Charles University in Prague
Faculty of Mathematics and Physics

Master Thesis

Pavel Klavík

Extending Partial Representations of Graphs

Department of Applied Mathematics

Supervisor

Prof. RNDr. Jan Kratochvíl, CSc.

Study program, specialization

Discrete Models and Algorithms
Discrete Mathematics and Combinatorial Optimization

2012

Acknowledgements

The results presented in this thesis are based on:

[29] Pavel Klavík, Jan Kratochvíl, and Tomáš Vyskočil.
Extending Partial Representations of Interval Graphs.
In *Theory and Applications of Models of Computation (TAMC) - 8th Annual Conference*, pages 276–285, 2011.

[27] Pavel Klavík, Jan Kratochvíl, Yota Otachi, and Toshiki Saitoh.
Extending Partial Representations of Subclasses of Chordal Graphs.
Submitted, pre-print: `http://arxiv.org/abs/1207.0255`

[26] Pavel Klavík, Jan Kratochvíl, Yota Otachi, Ignaz Rutter, Toshiki Saitoh, Maria Saumell, and Tomáš Vyskočil.
Extending Partial Representations of Proper and Unit Interval Graphs.
Submitted, pre-print: `http://arxiv.org/abs/1207.6960`

[28] Pavel Klavík, Jan Kratochvíl, Yota Otachi, Toshiki Saitoh, and Tomáš Vyskočil.
Extending Partial Representations of Interval Graphs.
Journal version of the TAMC paper, in preparation.

I hereby declare that I have written all text of this thesis on my own. I would like to thank coauthors of the above papers for fruitful discussions and many suggestions concerning problems and writing style. In order to indicate which results were obtained by myself, I shall describe chronologically how the results were discovered.

First of all, I would like to thank Prof. Jan Kratochvíl for supervising both my bachelor's thesis and my master's thesis. The origin of the partial representation extension problem is in my bachelor's thesis. We started working on interval graphs and later considered the problem for other graph classes. I would like to thank Pavol Hell for suggesting a PQ-tree approach. This thesis shows progress on the problems in the last two years.

Based on my bachelor's thesis, we wrote a paper (with many modifications based on many suggestions of the coauthors) which was accepted for the TAMC 2011 conference in Tokyo. After my talk there, Yota Otachi and Toshiki Saitoh contacted us with an idea how to extend proper intervals in linear time, based on a paper of Deng, Hell, and Huang. Later, this led me to constructing a polynomial time algorithm (based on

LP) for extending unit interval graphs. By this, I solved an open problem from my bachelor's thesis.

Since I have presented our open problems at several open problem sessions, extension of unit interval graphs was independently solved by Ignaz Rutter. During GD 2011, I discussed together with Ignaz and Jan our approach and possible faster combinatorial algorithms avoiding LP, and thus we decided to write a future paper together. The faster algorithm itself and several structural properties of unit interval representations were developed during fruitful discussions with Maria Saumell and Tomáš Vyskočil.

During MCW 2011, I had fruitful discussions with Yota Otachi and Toshiki Saitoh. We improved the running time of the algorithm for extending interval graphs to linear time, and created the NP-completeness reductions for chordal graphs.

I would like to thank Andrew Goodall and Maria Saumell for suggestions concerning introduction of this thesis. I would like to thank all the members and students of the Department of Applied Mathematics for the wonderful environment, support, and excellent background without which I could hardly have written this thesis. Also, I would like to thank Johann Sebastian Bach for his excellent music pieces which helped me greatly while writing this thesis.

Contents

Název práce:	Rozšiřování částečných reprezentací grafů
Autor:	Pavel Klavík
Katedra:	Katedra aplikované matematiky
Vedoucí práce:	Prof. RNDr. Jan Kratochvíl, CSc.
E-mail vedoucího:	honza@kam.mff.cuni.cz
Klíčová slova:	průnikové grafy, rozpoznávání, složitost, algoritmy

Abstrakt:

V této práci se zabýváme geometrickými průnikovými reprezentacemi grafů. Pro danou třídu se známý problém rozpoznávání ptá, jestli graf na vstupu náleží do této třídy. Studujeme zobecnění tohoto problému nazvané *rozšiřování částečných reprezentací.* Vstup je tvořen grafem spolu s částečnou reprezentací, jinými slovy část grafu je předkreslena. Problém se ptá, jestli je možné tuto částečnou reprezentaci rozšířit na reprezentaci celého grafu.

Tento problém studujeme pro třídy intervalových grafů, vlastních intervalových grafů, jednotkových intervalových grafů a chordálních grafů (ve formě reprezentací jako podstromy ve stromě). Popisujeme lineární algoritmy pro první dvě třídy a téměř kvadratický algoritmus pro jednotkové intervalové grafy. Pro chordální grafy uvažujeme různé verze problému a ukazujeme, že skoro všechny jsou NP-úplné.

Přestože třídy vlastních a jednotkových intervalových grafů jsou si rovny, problém rozšiřování částečných reprezentací je rozlišuje. Jednotkové intervalové grafy kladou dodatečné podmínky týkající se přesných pozic intervalů. V práci popisujeme novou strukturu jednotkových intervalových reprezentací, která umožňuje tyto dodatečné podmínky řešit.

Title: Extending Partial Representations of Graphs
Author: Pavel Klavík
Department: Department of Applied Mathematics
Supervisor: Prof. RNDr. Jan Kratochvíl, CSc.
Supervisor's e-mail: honza@kam.mff.cuni.cz
Keywords: intersection graphs, recognition, complexity, algorithms
Abstract:

In this thesis, we study geometric intersection representations of graphs. For a fixed class, the well-known recognition problem asks whether a given graph belongs to this class. We study a generalization of this problem called *partial representation extension*. Its input consists of a graph with a partial representation, so a part of the graph is pre-drawn. The problems asks whether this partial representation can be extended to a representation of the entire graph.

