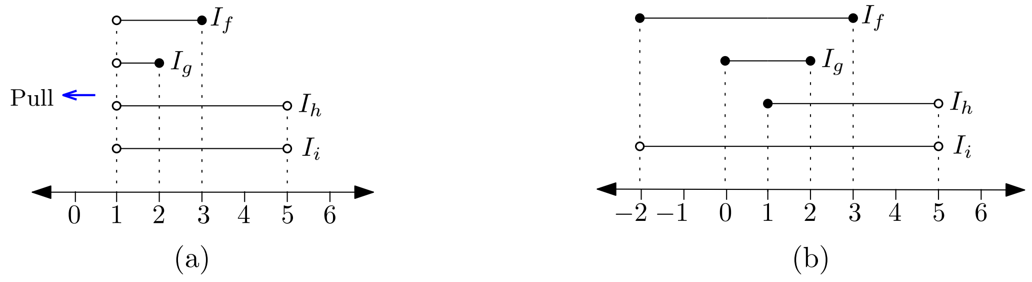


Figure 4.5: Figure 4.5a shows an initial tentative representation of G containing an interval I_f with a free left point and a fixed right. In Case 2, we select I_f, and then we satisfy the intersection of I_f with other intervals by pulling their left points, hence reaching the representation in Figure 4.5b. Observe that in Figure 4.5b, I_f satisfies its intersection with all other intervals as specified on G.

and I_m. Hence we must fix $l(I_i)$ if we wish to preserve the intersection between I_m and I_i.

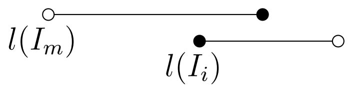


Figure 4.6: If some leftpoint $l(I_m)$ is to the left of $l(I_i)$ then we fix $l(I_i)$ as altering its position changes the intersection between I_i and I_m

Case 3: There exists an interval I_f in T with a fixed left point and a free right point
This case is symmetric to Case 2. Here we pull the right points of free intervals in T towards the right to satisfy their intersection with I_f, and then run the *fix_endpoints* subroutine.

When no free intervals remain in T, we check if the intervals obtained satisfy the edge weights in G. If all the weights are satisfied then we have an interval representation of G, otherwise we conclude that G is not an edge-weighted interval graph (correctness will be proved in section 4.2.2).

4.2.1 Pseudocode

In this section we give the pseudocode of the subroutine *fix_endpoints* followed by the psuedocode of the main algorithm.

For the *fix_endpoints* subroutine, we find the leftmost left point and the rightmost right point of T in $O(n)$ time, and then we fix an interval's left point if is it not equal to the leftmost point, and its right point if it is not equal to the rightmost point.

Algorithm 4.1 Subroutine for fixing right and left endpoints in the tentative interval representation T.

```
1: procedure FIX_ENDPOINTS($T$, $V(G)$)          ▷ $T$ is the tentative representation of $G$
2:     $rightmost \leftarrow 1$
3:     $leftmost \leftarrow 1$
4:     for each $v \in V(G)$ do ▷ Find the rightmost right point and the leftmost left point
5:         if $l(I_v) < leftmost$ then
6:             $leftmost \leftarrow l(I_v)$
7:         end if
8:         if $r(I_v) > rightmost$ then
9:             $leftmost \leftarrow r(I_v)$
10:        end if
11:    end for
12:    for each $v \in V(G)$ do                                   ▷ Fix the endpoints
13:        if $leftmost < l(I_v)$ then
14:            Fix $l(I_v)$
15:        end if
16:        if $rightmost > r(I_v)$ then
17:            Fix $r(I_v)$
18:        end if
19:    end for
20: end procedure
```

Algorithm 4.2 Checking if a given complete graph is an edge-weighted interval graph

```
1: Sort the edges of $G$ in increasing order of weight
2: Initialize $T$ to contain $n$ intervals that have free endpoints set to 1
3: **while** $T$ has free intervals **do**
4:     **if** $T$ contains only one free interval **then**
5:         Fix the endpoints of the free interval
6:     **end if**
7:     **if** all free intervals have both endpoints free **then**                ▷ Case 1
8:         Pick the smallest weight edge $(i, j)$ where both $i, j$ are free
9:         **for each** free interval $I_k \in T$ **do**
10:            $r(I_k) \leftarrow l(I_k) + s(i, j)$                ▷ Pull right points of free intervals
11:        **end for**
12:        **if** at least one of $i, j$ is a uniform vertex for free intervals **then**
13:            Fix the left and right points of the interval of the uniform vertex.
14:        **else**
15:            Fix the left point of $I_j$ and the right point of $I_i$.
16:        **end if**
17:    **else if** $\exists I_f \in T$ with a fixed right point and a free left point **then**    ▷ Case 2
18:        **for each** $k \in V(G), k \neq f$ and $l(I_k)$ is free **do**
19:            $l(I_k) \leftarrow min\{min\{r(I_f), r(I_k)\} - s(f, k), l(I_k)\}$
20:            $l(I_f) \leftarrow min\{l(I_f), l(I_k)\}$        ▷ Pull $l(I_f)$ and $l(I_k)$ as far left as required
21:        **end for**
22:        Fix the left point of $I_f$.
23:        FIX_ENDPOINTS($T$,$V(G)$)
24:    **else if** $\exists I_f \in T$ with a fixed left point and a free right point **then**    ▷ Case 3
25:        **for each** $k \in V(G), k \neq f$ and $r(I_k)$ is free **do**
26:            $r(I_k) \leftarrow max\{max\{l(I_f), l(I_k)\} + s(f, k), r(I_k)\}$
27:            $r(I_f) \leftarrow max\{r(I_f), r(I_k)\}$     ▷ Pull $r(I_f)$ and $r(I_k)$ as far right as required
28:        **end for**
29:        Fix the right point of $I_f$.
30:        FIX_ENDPOINTS($T$,$V(G)$)
31:    **end if**
32: **end while**
33: **if** $T$ is the interval representation of $G$ **then**
34:     $G$ is an edge-weighted interval graph with $T$ as a representation
35: **else**
36:     $G$ is not an edge-weighted interval graph
37: **end if**
```

4.2.2 Correctness Proof

In this section we prove the correctness of Algorithm 4.2.

Theorem 4.1. *Algorithm 4.2 correctly decides if an edge-weighted complete graph has an edge-weighted interval representation, and if it does, produces one such representation.*

Proof. **Preconditions:**

1. Graph G is a weighted complete graph.
2. The edges in G have positive weights.
3. The endpoints of all n intervals of T are free and set to 1.

Postcondition: Obtain a minimal edge-weighted interval representation M of G if there exists one.

Loop Invariants:

1. T is a tentative representation of the rightmost representation M.
2. (a) All free left points $\leq$ fixed left points
   (b) All free right points $\geq$ fixed right points
3. All free left points are equal. All free right points are equal.

We begin by proving that the invariants are true before entering the while loop. Invariant 1 is true from the definition of rightmost representation which says that all intervals in M must contain the point 1. Since all endpoints in T are free and equal, Invariants 2 and 3 are true before entering the loop.