We study this problem for classes of interval graphs, proper interval graphs, unit interval graphs and chordal graphs (in the setting of subtrees-in-tree representations). We give linear-time algorithms for the first two classes and an almost quadratic-time algorithm for unit interval graphs. For chordal graphs, we consider different versions of the problem and show that almost all cases are NP-complete.

Even though the classes of proper and unit interval graphs are known to be equal, the partial representation extension problem distinguishes them. For unit interval graphs, it poses additional restrictions concerning precise positions of intervals, and we describe a new structure of unit interval representations to deal with this.

1 Introduction

Geometric representations of graphs have been studied as a part of graph theory from its very beginning. Euler initiated the study of graphs by studying planar graphs in the setting of three-dimensional polytopes. (But this relation of planar graphs and polytopes was discovered much later by Cauchy. This is also the reason why vertices, edges and faces of graphs have geometrical names.) A theorem of Kuratowski [32] provides the first combinatorial characterization of planar graphs and can be considered as the start of modern graph theory. Nowadays, graphs and their drawing are deeply connected everywhere in graph theory.

One of the fundamental themes of mathematics is that one wants to understand the structure of big objects, and there are many methods designed to work with these big objects. If the object is highly symmetrical, one can try to factorize symmetrical parts to create a smaller 'equivalent' object which is easier to work with. Linear algebra methods can be applied to embed the object into a low dimension and to find fundamental directions in a linearized structure of the object. The technique we study here is finding a good visualization of the object which displays its structure well. For example, a good representation of a graph can visualize very well the information contained in this graph. Figure 1.1 shows examples of graph visualizations.

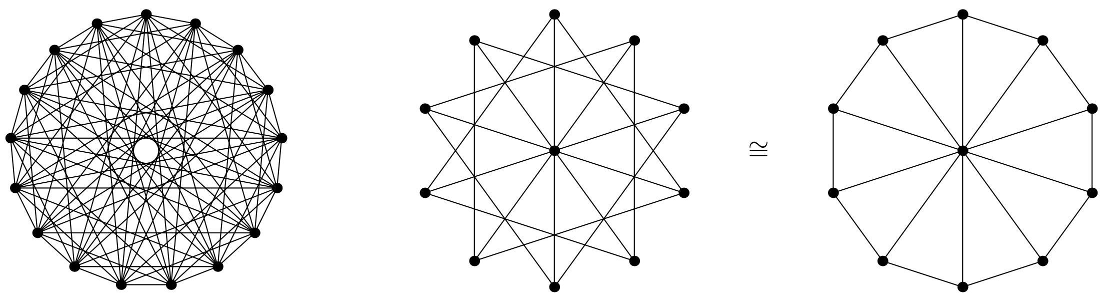

Figure 1.1: Three examples of graph visualizations. The structure of the graph on the left is not very understandable since the graph contains too many edges for this type of drawing. But even the drawing of the graph in the middle containing only a few edges is not very good. Furthermore, it is not obvious that the graph in the middle is isomorphic to the graph on the right, for which the structure of edges is completely clear.

Intersection Representations. There are graphs for which drawings as in Figure 1.1 are not convenient.[1] For example, a graph may contain many edges, which makes these drawings not very understandable even if the structure of the edges is very simple. Therefore it seems reasonable to study different types of drawings tailored to specific types of graphs.

In this thesis, we study intersection representations, which assign geometric objects to vertices of graphs and which encode edges by intersections of these objects. Formally, an intersection representation $\mathcal{R}$ of G is a collection of sets $\{R_v : v \in V(G)\}$ such that $R_u \cap R_v \neq \emptyset$ if and only if $uv \in E(G)$. Since every graph can be represented in this way [34], to obtain interesting classes of graphs we restrict the sets R_v, for instance to some specific geometric objects. For example, interval graphs have intersection representations in which every set R_v is a closed interval of the real line.

Classical examples of intersection-defined classes include interval graphs, circle graphs, permutation graphs, string graphs, convex graphs, and function graphs. As can be seen from the three books [17, 36, 44], geometric intersection graphs have been intensively studied for many reasons. First of all, graphs of these types naturally appear in applications, and sometimes an application directly gives an intersection representation. Also, many hard combinatorial problems can be solved very efficiently on these intersection-defined classes of graphs. Moreover, there are many interesting theoretical results connecting these classes to each other and to the rest of the graph theory.

1.1 The Partial Representation Extension Problem

When one considers a specific class of intersection-defined graphs, it is very natural to study a problem called *recognition*. The recognition problem of a class $\mathcal{C}$, denoted by $\textsc{Recog}(\mathcal{C})$, asks whether an input graph has a representation $\mathcal{R}$ belonging to this class $\mathcal{C}$. The study of recognition has many applications since some problems are easily solvable if we know a representation of the input graph.

For most of the intersection-defined classes, the complexity of their recognition is well-known and deeply understood. For example, interval graphs can be recognized in linear time [6, 9], while recognition of string graphs is NP-complete [30, 42]. (We note that belonging to NP is non-trivial since there are examples of string graphs requiring an exponential number of crossing points in every representation [31].) Our goal is to study the easily recognizable classes and explore if the recognition problem becomes harder when extra conditions are given with the input.

Partial Representation Extension. We study the following question of partial representation extension for intersection-defined classes of graphs. A *partial representation* $\mathcal{R}'$ of G is a representation of an induced subgraph G'. The vertices of G' are called *pre-drawn*. A representation $\mathcal{R}$ of the entire graph G extends the partial representation $\mathcal{R}'$ if $R'_v = R_v$ for every $v \in V(G')$. The partial representation extension for intersection-defined classes, first considered in [29], is the following decision problem:

[1]In these drawing, vertices are represented by points in the plane, and edges are represented by continuous curves connecting pairs of points.

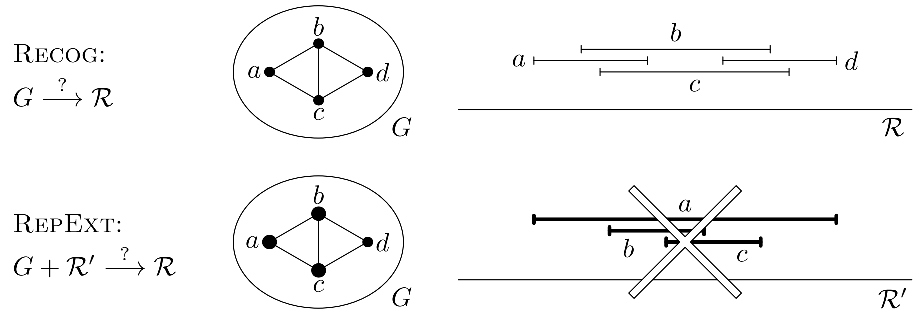
RECOG:
$G \xrightarrow{?} \mathcal{R}$
a
b
c
d
G
a
b
c
d
$\mathcal{R}$
REPEXT:
$G + \mathcal{R}' \xrightarrow{?} \mathcal{R}$
a
b
c
d
G
a
b
c
$\mathcal{R}'$

Figure 1.2: The graph G is an interval graph, but the partial representation $\mathcal{R}'$ is not extendible.

Problem:	REPEXT($\mathcal{C}$) (Partial Representation Extension of $\mathcal{C}$)
Input:	A graph G and a partial representation $\mathcal{R}'$.
Question:	Does G have a representation $\mathcal{R}$ extending $\mathcal{R}'$?

Figure 1.2 compares the recognition problem with the partial representation extension problem. In this thesis we investigate the complexity of partial representation extension for interval graphs, proper interval graphs, unit interval graphs, and chordal graphs.

Concerning other results for REPEXT, the paper [25] studies partial representation extension of permutation and function graphs. It shows that both classes are extendible in polynomial time, and that the problem becomes NP-complete if the partial representation specifies functions only partially. Bläsius and Rutter [5] give another linear-time algorithm for interval graphs extension, even through the paper itself deals with a more general problem and the algorithm is quite involved.

1.2 Classes under Consideration

This thesis has three major chapters, which are mostly independent. In each chapter we deal with the complexity of the partial representation extension problem for different classes using different techniques. We now describe in detail the classes we shall study.

Interval Graphs. In Chapter 2, we deal with *interval graphs*, which are one of the oldest and most studied classes of graphs, introduced by Hajos [19]. A graph is an interval graph if it has a representation $\mathcal{R}$ consisting of closed intervals on the real line, i.e., each vertex v is represented by a closed interval R_v in such a way that $R_u \cap R_v \neq \emptyset$ if and only if $uv \in E(G)$. We denote the class of interval graphs by INT.

Interval graphs have many useful theoretical properties, for example they are perfect and related to path-width decomposition. Most hard combinatorial problems are polynomially solvable for interval graphs: maximum clique, k-coloring, maximum independent set, etc. Also, interval graphs naturally appear in many applications concerning biology, psychology, time scheduling, and archaeology; see for example [40, 45, 3].

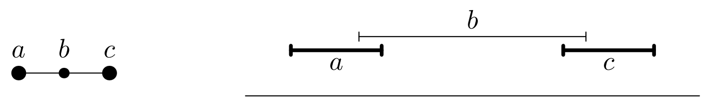

Figure 1.3: A partial representation of a path P_2 representing end-points far apart is extendible only proper intervals, but not by unit intervals.

Proper and Unit Interval Graphs. Chapter 3 deals with two famous subclasses of interval graphs. An interval representation is called *proper* if no interval is a proper subset of another interval (meaning $R_u \subseteq R_v$ implies $R_u = R_v$). An interval representation is called *unit* if the length of each interval is one. The class of *proper interval graphs* ($\mathsf{PROPER\ INT}$) consists of all interval graphs having proper interval representations, whereas the class of *unit interval graphs* ($\mathsf{UNIT\ INT}$) consists of all interval graphs having unit interval representations.

Several linear-time recognition algorithms are known for these classes [33, 20, 8, 7]. Also, solving many problems is even easier in the case of these classes. A famous result of Roberts [39] states that these two classes are equal:

$$\mathsf{PROPER\ INT} = \mathsf{UNIT\ INT}. \tag{1.1}$$

It is easy to see that $\mathsf{UNIT\ INT} \subseteq \mathsf{PROPER\ INT}$. To obtain the other inclusion, we modify a proper interval representation by scaling and shifting.

Most papers concentrate only on proper interval graphs which have an easier combinatorial structure. To the best of our knowledge, there are only a few papers working specifically with unit interval representations, and they either show different proofs of (1.1), or give algorithms constructing unit interval representations directly. For example a very nice paper of Corneil et al. [8] describes a linear-time algorithm constructing a unit interval representation in a grid of size $\frac{1}{n}$ (so each endpoint has the form $\frac{k}{n}$), where n is the number of vertices.