The correctness of the algorithm during the cases that may arise during each iteration of the while loop is as follows:

Case 1: Both endpoints of each free interval in T are free
The algorithm increases the length of all free intervals in T to $s(i, j)$ which is the smallest edge weight in the free subgraph F. As F is a clique, we know from Theorem 2.1 that M needs to contain a common subinterval of length $s(i, j)$ in the intervals of F. Due to Invariant 1, we know that the left points of the intervals in T are as far right as they

can possibly be. Since in Case 1 we do not change the left points of the free intervals, T continues to be the rightmost tentative representation. As T is the rightmost interval representation which has the common intersection of length $s(i,j)$ between its free intervals, therefore T has to be contained in M.

Before entering Case 1, we know from the loop invariants that the left points of all free intervals are equal and the right points of all free intervals are equal. Therefore, to increase the length of free intervals to $s(i,j)$, the algorithm pulls forward the right points of all free intervals by an equal amount, hence ensuring that all right points in their new place are equal. The pulling of the free right points leaves behind the fixed right points, hence all fixed right points are before (less than) the free right points. After increasing the length of free intervals, the algorithm fixes one left and one right point (which may belong to the same or different intervals). The newly fixed right point is therefore equal to the free right points, and since no left point was moved in Case 1, the newly fixed left point is equal to the free left points, which are equal to each other. Therefore Case 1 does not violate invariants 2 and 3.

Case 2: $\exists f$ such that I_f has a fixed right point and a free left point
Let $I_v, v \neq f$ be an interval with a free left point in T. The right point of v can be either free or fixed. If v has a fixed right point, then we have no choice but to pull the left points of I_f and I_v to satisfy $s(f,v)$, since we can't change the right points of either I_f or I_v. If v has a free right point, then we know that $r(I_f) \leq r(I_v)$, and hence again we have to pull the left points to satisfy $s(f,v)$. Therefore we observe that the only way to satisfy the intersection between v and f is by pulling their left points. Any interval representation M where I_f stops at $r(I_f)$ must have the left points of intervals I_v and I_f at $min\{r(I_f), r(I_v)\} - s(f,v)$ to satisfy the given intersection. Hence, the left points are as far right as they can be while still satisfying their intersection with I_f, and T is the rightmost tentative representation. Therefore, the intervals obtained in T after Case 2 must be contained in the corresponding intervals of M.

Observe that $l(I_f)$ is pulled as far left as required to satisfy its specified intersections with free intervals, hence $l(I_f)$ is equal to the left point of any free interval in T. Fixing $l(I_f)$ after Case 2, therefore, does not violate Invariant 2. Additionally, since the subroutine *fix_endpoints* fixes all left points that lie after any other left point and fixes right points that lie before any other right point, the call to *fix_endpoints* subroutine ensures that Invariants 2 and 3 are satisfied.

Case 3: $\exists v \in V$ such that I_v has a fixed left point and a free right point

Symmetric to Case 2 above except for Invariant 1. We know from Invariant 1 that T is a rightmost representation, so the left end points of the intervals are as far right as possible. Since in this case we pull all free intervals towards the right, the left endpoints of the intervals stay the same, and Invariant 1 is satisfied.

In each case we fix at least one end point, therefore eventually we fix all the endpoints in T, and the loop terminates.

Note that in all cases, we only pull an endpoint to satisfy a specified intersection, hence T is a minimal representation.

After all the intervals are fixed and T cannot be changed anymore, we check if the intervals in T satisfy the edge-weights of G. If all intersections are satisfied then we have found an edge-weighted interval representation of G, otherwise there does not exist any edge-weighted interval representation of G because all through the algorithm we have maintained a tentative interval representation where we tried to satisfy as many intersections as possible. □

4.2.3 Runtime Analysis

Theorem 4.2. *Algorithm 4.2 runs in $O(m \cdot \log n)$ time.*

Proof. As we are dealing with complete graphs, $n^2 = O(m)$. Sorting the edges takes $O(m \cdot \log m) = O(m \cdot \log n)$ time. We will show that the rest of the algorithm runs in $O(n^2) = O(m)$ time.

Each case fixes at least one endpoint of the n intervals. We can keep track of the number of free intervals and the number of intervals with both endpoints free by updating these counts at a cost of $O(1)$ each time we fix an endpoint. Thus, we can distinguish Case 1 from Cases 2 and 3 in $O(1)$ time.

To find the smallest weight edge in the free subgraph F in Case 1, we traverse the sorted list of edges and select the first edge (i, j) where both i and j are free, and then save the pointer to that edge. The pointer can help us in quickly finding the smallest free edge next time. Since we traverse the list only once in the algorithm, total time taken by Case 1 for finding the smallest weight-edges in F is $O(m)$. Additionally, the time taken to increase the lengths of all free intervals to $s(i, j)$ is $O(n)$. Lastly, to check if either i (and j) is uniform in the free graph F, we check if the edges adjacent to i (or j) in F have equal edge-weights—taking $O(n)$ time. Therefore, Case 1 takes $O(n)$ time.

If Case 1 does not apply, we select any free interval I_f by iterating over the list of intervals in $O(n)$ time, and apply Case 2 or Case 3 as relevant. Next we analyze the work done inside the Cases 2 and 3; the work done by the two cases is same as they are symmetric.

Recall that in each of cases 2 and 3 we first satisfy the the intersection of all free intervals with I_f, and then we run the *fix_intervals* subroutine. Satisfying the intersection weights $s(f, v), \forall v \in V(G)$ takes $O(n)$ time as we modify the endpoint of every free interval I_v. Also, from the pseudocode we can see that the running time of *fix_endpoints* is $O(n)$. Therefore, the total time taken by Case 2 (and Case 3) is also $O(n)$.

Since each case fixes at least one endpoint, every interval needs to enter the cases at most twice in order to get both of its endpoints fixed. Therefore, time taken for all intervals is $O(n^2) = O(m)$.

□

Chapter 5

Obtaining the Edge-Weighted Interval Representation of General Graphs

In Chapter 3, we defined the protrusion relation R_s and related terms. In this chapter, we explain how the properties of PQ-trees (see Section 2.3 in Chapter 2) can be leveraged to improve the running time of computing the protrusion relation R_s for a given graph. The main idea of our algorithm is to identify a set of vertices that can be used to find most of the protrusion relationships, and find the remaining protrusions using Algorithm 4.2 (algorithm for complete graphs).

Note that since R_s is not symmetric, throughout this chapter we will test for both (u, v) and $(v, u) \in R_s$ for any edge $(u, v) \in V(G)$.

We start by examining the protrusion relationships that can be found using one vertex w.

Claim 5.1. If vertices u, v, w form a triangle in the graph and u does not protrude from w, then $(u, v) \in R_s$ if and only if $s(u, w) > s(u, v)$.

Proof. Consider any minimal s-respecting representation M of G. If I_u protrudes from I_v, then $s(u, v)$ must be less than the length of I_u. Since u does not protrude from w, $s(u, w)$ must be equal to the length of I_u. Therefore, if $(u, v) \in R_s$ and u does not protrude from w, then $s(u, w) > s(u, v)$.