At first, this ignoring of specific properties of unit interval representations does not seem like a big problem. For example, when one works with a problem like vertex coloring or hamiltonicity, exactly the same situation is obtained for unit interval graphs as for proper interval graphs. But when one studies representations restricted in some way, this equivalence does not hold anymore. Indeed, the equivalence is just stating that whenever a graph has a proper interval representation, it also has some other unit interval representation.

In the case of partial representation extension, we need to distinguish these classes; see Figure 1.3. Partial unit interval representations put additional restrictions which we need to deal with, and therefore we develop some additional structure. There is a good lesson one can take out of this: *Whenever there are two equivalent objects, one needs to know precisely under which conditions this equivalence holds.*

Chordal Graphs. In Chapter 4, we deal with *chordal graphs* (also called *triangulated graphs*). One definition of this class states that a graph is chordal if it does not contain an induced cycle of length four or more, i.e., each "long" cycle is triangulated. An equivalent definition, which we use in this thesis, states that chordal graphs are intersection graphs of subtrees of a tree. More precisely, for every chordal graph G

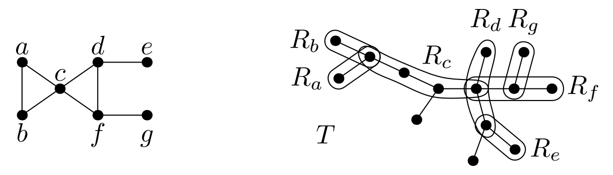

Figure 1.4: An example of a chordal graph with one of its representations.

there exists a tree T and a collection $\mathcal{R} = \{R_v : v \in V(G)\}$ of subtrees of T such that $R_u \cap R_v \neq \emptyset$ if and only if $uv \in E(G)$. For an example of a chordal graph and one of its intersection representations, see Figure 1.4.

The class of chordal graphs is well-studied and has many wonderful properties. Chordal graphs are closed under induced subgraphs and contain so called *perfect elimination schemes*, closely related to optimal reorderings of sparse matrices for Gaussian elimination. Also, chordal graphs are perfect and many hard combinatorial problems are easy to solve on chordal graphs: maximum clique, maximum independent set, k-coloring, etc. Chordal graphs can be recognized in time $\mathcal{O}(n+m)$ [41].

When chordal graphs are viewed as *subtrees-in-tree graphs* (T-in-T), it is natural to consider two other possibilities: *subpaths-in-path graphs* (P-in-P) and *subpaths-in-tree graphs* (P-in-T) which are also called *path graphs.* For example the graph in Figure 1.4 is a path graph but not a subpath-in-path graph. In addition, we consider a proper version of subpath-in-path graphs (PROPER P-in-P) which are P-in-P graphs having a representation such that $R_u \subseteq R_v$ implies $R_u = R_v$.[2]

It is easy to see that

$$\textsf{PROPER P-in-P} = \textsf{PROPER INT} \qquad \text{and} \qquad \textsf{P-in-P} = \textsf{INT}.$$

These subpaths-in-path representations can be viewed as a discretization of representations on the real line and for a refined enough discretization we are able to represent every interval graph as a subpath-in-path graph. Therefore, the classes PROPER P-in-P and P-in-P are recognizable in linear time. The current fastest algorithm for path graph recognition runs in time $\mathcal{O}(nm)$ [15, 43].

So why do we study these classes again separately in Chapter 4? The reason is that the partial representation extension problem puts additional constraints on representations which makes the subpath-in-path classes behave slightly differently compared to their real line counterparts. Actually, the RepExt(PROPER P-in-P) and RepExt(P-in-P) problems are much closer to RepExt(UNIT INT) and very similar techniques can be applied; all these problems involve construction of representations in limited spaces.

It is not immediately clear how to define partial representations of subtrees in a tree, and we analyse four different definitions and show how the complexity changes.

[2]One might also define proper versions of P-in-T and T-in-T classes but these classes are not normally considered. If an arbitrary tree T can be chosen, every representation $\mathcal{R}$ can be modified to a proper representation by attaching leaves to the tree and enlarging all subtrees of $\mathcal{R}$. Of course, if T is restricted (for example by a partial representation), then this is not the case and some interesting differences may appear. These classes seem to be possible directions for future research.

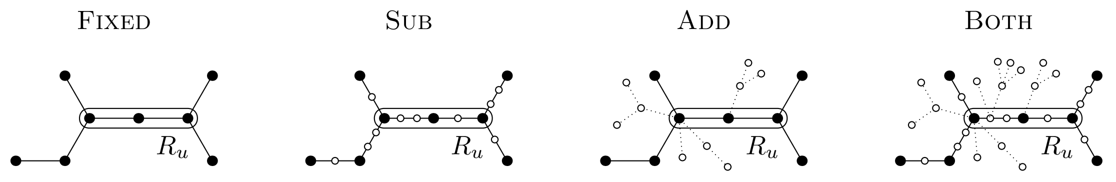

Figure 1.5: Four possible modifications of T' with a single pre-drawn vertex u. The added branches in T are denoted by dots and new vertices of T are denoted by small circles.

A partial representation $\mathcal{R}'$ specifies some tree T' and gives one subtree $R_u \subseteq T'$ for each $u \in V(G')$. The representation $\mathcal{R}$ extending $\mathcal{R}'$ is placed in a tree T which is created by some modification of T'. We consider four possible modifications and get different extension problems:

- FIXED – the tree is not modified at all, i.e., $T = T'$.
- SUB – the tree can only be subdivided, i.e., T is a subdivision of T'.[3]
- ADD – we can add branches to the tree, i.e., T' is a subgraph of T.
- BOTH – we can both add branches and subdivide, i.e., a subgraph of T is a subdivision of T', or in other words T' is a topological minor of T.