For the other direction, notice that if $s(u, w) > s(u, v)$, then $(u, v) \in R_s$ by the definition of R_s. □

Furthermore, we can use w to detect some more protrusion relations as follows:

Claim 5.2. If vertices u, v, w form a triangle in the graph, and u protrudes from w, and v does not protrude from w, then $(u, v) \in R_s$.

Proof. Consider an s-respecting minimal representation M of G. Since v does not protrude from w, I_v shares all its points with I_w. On the other hand, since u protrudes from w, I_u contains a point p outside I_w. As u, v are connected in the graph, $I_u \cap I_v \neq \phi$, and by definition, u protrudes from v and $(u, v) \in R_s$. $\square$

Additionally, for any vertex w, we can find all protrusions of the form (u, w) and (w, u) in $O(m)$ time.

Claim 5.3. For any vertex w, we can find all u such that $(u, w) \in R_s$ and all u such that $(w, u) \in R_s$ in $O(m)$ time by testing all edges (u, v).

Proof. Assume that u protrudes from w. Since $(u, w) \in R_s$, by definition there must exist a vertex v that also protrudes from w and satisfies either $s(u, v) > s(u, w)$ or $s(u, v) > s(v, w)$. Therefore, by testing if $s(u, v) > s(u, w)$ for every edge (u, v), we can find all vertices u, v that protrude from w.

All protrusions of the form (w, u) can also be found from the definition of R_s as described next. Observe that w protrudes from u if there exists an edge (u, v) such that $s(v, w) > s(u, v)$ or $s(v, w) > s(u, w)$. Therefore, by testing the above condition for every edge (u, v), we can detect all $(w, u) \in R_s$ in $O(m)$ time. $\square$

To compute R_s, we first find the connected components of the given graph, and then we find the PQ-tree of each component. The following sections present algorithms for computing R_s when the root of the PQ-tree of a component G_s is a P-node or a Q-node. Note that if the root is a leaf, then the graph is a clique, so we use Algorithm 4.2 to find the edge-weighted interval representation of G_s.

5.1 Computing R_s when PQ-tree has a root P-node

From Claim 2.1 in Chapter 2, we know that there must be at least one vertex w in G_s that covers the root of the tree, and therefore, is adjacent to every other vertex in G_s. Using Claim 5.3, Claim 5.2, and Claim 5.1, we will find a set $D_w \subseteq R_s$ of protrusion relations determined by w in time $O(m)$.

We first describe how to compute D_w, and then characterize which protrusions of G_s are missing in D_w.

The set D_w is computed using the following tests :

Test 1 Use Claim 5.3 to find all v such that w protrudes from v, i.e. $(w, v) \in R_s$. Add all such pairs to D_w.

Test 2 Use Claim 5.3 to find the set P_w of vertices v that protrude from w, i.e. $P_w = \{v : (v, w) \in R_s\}$. Add all pairs $(v, w) : v \in P_w$ to D_w. Let $\overline{P_w} = \{v : v \notin P_w\}$. Therefore, $\overline{P_w}$ is the set of vertices that do not protrude from w.

Test 3 For every edge (u, v) with $u \in P_w$, $v \in \overline{P_w}$, add (u, v) to D_w, as justified by Claim 5.2 (recall that since w is adjacent to all vertices, u, v, w form a triangle).

Test 4 For every edge (u, v) with $u \in \overline{P_w}$, test if $s(u, w) > s(u, v)$ and if so, add (u, v) to D_w, as justified by Claim 5.1.

The pseudocode of computing the set D_w using the above tests is given in Algorithm 5.2.

We now characterize the protrusions $(u, v) \in R_s$ of G_s that are missing from D_w.

Lemma 5.1. *If $(u, v) \in R_s$ but not in D_w, then u and v protrude from w.*

Proof. Test 1 and 2 above add in D_w all protrusions of the form (u, w) and (w, u) as justified by Claim 5.3. Moreover, as justified by Claim 5.1, Test 4 adds to D_w all protrusions (u, v) where u does not protrude from w. Also, as justified by Claim 5.2, Test 3 adds to D_w all protrusions $(u, v) \in R_s$ where v does not protrude from w. Therefore, the only protrusions (u, v) not in D_w are where both u and v protrude from w. □

From Lemma 5.1, we see that the only protrusions missed by D_w are between vertices in P_w. We will now restrict our attention to $P_w^{'} = P_w \cup \{w\}$. To justify this, we will prove that any protrusion between $u, v \in P_w$ is witnessed by some element of $P_w^{'}$.

Claim 5.4. Any protrusion relation $(u, v) \in R_s$ between $u, v \in P_w$ is witnessed by some $x \in P_w^{'}$.

Proof. Consider any minimal s-respecting interval representation M of G. Suppose vertices $u, v \in P_w$ with $(u, v) \in R_s$. Then I_u protrudes from I_v in M, and hence there must exist at least one point $p \in I_u \setminus I_v$.

If there exists a point $p \in I_u \setminus I_v$ such that $p \in I_w$, then w witnesses the protrusion (see Figure 5.1a), and we know that $w \in P_w^{'}$.

Otherwise suppose that x witnesses the protrusion. Then I_x contains a point $p \in I_u \setminus I_v$, and since $p \notin I_w$, therefore I_x protrudes from I_w, so $x \in P_w^{'}$ (see Figure 5.1b).

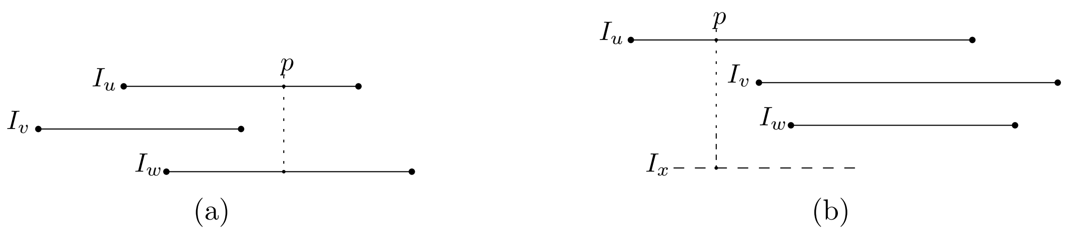


Figure 5.1: Either w witnesses $(u, v) \in R_s$ (Figure 5.1a), or $P_w^{'}$ contains a vertex x that witnesses $(u, v) \in R_s$ (Figure 5.1b)

□

Observation 5.1. To find all protrusions of the subgraph G' induced by a set U of vertices of G, it suffices to find the protrusion relation of each maximal clique of G'.

Proof. If $u, v \in U$ and u protrudes from v as witnessed by $x \in U$, then u, v, x form a triangle, and thus are contained in some maximal clique of U. □

Thus it suffices to find the protrusions in each maximal clique of G'—the subgraph induced on vertex set $P_w^{'}$. We can do this using Algorithm 4.2. Finally, we will show that G' has at most two maximal cliques, thus making our approach efficient.

Claim 5.5. Subgraph G' has at most two maximal cliques.

Proof. Consider a minimal s-respecting representation of G_s. It gives a linear ordering of the maximal cliques of G_s, corresponding to a frontier of the PQ-tree for G_s. Let A and B be the first and last maximal cliques in this ordering. Note that w is in all the cliques of G_s, so it corresponds to an interval spanning all the cliques.

Consider any element u of P_w. Then I_u must protrude from I_w, so u is contained in A or B (or both).

We claim that $A_w = A \cap P_w^{'}$ and $B_w = B \cap P_w^{'}$ are the only maximal cliques in G'. We prove this by contradiction. Let C be another maximal clique in G'. We must have some $y \in C \setminus A$ since C is not contained in A, and similarly, some $x \in C \setminus B, x, y \in P_w$.

Then $x \in A$, and $y \in B$ since every vertex of P_w is in A or B. We now consider the structure of the PQ-tree of G. Let $N_1, \cdots, N_t$ be the children of the root of the PQ-tree, ordered as in our representation.

Since $x \notin B, x$ does not cover the root, and hence from Corollary 2.1, it must be in only one child of the root, namely N_1. Since $x \in C$, therefore C is in N_1.

Similarly, since $y \notin A, y$ must be in only one child of the root, namely N_t. Since $y \in C$, therefore C is in N_t.

Any clique cannot be in two children of the root, and hence we arrive at a contradiction. Therefore, A_w and B_w are the only two cliques in G'.

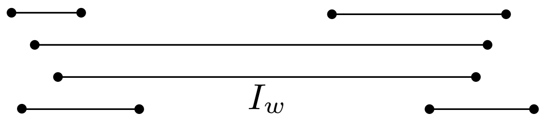


Figure 5.2: Intervals protruding from interval I_w lie in at most two maximal cliques

□

5.1.1 Summary of the Algorithm and Pseudocode

When the root of the PQ-tree of a component G_s is a P-node, we first find a vertex w that covers the root of the PQ-tree. Then, we find the set P_w of vertices that protrude from w (pseudocode given in Algorithm 5.1), followed by computing set D_w of protrusions relations by conducting Tests 1-4 (pseudocode given in Algorithm 5.2). After that, we find the maximal cliques of the subgraph induced by the vertices in $P_w \cup \{w\}$, and check if we have at most two maximal cliques, if not, then G is not an edge-weighted interval graph. If there are at most two maximal cliques, we use Algorithm 4.2 on each of those cliques to obtain their edge-weighted interval representation and then we obtain their protrusion relations. As a result, we obtain all protrusions of the component G_s.

We give below the psuedocode of the subroutine to find set P_w.

Algorithm 5.1 Determining the set P_x of vertices protruding from a given vertex x

```
procedure COMPUTE $P(x)$
    for each edge $(u, v) \in E(G_s)$ do
        if $s(u, v) > s(u, x)$ then
            Add $u$ and $v$ to $P_x$
        end if
    end for
return $P_x$
end procedure
```

Next, we give the pseudocode of the algorithm for finding all protrusions D_x found by using Tests 1-4 on a vertex x.

Algorithm 5.2 Computing set D_x of protrusions using a single vertex x

```
procedure COMPUTE $D(x, P_x)$
    for each edge $(u, v) \in E(G_s)$ do ▷ Test 1: Add to $D_x$ protrusions of the form $(x, u)$
        if $s(v, x) > s(u, v)$ or $s(v, x) > s(u, x)$ then
            Add $(x, u)$ to $D_x$
        end if
    end for
    for each vertex $v \in V(G_s)$ do    ▷ Test 2: Add all protrusions $(v, x) : v \in P_x$, to $D_x$
        if $v \in P_x$ then
            Add $(v, x)$ to $D_x$
        end if
    end for
    for each edge $(u, v) \in E(G_s)$ do    ▷ Test 3: Find protrusions using Claim 5.2
        if $u \in P_x$ and $v \in \overline{P_x}$ then
            Add $(u, v)$ to $D_x$
        end if
    end for
    for each edge $(u, v) \in E(G_s)$ do    ▷ Test 4: Find protrusions using Claim 5.1
        if $u \in \overline{P_x}$ and $s(u, x) > s(u, v)$ then
            Add $(u, v)$ to $D_x$
        end if
    end for
return $D_x$
end procedure
```

Finally, we give the pseudocode of finding all protrusions of G_s.

Algorithm 5.3 Determining R_s when the root of a PQ-tree is a P-node

```
1: for each edge $(u, v)$ in $E(G_s)$ do    ▷ As $R_s$ is not symmetric, we must also test if $(v, u) \in R_s$. Hence we convert the subgraph $G_s$ to a directed graph by adding edge $(v, u)$
2:     Add edge $(v, u)$ to $E(G_s)$
3: end for
4: Select a vertex $w$ that covers the root node of the $PQ$-tree
5: $P_w \leftarrow$ COMPUTE P$(w)$
6: $D_w \leftarrow$ COMPUTE D$(w,\ P_w)$
7: $P_w^{'} \leftarrow P_w \cup \{w\}$
8: if Subgraph induced by the vertices in $P_w^{'}$ has at most two maximal cliques $A$, $B$ then
9:     Obtain the $s$-respecting representation of $A$ and $B$ using Algorithm 4.2 to determine the protrusion relations of the edges in $P_w$
10: else
11:     $G$ is not an edge weighted interval graph
12: end if
```

5.1.2 Runtime Analysis and Proof of Correctness

We now provide a proof of correctness and the run time analysis for Algorithm 5.3.

Lemma 5.2. *If the root of the PQ-tree of a graph G_s is a P-node, then Algorithm 5.3 finds all protrusions of G_s.*

Proof. Lemma 5.1 states that Tests 1-4 add all protrusions (u, v) to D_w except those for which $u, v \in P_w$. The protrusions $(u, v) : u, v \in P_w$ are detected by examining the interval representations obtained by Algorithm 4.2 on the maximal cliques of the subgraph G' induced by vertices in $P_w^{'}$ as implied by Claim 5.4. Hence, all protrusions of G_s are detected using Algorithm 5.3. □

Lemma 5.3. *Algorithm 5.3 runs in time $O(m \cdot \log n)$.*

Proof. Observe that Tests $1, 3, 4$ and Algorithm 5.1 each spend constant time on every edge of G_s, and Test 2 takes $O(n)$ time, hence the runtime of Algorithm 5.2 is $O(m)$. We spend $O(m)$ time to find the maximal cliques A and B of the subgraph induced by vertices in $P_w^{'}$.

The running time of Algorithm 4.2 to compute the s-respecting interval representation of A and B is $O(m \cdot \log n)$, and we spend $O(m)$ time on the interval representations of A and B to obtain their protrusion relations. Therefore, the total running time of our algorithm is $O(m \cdot \log n)$. □

5.2 Computing R_s when PQ-tree has a root Q-node

When the root of the PQ-tree of G_s is a Q-node, we cannot use Algorithm 5.3 as not always is there a vertex w that covers the root Q-node, and even if such a vertex exists, Claim 5.5 does not hold for it. Therefore, in this section, we employ other properties of PQ-trees and Q-nodes to obtain the protrusion relation of G_s. Recall that the ordering of the children of a Q-node is unique up to reflection. We will use the ordering $O_q = N_1, \cdots, N_t$ of the children of the root Q-node of the PQ-tree while computing the protrusions of G_s.