For an illustration of the four modifications, see Figure 1.5.

We denote these problems by REPEXT($\mathcal{C}, \mathfrak{T}$) where $\mathfrak{T}$ denotes the type. Constructing a representation in a specified tree T' is interesting even if no subtree is predrawn, i.e., G' is empty; this problem is denoted by RECOG*($\mathcal{C}, \mathfrak{T}$). Clearly, hardness of the RECOG* problem implies the hardness of the corresponding REPEXT problem.

1.3 Motivation

There are several strong reasons for studying partial representation extension problems which we illustrate in this thesis.

Other Extension Problems. The partial representation extension problems belong to a general paradigm of problems where a *partial solution* is given and the task is to extend it. Every *partial solution extension problem* is at least as hard as the original problem without any partial solution, which corresponds to a well-known saying in architecture that it is much easier to build a house from scratch. Sometimes, partial solution extension problems show unusual changes in their complexity. We give a few examples of these problems.

- Every k-regular bipartite graph is k-edge-colorable. But if some edges are precolored, the extension problem becomes NP-complete even for $k = 3$ [11], and even when the input is restricted to planar graphs [35].

[3]Let $xy \in E(T')$ be a subdivided edge (with a vertex z added in the middle). Then also pre-drawn subtrees containing both x and y are modified and contain z as well. So technically in the case of subdivision, it is not true that $R'_u = R_u$ for every pre-drawn interval but from the topological point of view the partial representation is extended.

- A similar situation holds for vertex coloring. In general, k-coloring is NP-complete, but it is solvable efficiently for some simple classes like bipartite graphs or interval graphs. When some vertices are pre-colored, extension of these colorings is NP-complete even if the input graph is restricted to a bipartite graph [21] or an interval graph [4].
- Surprisingly, extension of partial representations of graphs has only recently been considered. For planar graphs, partial representation extension is solvable in linear time [2]. To complete the picture, every planar graph admits a straight-line drawing, but partial representation extension of such representations is NP-complete [37].

The paper [29] gives the first polynomial results concerning extension of partial representations of intersection-defined classes. A common yet quite surprising property of intersection representations is that for most of these classes the partial representation extension problem remains polynomially solvable, unlike the case of partial coloring extension problems.

Better Understanding of Structure. When one wants to solve a partial solution extension problem in polynomial time, a good understanding of the structure of possible solutions seems to be necessary. One can analyse a partial solution and find a solution extending this partial solution using this structure.

Consider again vertex k-coloring of bipartite graphs. For trivial reasons, every bipartite graph is k-colorable for $k \geq 2$. We know this independently of the structure of the graph. But without understanding the structure of possible vertex k-colorings, one can hardly solve the partial coloring extension problem. The result that this problem is NP-hard can be translated as that there is (very likely, unless P = NP) no good structure of possible colorings one can use to solve the extension problem.

So a good property of partial solution extension problems is that if they are solvable in polynomial time, they force one to get a very good insight into the structure of all possible solutions. All algorithms for partial representation extension we know work on the following principle. One works with all possible solutions stored in some efficient way. Then some specific conditions are derived from the partial representation, and tested efficiently for all possible solutions. If some solution satisfies the derived conditions, it can be used as a representation extending the partial representation. If no solution satisfies the conditions, the partial representation is not extendible.

In many cases the structure which can be used is already known and well understood. For interval graphs, one can use a tree structure called PQ-trees which compactly represents all possible solutions. For proper interval graphs, we use the known 'uniqueness' of the solution. The paper [25] uses for permutation and function graphs a known relation of all solutions with the modular decomposition of the input graph. For unit interval graphs, no sufficient structure was known and we describe completely new structural properties in Sections 3.2.1, 3.3.2, 3.3.3, and 3.3.4. Lastly, our hardness results concerning chordal graph extension can be interpreted as saying that there is no strong structure one could use. In many cases the partial representation extension problems of chordal graphs (and their subclasses) can solve very hard problems concerning integer partitioning.

To recapitulate, we believe that the partial representation extension problem is interesting because when one wants to attack this problem, a new structure of all representations has to be discovered and used (if it is not already known). Motivations like these appear all over mathematics. Hilbert once stated about Fermat's Last Theorem that a good mathematical problem is like a goose hatching golden eggs; when people try to attack it, new structures and properties are discovered and understood. Indeed, this specific case of unsolvability of diophantine equations is mostly interesting because by solving it many new techniques and structures in algebraic number theory were discovered. Of course, we are not claiming that the partial representation extension problem is comparable to Fermat's Last Theorem, just that the property 'to solve the problem, you need to get a better understanding of the structure' is generally desirable.

Applications and New Techniques. Another nice property of partial representation extension problems is that algorithms or techniques developed for them can be applied to other problems. For example, using our structural properties one can easily answer the following algorithmic question: Given a unit interval graph, what is the length of a smallest possible segment of the real line such that the representation can be constructed within this segment? Additionally, the opposite question of what is the size of the widest possible unit interval representation of a given connected graph can be answered. For example, for a path having $2n$ vertices, the length $n + \varepsilon$ for any $\varepsilon > 0$ is sufficient, and the widest representation has length $2n$. We are not aware of any previous structural or algorithmic results of this type.