Notice that for any $(u, v) \in E(G_s)$, if u is contained in a maximal clique that v is not contained in, then u must protrude from v in all minimal s-respecting representations of G_s—this simple observation can be used to find most of the protrusions in G_s. The sets of maximal cliques of G_s that any two vertices u and v are contained in can be compared in constant time if we precompute the *start-clique* and *end-clique* of each $v \in V(G_s)$ relative to some frontier F of the PQ-tree, defined next. The *start-clique* $S(v)$ of a vertex v is the first clique in F that v appears in. Likewise, the *end-clique* of v is the last clique in F that v appears in. If $S(u)$ appears before $S(v)$ or $E(u)$ comes after $E(v)$ in F, then u must protrude from v in all edge weighted interval representations of G. In other parts of the algorithm we use related terms *start-node* and *end-node* of a vertex v. The *start-node* $SN(v)$ is the node N_i of O_q where v appears for the first time in F. Likewise, the *end-node* $EN(v)$ of a vertex v is the node N_j of O_q where v appears for the last time in F. We describe how to compute $S(v)$ and $E(v)$, and $SN(v)$ and $EN(v)$ for all vertices $v \in V(G_s)$ in $O(m)$ time in Subsection 5.2.2.

If u and v start in the same clique or end in the same clique, then we may not be able to find all the protrusions between u and v by comparing their start and end-cliques. We find these protrusions using a subset W of long vertices of O_q satisfying the properties below. Recall that long vertices of a Q-node appear in multiple children of the node.

Property 1 For any $i = 1, \dots, t-1$ there exists a vertex $w \in W$ that covers N_i and N_{i+1}. We denote the vertex $w \in W$ that covers node N_i and N_{i+1} as w_i.

Property 2 For any $w, w' \in W$, if $(w, w') \in E(G_s)$, then $(w, w') \in R_s$.

Such a set W can be found in $O(m)$ time as shown in Subsection 5.2.2.

We leverage the above-mentioned properties of PQ-trees and set W in the following tests to compute a relation $D \subseteq R_s$:

Test 1 For each edge (u, v), if u is in a clique that v is not in, then add (u, v) to D. We can do this by checking if $S(u)$ is ordered before $S(v)$ or if $E(u)$ is ordered after $E(v)$ in F.

Test 2 As justified by Property 2 of set W, add (w, w') to D if $w, w' \in W$ and $(w, w') \in E(G_s)$.

Test 3 For each edge (u, v), if u and v both start in a clique C_i of $N_i, i \neq 1$, then test if w_{i-1} witnesses u protruding from v. If so, add (u, v) to D.

Test 4 Similarly for each edge (u, v), if both u and v end in a clique C_i of $N_i, i \neq t$, then test if w_i witnesses u protruding from v. If so, add (u, v) to D.

We then add more protrusions of G_s to set D by using Algorithm 5.1 to compute the sets P_{w_1} and $P_{w_{t-1}}$ of vertices that protrude from w_1 and w_{t-1} respectively, and then computing relations D_{w_1} and $D_{w_{t-1}}$ by using Tests 1-4 on page 32 to obtain all protrusions of the following form:

1.1 (w_1, v)

1.2 (v, w_1)

1.3 $(u, v), u \in P_{w_1}, v \in \overline{P_{w_1}}$

1.4 $(u, v), u \in \overline{P_{w_1}}$

2.1 (w_{t-1}, v)

2.2 (v, w_{t-1})

2.3 $(u, v), u \in P_{w_{t-1}}, v \in \overline{P_{w_{t-1}}}$

2.4 $(u, v), u \in \overline{P_{w_{t-1}}}$

We then add the protrusions from D_{w_1} and $D_{w_{t-1}}$ to D. Let us define the set $P'_{w_1} = \{v : SN(v) = N_1;$ and $v \in P_{w_1}\} \cup \{w_1\}$ and set $P'_{w_{t-1}} = \{v : EN(v) = N_t;$ and $v \in P_{w_{t-1}}\} \cup \{w_{t-1}\}$. We claim that the only protrusions (u, v) of G_s that are not in D are where $u, v \in P'_{w_1}$ or $u, v \in P'_{w_{t-1}}$.

Claim 5.6. If $(u, v) \in R_s$ and $(u, v) \notin D$, then u, v must satisfy one of the following conditions:

- u and v protrude from w_1 and start in N_1
- u and v protrude from w_{t-1} and end in N_t

Proof. Consider a minimal s-respecting representation M of G_s. Flip the representation if necessary so that the N_i's appear in the order $N_1, \cdots, N_t$ from left to right.

Consider $(u, v) \in R_s$. Then I_u protrudes from I_v in the representation, say on the left (the other case is symmetric).

In the case when I_u starts in a clique before the clique where I_v starts in M, we have put (u, v) in D by Test 2 above.

In the case when both u and v start in the same clique C_i of node $N_i, i \neq 1$, we claim that the protrusion is witnessed by w_{i-1}, and hence is added to D by Test 3. We know from the definition of W that w_{i-1} starts in a node that is ordered before N_i in M, and since both u and v start in N_i, w_i protrudes from v on the left, and by Corollary 3.3, witnesses the protrusion of u from v. Therefore (u, v) is added to D by Test 3.

If u and v start in N_1, and at least one of u or $v \in \overline{P_{w_1}}$, then the protrusion is added to set D as justified by Claim 5.2 and Claim 5.1.

Therefore, the only case where I_u protrudes from I_v on the left in M and (u, v) is not added to D is where both u and v start in N_1 and protrude from w_1.

By symmetry, the only case where I_u protrudes from I_v on the right in M and (u, v) is not added to D is where both u and v end in N_t and protrude from w_{t-1}. □

This claim implies that all protrusions $(u, v) \in R_s$ that are not in D are between vertices in set P'_{w_1} or between vertices in set $P'_{w_{t-1}}$. Let us define G_1 to be the subgraph of G induced by set P'_{w_1}, and G_2 to be the subgraph of G induced by set $P'_{w_{t-1}}$. We claim that both G_1 and G_2 are cliques, and hence we can obtain the protrusions of G_1 and G_2 by using Algorithm 4.2.

Claim 5.7. Subgraph G_1 and G_2 are cliques.

Proof. Consider a minimal s-respecting interval representation M of graph G_s. Flip the representation if necessary so that N_i's appear in the order $N_1, \cdots, N_t$ from left to right.

The representation gives an ordering of the cliques. Let A be the first clique and B be the last clique in M. Then A is a clique of N_1 and B is a clique of N_t.

We will prove that P'_{w_1} is a subset of A and $P'_{w_{t-1}}$ is a subset of B.

Observe that all intervals that protrude from I_{w_1} on the left in M lie in A. Also, as w_1 is a long vertex it covers N_1, so it is in clique A.

Now let us look at the intervals that protrude from I_{w_1} on the right in M and start in N_1; consider one such interval I_v. Note that the end-clique of w_1 lies outside N_1. As I_v protrudes from I_{w_1} on the right, I_v must appear in N_2, and hence it is a long interval. As I_v is a long interval, it covers node N_1, and hence it must lie in clique A. Thus, G_1 is the subset of clique A and for this reason is a complete graph.

The proof is symmetric for G_2. □

Therefore, to find the protrusion relations in P'_{w_1} and $P'_{w_{t-1}}$, we simply examine the minimal s-respecting interval representation of G_1 and G_2. We now prove that all the protrusions relations in P_{w_1} and $P_{w_{t-1}}$ that are not in D are detected by examining the s-respecting interval representation of G_1 and G_2.

Claim 5.8. The protrusions (u, v) of G_s not in D are witnessed by some element of P'_{w_1} or $P'_{w_{t-1}}$.

Proof. Consider a minimal s-respecting interval representation M of graph G_s. Flip the representation if necessary so that N_i's appear in the order $N_1, \cdots, N_t$ from left to right. Consider a protrusion (u, v) where $u, v \in P_{w_1}$, assume that u protrudes from v on the left in M. We will prove this protrusion of u from v is either already added to D by tests on page 38, or is witnessed by an element of P'_{w_1}.

In the case when u starts in a clique to the left of the clique that v starts in, the protrusion is already added to D by Test 2.

Since u protrudes from v on the left in M, we know from Lemma 3.1 that there must exist a vertex $x \in V(G_s)$ that witnesses this protrusion and from Corollary 3.3 that I_x also protrudes from I_v on the left in M. We know from Corollary 3.2 that u protrudes from v in all s-respecting interval representations of any subgraph of G containing vertices u, v, x. Hence, by examining the minimal s-respecting interval representations of G_1 obtained by Algorithm 4.2, we can detect the protrusion of u from v as long as $x \in P'_{w_1}$. Therefore we will now prove that there exists a vertex in P'_{w_1} whose interval protrudes from I_v on the left in M.

The case when I_{w_1} protrudes from I_v on the left in M is trivial as $w_1 \in P'_{w_1}$.

Consider the case where I_{w_1} does not protrude from I_v on the left in M, i.e. $l(I_v) \leq l(I_w)$. Since u protrudes from v on the left in M, we know from Lemma 3.1 that there must exist a vertex $x \in V(G_s)$ that witnesses this protrusion. Additionally, from Corollary 3.3 we know that I_x protrudes from v on the left in M i.e. $l(I_x) < l(I_v)$. Note that I_x must start in N_1 as it intersects I_u, which starts in N_1. Also, since $l(I_x) < l(I_v) \leq l(I_w)$, therefore I_x protrudes from I_{w_1} on the left in M, and thus $x \in P'_{w_1}$.

Now consider when $u, v \in P'_{w_1}$ and u protrudes from v on the right in M. We prove that this protrusion is either already added to D or is witnessed by some element in $P_{w'_{t-1}}$.

Since u protrudes from v on the right, we will look at the end-nodes of u and v. If u ends in a clique that appears to the right of the end-clique of v in M, then the protrusion is added to D by Test 2.

If u, v both end in a node $N_i, i \neq t$, then the protrusion is added to D by Test 4.

If u, v end in N_t, and u does not protrude from w_{t-1}, then the protrusion is added to D from $D_{w_{t-1}}$. Additionally, if u protrudes from w_{t-1} but v does not, then again the protrusion is added to D by $D_{w_{t-1}}$. Lastly, if both u, v protrude from w_{t-1}, then the protrusion will be witnessed by some element in $P'_{w_{t-1}}$ as the proof is symmetric for $u, v \in P'_{w_{t-1}}$. □

We now claim that all protrusions of G_s have been detected by our algorithm.

Lemma 5.4. *Algorithm 5.5 detects all the protrusions of G_s when the root of the PQ-tree of G_s is a Q-node.*

Proof. The proof follows from Claim 5.6 and Claim 5.8. □

In following subsections we give the algorithms for finding the start and end-cliques and start and end-nodes of all vertices, and for computing set W.

5.2.1 Finding Start-Cliques and End-Cliques, and Start-Nodes and End-Nodes of Vertices

First, we obtain the frontier of a PQ-tree of G_s. Then, we visit the maximal cliques of G_s in the order specified by F. For every clique C that we visit, we check if each vertex v contained in C has been assigned a start-clique, and if not, then we assign C as the start-clique of v. Therefore, assigning the start-clique of each vertex of G_s takes $O(m)$

time as each vertex v appears in $deg(v)$ nodes from O_q. The end-clique of each vertex can be found by flipping F and using a similar process. Hence the total time taken for finding $S(v)$ and $E(v)$ $\forall v \in V(G_s)$ is $O(m)$.

Note that the PQ-tree of a graph has linear size. Therefore, we can traverse the PQ-tree once to find for each clique which child of the root it is descended from, and hence find the start-node $SN(v)$ and the end-node $EN(v)$ for all vertices of G_s in linear time.

5.2.2 Algorithm for Computing Set W

As mentioned earlier, we wish to find a set W of vertices with the following properties:

Property 1 For any $i = 1, \ldots, t-1$ there exists a vertex $w \in W$ that covers N_i and N_{i+1}. We denote the vertex $w \in W$ that covers node N_i and N_{i+1} as w_i.

Property 2 For any $w, w' \in W$, if $(w, w') \in E(G_s)$, then $(w, w') \in R_s$

There are various ways to find a set W, we give one here (perhaps not the most efficient). We first find a set W' of vertices satisfying Property 1 above, and then modify W' such that it satisfies Property 2 as well. Note that since G_s is a connected component, the set of all vertices $V(G_s)$ already satisfies Property 1. Therefore, we initialize W' to the vertex set of G_s, i.e. $W^{'} = V(G_s)$.

We will now modify set W' such that it satisfies Property 2 by leveraging the fact that a vertex u protrudes from a vertex v if u is contained in a clique C and C does not contain v. Hence we remove every vertex v from W' if there exists a vertex $u \in N(v)$ such that $u \in W'$ and the set of maximal cliques containing u is a superset of the set of maximal cliques containing v. By doing so we ensure that for any two vertices $u, v \in W'$, u is in a maximal clique that does not contain v, and vice-versa. As a result, if vertices $u, v \in W'$ and $(u, v) \in E(G)$, then u protrudes from v, and since W' now satisfies Property 2 from above, we set W to W'. The time taken for computing set W is $O(m)$ as for each vertex $v \in W$, we compare the set of maximal cliques containing each $u \in N(v)$ with the set of maximal cliques containing v in constant time by comparing the start and end-cliques of v and u.

After obtaining the set W, we associate each node N_i with a vertex $w_i \in W$ such that for each $N_i \in O_q, i \neq t, w_i$ covers N_i and N_{i+1}. We do so by visiting each node $N_i \neq EN(v)$ that a vertex $v \in W$ appears in, and setting w_i to v. Observe that some w_i's will be reassigned multiple times; however, this does not affect the correctness or running time of the algorithm. Also note as each $v \in W$ lies in $O(deg(v))$ nodes in O_q, hence the running time of assigning w_i to every $N_i \neq N_t \in O_q$ is $O(m)$.