Also, a recent related problem of simultaneous graph representations was introduced by Jampani and Lubiw [22, 23]. The input gives graphs $G_1, \dots, G_k$ with common vertices I, meaning $V(G_i) \cap V(G_j) = I$ for every $i \neq j$. The problem asks whether there exist representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ such that each $\mathcal{R}^i$ represents G_i and the representations are equal on I, meaning $R^i_v = R^j_v$ for every $v \in I$ and $i \neq j$. We denote this problem by $\textsc{SimRep}(\mathcal{C})$. For an example, see Figure 1.6.

Simultaneous representations are closely related to partial representations and we discuss this connection in the concluding chapter. Using this relation, the paper [5] solves $\textsc{RepExt}(\textsf{INT})$ in time $\mathcal{O}(n+m)$ by solving another simultaneous representations problem. The simultaneous representations problem similarly distinguishes between proper and unit interval graphs. We already have some partial results (not written anywhere yet) which show than many of the techniques developed in Chapter 3 can

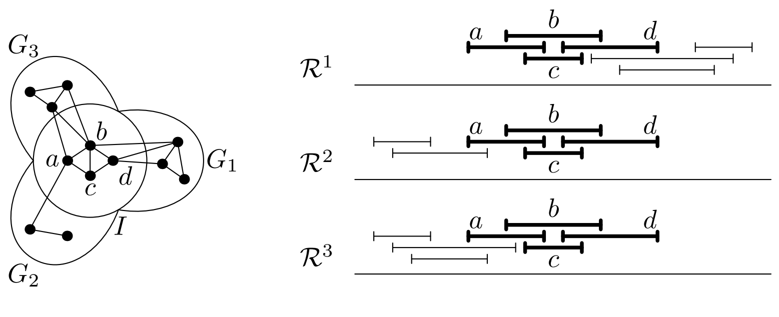

Figure 1.6: Simultaneous interval representations of three graphs G_1, G_2 and G_3 with $I = \{a, b, c, d\}$.

be applied to the SimRep(PROPER INT) and SimRep(UNIT INT) problems, and that both problems can be solved in polynomial time if the graphs $G_1, \dots, G_k$ are connected.

There is also another connection of the partial representation extension of interval graphs to much harder problems concerning Allen algebras, and again we discuss this connection in more details in the concluding chapter.

1.4 Results of This Thesis

We present a short list of the main results of this thesis, listed by chapter.

1.4.1 Results of Chapter 2

In this chapter, we deal with the partial representation extension problem for interval graphs. Our main result is:

Theorem 1.1. *The problem* RepExt(INT) *is solvable in time $\mathcal{O}(n+m)$ where n is the number of vertices and m is the number of edges.*

We note that to get this running time we make some minor and very natural assumptions on the input, more details in Section 1.5. Our algorithm is based on a PQ-tree approach for recognition of interval graphs [6]. We derive a partial ordering from the partial representation and test whether this partial ordering is compatible with the PQ-tree. We show that the representation can be extended if and only if the partial ordering is compatible.

1.4.2 Results of Chapter 3

We deal with the classes of proper and unit interval graphs. For the first class, we prove:

Theorem 1.2. *The problem* RepExt(PROPER INT) *is solvable in time $\mathcal{O}(n+m)$ where n is the number of vertices and m is the number of edges.*

Again, some natural assumptions on the input are necessary; see Section 1.5. Our approach is based on the 'uniqueness' of the left-to-right ordering of intervals in every representation. We give a simple characterization of all extendible instances, and the algorithm just needs to test this characterization in time $\mathcal{O}(n + m)$.

Next, we deal with the RepExt(UNIT INT) problem. Actually, we mostly deal with a more general problem called *bounded representation* of unit interval graphs, BoundRep for short:

Problem:	BoundRep (Bounded Representation of UNIT INT)
Input:	A graph G and some lower/upper bounds for the positions of some intervals.
Question:	Does G have a unit interval representation which respects the bounds?

Unfortunately, this problem is hard in general.

Theorem 1.3. *The* BOUNDREP *problem is* NP*-complete.*

In each representation of a graph with c components, these components are ordered from left to right according to the position of their intervals; see Section 1.5 for details. Let us denote this ordering $\blacktriangleleft$, so $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$. Also let $D(r)$ denote the complexity of dividing numbers of length r in binary. Our computational model is the Turing machine and the best known algorithm achieves $D(r) = \mathcal{O}(r \log r 2^{\log^* r})$ [13]. We get the following main result of this chapter:

Theorem 1.4. *The* BOUNDREP *problem with a prescribed ordering* $\blacktriangleleft$ *can be solved in time* $\mathcal{O}(n^2 + nD(r))$*, where* r *is the size of the input.*

The algorithm makes $\mathcal{O}(n^2)$ combinatorial iterations, each of them taking time $\mathcal{O}(1)$. The additional time $\mathcal{O}(nD(r))$ is used for arithmetic operations with the bounds.

This result gives the following corollaries:

Corollary 1.5. *The* BOUNDREP *problem can be solved in time* $\mathcal{O}((n^2 + nD(r))c!)$*, where* c *is the number of components of* G *and* r *is the size of the input.*

Corollary 1.6. *The* REPEXT(UNIT INT) *problem can be solved in time* $\mathcal{O}(n^2 + nD(r))$*, where* r *is the size of the input.*

1.4.3 Results of Chapter 4

We consider the complexity of the RECOG* and REPEXT problems for chordal graphs and its three subclasses and all four types. Our results are displayed in Figure 1.7.

- All NP-complete results are reduced from the 3-PARTITION problem. The reductions are very similar and the basic case is REPEXT(PROPER P-in-P, FIXED). Also, the reductions are very close to the reduction in Theorem 1.3.
- Polynomial cases for PROPER P-in-P and P-in-P are based on known algorithms for recognition and extension, and use similar techniques as the ones described in Chapters 2 and 3. Unlike for the real line the space in T is limited, and we adapt the algorithm for the specific problems.