Algorithm 5.4 Procedure for computing set W

```
procedure COMPUTE W
    $W' \leftarrow V(G_s)$
    for each $v \in W'$ do
        for each $u \in N(v)$ do
            if $u \in W'$ and $S(u) \leq S(v)$ and $E(u) \geq E(v)$ in $F$ then
                Remove $v$ from $W'$
            end if
        end for
    end for
    $W \leftarrow W'$
    for each vertex $v \in W$ do                    ▷ Assign a $w_i$ to each $N_i \in O_q$
        for each $N_i \neq EN(v)$ that $v$ appears in do
            $w_i \leftarrow v$
        end for
    end for
return $W$
end procedure
```

5.2.3 Summary of the Algorithm and Pseudocode

When the root of the PQ-tree of a connected component G_s is a Q-node, we first obtain the ordering O_q of the immediate children of the root. The ordering is then used to find the start and end-clique, and the start and end-node of each vertex as described in Subsection 5.2.1. Next, we compute the set W as described in Subsection 5.2.2. After that we use Tests 1-4 to compute set D. Then we compute sets D_{w_1} and $D_{w_{t-1}}$ and add the protrusions in these sets to D. Finally, we find the maximal cliques G_1 and G_2 of the subgraphs of G_s induced by vertex sets P'_{w_1} and $P'_{w_{t-1}}$, and run Algorithm 4.2 on G_1 and G_2 to find all the protrusions of G_s.

The psuedocode of the algorithm is as follows:

Algorithm 5.5 Determining R_s when the root of a PQ-tree is a Q-node

```
1: Obtain the ordering $O_q$ and the frontier $F$ from the $PQ$-tree of $G_s$.
2: for each edge $(u, v)$ in $E(G_s)$ do        ▷ As $R_s$ is not symmetric, we must also test if $(v, u) \in R_s$. Hence we convert the subgraph $G_s$ to a directed graph by adding edge $(v, u)$
3:     Add edge $(v, u)$ to $E(G_s)$
4: end for
5: for each vertex $v \in V(G_s)$ do
6:     Determine the start-clique $S(v)$ and the end-clique $E(v)$ and start-node $SN(v)$ and end-node $EN(v)$of $v$
7: end for
8: $W \leftarrow$ COMPUTE W
9: for each edge $(u, v) \in E(G_s)$ do        ▷ Test 1
10:     if both $u, v \in W$ then
11:         Add $(u, v)$ to $D$
12:     end if
13: end for
14: for each edge $(u, v) \in E(G_s)$ do        ▷ Test 2
15:     if $S(u)$ before $S(v)$ or $S(u)$ after $S(v)$ in $F$ then
16:         Add $(u, v)$ to $D$
17:     end if
18: end for
19: for each edge $(u, v) \in E(G_s)$ do        ▷ Test 3
20:     if $S(u) = S(v)$ and $SN(u) \neq N_1$ then
21:         if $s(u, w_{i-1}) > s(u, v)$ or $s(u, w_{i-1}) > s(v, w_{i-1})$ then
22:             Add $(u, v)$ to $R_s$
23:         end if
24:     end if
25: end for
26: for each edge $(u, v) \in E(G_s)$ do        ▷ Test 4
27:     if $E(u) = E(v)$ and $EN(u) \neq N_t$ then
28:         if $s(u, w_i) > s(u, v)$ or $s(u, w_i) > s(v, w_i)$ then
29:             Add $(u, v)$ to $R_s$
30:         end if
31:     end if
32: end for
```

```
33: $P_{w_1} \leftarrow$ COMPUTE P($w_1$)    ▷ Compute the set of vertices protruding from $w_1$
34: $D_{w_1} \leftarrow$ COMPUTE D($w_1$, $P_{w_1}$) ▷ Compute the set of protrusions obtained by using $w_1$
35: $P_{w_{t-1}} \leftarrow$ COMPUTE P($w_{t-1}$)    ▷ Compute the set of vertices protruding from $w_{t-1}$
36: $D_{w_{t-1}} \leftarrow$ COMPUTE D($w_{t-1}$, $P_{w_{t-1}}$)    ▷ Compute the set of protrusions obtained by
    using $w_{t-1}$
37: $D \leftarrow D_{w_1} \cup D_{w_{t-1}}$
38: **for each** vertex $v \in P_{w_1}$ **do**    ▷ Add the vertices in $P_{w_1}$ that start in $N_1$ to set $P^{'}_{w_1}$
39:     **if** $SN(v) = N_1$ **then**
40:         Add $v$ to $P^{'}_{w_1}$
41:     **end if**
42: **end for**
43: Add $w_1$ to $P^{'}_{w_1}$
44: **for each** vertex $v \in P_{w_{t-1}}$ **do** ▷ Add the vertices in $P_{w_{t-1}}$ that end in $N_t$ to set $P^{'}_{w_{t-1}}$
45:     **if** $EN(v) = N_t$ **then**
46:         Add $v$ to $P^{'}_{w_{t-1}}$
47:     **end if**
48: **end for**
49: Add $w_{t-1}$ to $P^{'}_{w_{t-1}}$
50: **if** all vertices of $P^{'}_{w_1}$ form a clique $A$ and all vertices of $P^{'}_{w_{t-1}}$ form a clique $B$ **then**
51:     Using Algorithm 4.2, obtain the $s$-respecting interval representations $M_1$ and $M_2$
    of $A$ and $B$, respectively
52:     Examine $M_1$ and $M_2$ and add their protrusions to $D$
53: **else**
54:     $G_s$ (and hence $G$) is not an edge-weighted interval graph
55: **end if**
```

Lemma 5.5. *Algorithm 5.5 runs in $O(m \cdot \log n)$ time.*

Proof. Each of the tests conducted in Algorithm 5.5 does constant work on each edge. Hence the runtime of conducting the tests is $O(m)$. Since we run Algorithm 5.3 twice (to compute D_{w_1} and $D_{w_{t-1}}$), and Algorithm 4.2 twice (on G_1 and G_2), the total running time of the algorithm is $O(m \cdot \log n)$. $\square$

Chapter 6

Conclusion and Future Work

We studied a variation of interval graphs called edge-weighted interval graphs. First, we presented Algorithm 4.2 in Chapter 4 to recognize complete edge-weighted interval graphs in $O(m \cdot \log n)$ time. In Chapter 5, we improved the run time of Köbler's algorithm [28] for general graphs by computing the protrusion relation R_s in $O(m \cdot \log n)$ time by using PQ-trees. Therefore, we arrive at an $O(m \cdot \log n)$ time algorithm for recognizing edge-weighted interval graphs.

Note that the $\log n$ factor in our running time is due to sorting the edges by weight in Algorithm 4.2; hence, our algorithm can be improved if we can avoid sorting of edges. We are also interested if the techniques from our algorithm can be applied to obtain approximation algorithms for recognizing other types of weighted intersection graphs such as weighted boxicity graphs.

The following problems are interesting for future work:

- Can we find a linear time algorithm for recognizing edge-weighted interval graphs?
- Using techniques from our algorithm, can we find algorithms or approximation algorithms for recognizing other weighted intersection graphs (such as weighted boxicity graphs)?