Also, we study the parametrized complexity of these problems with respect to three parameters: The number of pre-drawn subtrees k, the number of components c and the size t of the tree T'. In some cases, the parametrization does not help and the problem is NP-complete even if the value of the parameter is zero or one. In other cases, the problems are fixed-parameter tractable (FPT), W[1]-hard or in XP.

The main result concerning parametrization is the following. The BINPACKING problem is a well-known problem concerning integer partitions; more details in Section 4.1.3. For two problems A and B, we denote by $A \le B$ a polynomial reduction and by $A \le_{\rm wtt} B$ a weak truth-table reduction which means that more instances of the problem B can be used to solve A. (More precisely, this number of B-oraculum questions used to solve A is bounded by a computable function.)

Theorem 1.7. BINPACKING $\le$ REPEXT(PROPER INT, FIXED) $\le_{\rm wtt}$ BINPACKING *where the weak truth-table reduction needs to solve* 2^k *instances of* BINPACKING.

		PROPER P-in-P	P-in-P	P-in-T	T-in-T
FIXED	RECOG*	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$	NP-complete	NP-complete
	REPEXT	NP-complete	NP-complete	NP-complete	NP-complete
SUB	RECOG*	$\mathcal{O}(n+m)$ [33, 8]	$\mathcal{O}(n+m)$ [6, 9]	NP-complete	NP-complete
	REPEXT	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$	NP-complete	NP-complete
ADD	RECOG*	$\mathcal{O}(n+m)$ [33, 8]	$\mathcal{O}(n+m)$ [6, 9]	$\mathcal{O}(nm)$ [15, 43]	$\mathcal{O}(n+m)$ [41]
	REPEXT	$\mathcal{O}(n+m)$	NP-complete	NP-complete	NP-complete
BOTH	RECOG*	$\mathcal{O}(n+m)$ [33, 8]	$\mathcal{O}(n+m)$ [6, 9]	$\mathcal{O}(nm)$ [15, 43]	$\mathcal{O}(n+m)$ [41]
	REPEXT	$\mathcal{O}(n+m)$	$\mathcal{O}(n+m)$ [5]	**open**	NP-complete

Figure 1.7: The complexity of the different problems for all four considered classes. Results without references are new results of this thesis.

1.5 Notation and Preliminaries

As usual, we reserve n for the number of vertices and m for the number of edges of the graph G. We denote the set of vertices by $V(G)$ and the set of edges by $E(G)$. For a vertex v, we let $N(v) = \{x, vx \in E(G)\}$ denote the open neighborhood of v, and $N[v] = N(v) \cup \{v\}$ its closed neighborhood. We call maximal connected subgraphs of a graph its *components*.

For each interval R_u, we denote by ℓ_u and r_u the positions of its left and right endpoints. For INT, PROPER INT and UNIT INT classes, these are positions on the real line. For PROPER P-in-P and P-in-P classes, these are vertices of the path T. For numbered vertices $v_1, \dots, v_n$, we abbreviate endpoints of v_i just as ℓ_i and r_i.

Topology of Components. The following property works quite generally for many intersection-defined classes of graphs, and works for all classes studied in this thesis. The only condition required for this property is that the sets R_v are connected subsets of some topological space, for example $\mathbb{R}^k$. (As a negative example, this property does not hold for 2-interval graphs. A graph is a 2-interval graph if each R_v is union of two closed intervals.) Let C be a component of G. Then the property is that for each representation $\mathcal{R}$, $\bigcup_{v \in C} R_v$ is a connected subset of the space, and we call this subset *area of* C. Clearly, the areas of components are pairwise disjoint.

For classes of interval graphs (both on the real line, and as subpaths-in-path graphs), the areas of the components have to be ordered from left to right. Let us denote this ordering $\blacktriangleleft$, so we have $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$. For different representations $\mathcal{R}$, we can have different orderings $\blacktriangleleft$. When there is no restriction on $\mathcal{R}$, it is possible to create a representation in every one of $c!$ possible orderings.

(Un)located Components. A partial representation $\mathcal{R}'$ divides the components into two types. A component C is called *located* if it contains at least one pre-drawn vertex, and it is called *unlocated* otherwise. For located components, we have partial information about their position. The positions of unlocated components in a representation are much more free.

For classes of interval graph, the located components are ordered from left to right. A trivial necessary condition for an extendible partial representation is that the pre-drawn intervals of each component appear consecutively. Indeed, if $u, v \in C$, $w \in C'$ and R_w is between R_u and R_v where C and C' are two distinct components, then the partial representation is clearly not extendible. For every representation $\mathcal{R}$ extending the partial representation, the ordering $\blacktriangleleft$ has to agree with this left-to-right order of the located components.

For problems as RepExt(PROPER INT) or RepExt(UNIT INT), unlocated components are not very interesting; they can be placed on the real line far to the right, without interfering with the partial representation at all. For problems in Chapter 4, the space in T is limited and the unlocated components have to be placed somewhere. In many cases, the existence of unlocated components is necessary for some problems to be NP-complete.

Remarks Concerning Input. To obtain the linear-time algorithms described in this thesis, we need some reasonable assumption on the partial representation which is given by the input. Similarly, most of the graph algorithms cannot achieve better running time than $\mathcal{O}(n^2)$ if the input graph is given by an adjacency matrix instead of a list of neighbors for each vertex.