References

[1] Alam, M., Biedl, T., Felsner, S., Kaufmann, M., and Kobourov, S. G. Proportional contact representations of planar graphs. *Journal of Graph Algorithms and Applications 16*, 3 (2012), 701–728.

[2] Alam, M. J., Biedl, T., Felsner, S., Gerasch, A., Kaufmann, M., and Kobourov, S. G. Linear-time algorithms for hole-free rectilinear proportional contact graph representations. *Algorithmica 67*, 1 (2013), 3–22.

[3] Alam, M. J., Kobourov, S. G., Liotta, G., Pupyrev, S., and Veeramoni, S. 3D proportional contact representations of graphs. In *Information, Intelligence, Systems and Applications, IISA 2014, The 5th International Conference on* (2014), IEEE, pp. 27–32.

[4] Bárány, I., Katchalski, M., and Pach, J. Quantitative Helly-type theorems. *American Mathematical Society 86*, 1 (1982).

[5] Benzer, S. On the topology of the genetic fine structure. *Proceedings of the National Academy of Sciences 45*, 11 (1959), 1607–1620.

[6] Booth, K. S. PQ-tree algorithms. Tech. rep., California Univ., Livermore (USA). Lawrence Livermore Lab., 1975.

[7] Booth, K. S., and Lueker, G. S. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ–tree algorithms. *Journal of Computer and System Sciences 13*, 3 (1976), 335–379.

[8] Carlisle, M. C., and Lloyd, E. L. On the k-coloring of intervals. *Discrete Applied Mathematics 59*, 3 (1995), 225–235.

[9] Corneil, D. G. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. *Discrete Applied Mathematics 138*, 3 (2004), 371 – 379.

[10] de Fraysseix, H., and de Mendez, P. O. Representations by contact and intersection of segments. *Algorithmica 47*, 4 (2007), 453–463.

[11] Deng, X., Hell, P., and Huang, J. Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM Journal on Computing 25*, 2 (1996), 390–403.

[12] Ehrlich, G., Even, S., and Tarjan, R. E. Intersection graphs of curves in the plane. *Journal of Combinatorial Theory, Series B 21*, 1 (1976), 8–20.

[13] Eschen, E. M., and Sinrad, J. P. An $O(n^2)$ algorithm for circular-arc graph recognition. In *Proceedings of the Fourth annual ACM-SIAM Symposium on Discrete Algorithms* (1993), Society for Industrial and Applied Mathematics, pp. 128–137.

[14] Fulkerson, D., and Gross, O. Incidence matrices and interval graphs. *Pacific Journal of Mathematics 15*, 3 (1965), 835–855.

[15] Gavril, F. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing 1*, 2 (1972), 180–187.

[16] Gilmore, P., and Hoffman, A. A characterization of comparability graphs and of interval graphs. *Selected papers of Alan Hoffman with commentary 16* (2003), 65.

[17] Golumbic, M., Kaplan, H., and Shamir, R. On the complexity of DNA physical mapping. *Advances in Applied Mathematics 15*, 3 (1994), 251 – 261.

[18] Golumbic, M. C. *Algorithmic graph theory and perfect graphs*, second ed. Elsevier, 2004.

[19] Habib, M., McConnell, R., Paul, C., and Viennot, L. LexBFS- and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science 234*, 1 (2000), 59–84.

[20] Hajós, G. Über eine art von graphen. *Internationale mathematische nachrichten 11* (1957), 65.

[21] Hliněný, P. Contact graphs of line segments are NP-complete. *Discrete Mathematics 235*, 1-3 (2001), 95–106.

[22] Hsu, W.-L. A simple test for interval graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science* (1992), Springer, pp. 11–16.

[23] Hsu, W. L. O(mn) algorithms for the recognition and isomorphism problems on circular-arc graphs. *SIAM Journal on Computing 24*, 3 (1995), 411–439.

[24] Jampani, K. R., and Lubiw, A. Simultaneous interval graphs. In *International Symposium on Algorithms and Computation* (2010), vol. 6506 of *Lecture Notes in Computer Science*, Springer, pp. 206–217.

[25] Jungck, J. R., Dick, G., and Dick, A. G. Computer-assisted sequencing, interval graphs, and molecular evolution. *Biosystems 15*, 3 (1982), 259–273.

[26] Kaplan, H., and Nussbaum, Y. A simpler linear-time recognition of circular-arc graphs. *Algorithmica 61*, 3 (2011), 694–737.

[27] Kendall, D. Incidence matrices, interval graphs and seriation in archeology. *Pacific Journal of Mathematics 28*, 3 (1969), 565–570.

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

[29] Korte, N., and Möhring, R. H. An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal on Computing 18*, 1 (1989), 68–81.

[30] Kratochvíl, J. String graphs. II. recognizing string graphs is NP-hard. *Journal of Combinatorial Theory, Series B 52*, 1 (1991), 67–78.

[31] Kratochvíl, J. A special planar satisfiability problem and a consequence of its NP–completeness. *Discrete Applied Mathematics 52*, 3 (1994), 233–252.

[32] Kratochvíl, J., and Matoušek, J. Intersection graphs of segments. *Journal of Combinatorial Theory, Series B 62*, 2 (1994), 289–315.

[33] Lekkerkerker, C., and Boland, J. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae 51*, 1 (1962), 45–64.

[34] McConnell, R. M. Linear-time recognition of circular-arc graphs. *Algorithmica 37*, 2 (2003), 93–147.

[35] McDiarmid, C., and Müller, T. Integer realizations of disk and segment graphs. *Journal of Combinatorial Theory, Series B 103*, 1 (2013), 114–143.

[36] McKee, T. A., and McMorris, F. R. *Topics in intersection graph theory.* SIAM, 1999.

[37] Pe'er, I., and Shamir, R. *Interval graphs with side (and size) constraints*, vol. 979 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 1995, pp. 142–154.

[38] Roberts, F. S. *Food webs, competition graphs, and the boxicity of ecological phase space*, vol. 642 of *Lecture Notes in Mathematics.* Springer, Berlin, Heidelberg, 1978, pp. 477–490.

[39] Rose, D. J., Tarjan, R. E., and Lueker, G. S. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing 5*, 2 (1976), 266–283.

[40] Schaefer, M., Sedgwick, E., and Štefankovič, D. Recognizing string graphs in NP. *Journal of Computer and System Sciences 67*, 2 (2003), 365–380.

[41] Schaefer, M., and Štefankovič, D. Decidability of string graphs. *Journal of Computer and System Sciences 68*, 2 (2004), 319 – 334.

[42] Simon, K. *A new simple linear algorithm to recognize interval graphs*, vol. 553 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 1991, pp. 289–308.

[43] Skrien, D. Chronological orderings of interval graphs. *Discrete Applied Mathematics 8*, 1 (1984), 69–83.

[44] Tucker, A. Matrix characterizations of circular-arc graphs. *Pacific Journal of Mathematics 39*, 2 (1971), 535–545.

[45] Yamamoto, N. Weighted interval graphs and their representations. *Masters Thesis. Tokyo Inst. of Technology* (2007).

[46] Yannakakis, M. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic Discrete Methods 3*, 3 (1982), 351–358.