First, we consider classes INT and PROPER INT. We say that the partial representation is *sorted* if it gives the pre-drawn intervals sorted from left to right. We say that the partial representation is *normalized* if all its endpoints are integers between 0 and $2k$ (where k is the number of pre-drawn intervals). Notice that from a normalized representation, we can produce a sorted representation in time $\mathcal{O}(k)$, and vice versa.

To get linear time in Theorems 1.1 and 1.2, we assume that the input partial representation is given sorted or normalized. If this assumption is not satisfied, the algorithms would need additional time $\mathcal{O}(k \log k)$ to sort the partial representation. This assumption makes sense since the solutions of RepExt for these classes depends only on the topology of the partial representation, not on the precise positions of endpoints.

Concerning classes PROPER P-in-P and P-in-P, we assume an input of the following type. For the path T in which the representation should be constructed, the input only specifies its length which is polynomial with respect to n and m. Then for every pre-drawn subpath R_v, only two numbers ℓ_v and r_v are given. Moreover, we assume that these pre-drawn endpoints are sorted from left to right, otherwise additional time $\mathcal{O}(k \log k)$ is required.

2 Extending Interval Graphs

In this chapter, we solve the problem $\textsc{RepExt}(\mathsf{INT})$ in time $\mathcal{O}(n+m)$. In the first section, we describe how PQ-trees work and how to apply interval orders on them fast. The problem solved in the first section is an example of a more general type of problems: Having a set of elements with some tree structure and a partial ordering, we want to order the elements according to this partial ordering while maintaining the tree structure.

In the second section, we show how to solve the partial representation extension problem using PQ-trees and interval orders. Let G be an input graph of $\textsc{RepExt}(\mathsf{INT})$. It is important to stress out that this interval order is based on a completely different interval graph H where the vertices of H correspond to the maximal cliques of G. This distinction between G and H becomes more clear since G is represented by closed intervals and H is represented by open intervals.

2.1 PQ-trees and Interval Orders

To describe PQ-trees, we start with a motivational problem. An input of the *consecutive ordering problem* is a set E of elements and restricting sets $S_1, S_2, \ldots, S_k$. The task is to find a (linear) ordering of E such that every S_i appears consecutively (as one block) in this ordering.

Example 2.1. Consider elements $E = \{a,b,c,d,e,f,g,h\}$ and restricting sets $S_1 = \{a,b,c\}$, $S_2 = \{d,e\}$, and $S_3 = \{e,f,g\}$. For example, orderings $abcdefgh$ and $fgedhacb$ are feasible. On the other hand, orderings $\underline{ac}defg\underline{b}h$ (violates S_1) and $d\underline{ef}h\underline{g}abc$ (violates S_3) are not feasible.

PQ-trees. A PQ-tree is a tree structure that allows to solve consecutive ordering efficiently. Moreover, it stores all feasible orderings for a given input.

The leaves of the tree correspond one-to-one to the elements of E. The inner nodes are of two types: The *P-nodes* and the *Q-nodes*. For every node, the order of its children is fixed. A PQ-tree T represents one ordering $<_T$, given by the ordering of the leaves from left to right, see Figure 2.1.

To obtain other feasible orderings, we can reorder children of inner nodes. Children of a P-node can be reordered in an arbitrary way. On the other hand, we can only

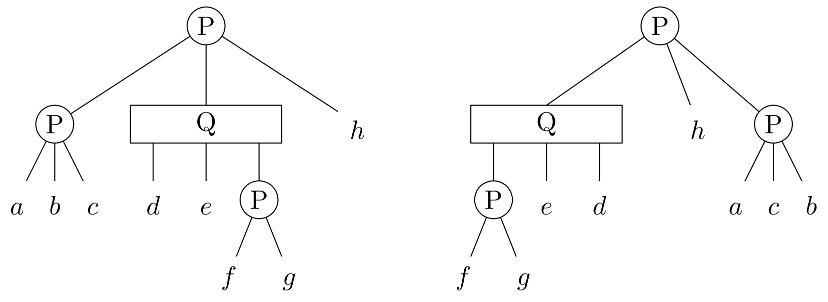

Figure 2.1: PQ-trees representing orderings $abcdefgh$ and $fgedhacb$.

reverse an order of children of a Q-node. Two trees are equivalent if we can change one tree to the other only using these operations. For example, the trees in Figure 2.1 are equivalent. Every equivalence class of PQ-trees corresponds to all the orderings feasible for some input sets. The equivalence class of the PQ-trees in Figure 2.1 corresponds to the input sets in Example 2.1.

For the purpose of this thesis, we only need to know that a PQ-tree can be constructed in time $\mathcal{O}(e + k + t)$ where e is the number of elements of E, k is the number of restricting sets and t is the total size of restricting sets. Booth and Lueker [6] describe details of their construction.

2.1.1 Applying a Partial Ordering

Suppose that T is a PQ-tree and we have a partial ordering $\lhd$ of its elements (leaves). We say that a reordering T' of the PQ-tree T is *compatible* with $\lhd$ if the ordering $<_{T'}$ extends $\lhd$, meaning $a \lhd b$ implies $a <_{T'} b$.

Problem:	Reorder$(T, \lhd)$
Input:	A PQ-tree T and a partial ordering $\lhd$.
Question:	Is it possible to reorder T to T', compatibly with $\lhd$?

The algorithm we are going to describe works even for a general relation $\lhd$. For example, $\lhd$ does not have to be transitive (as in example in Figure 2.2) or even acyclic (but in such a case, of course, no solution exists).

Proposition 2.2. *The problem* Reorder$(T, \lhd)$ *is solvable in in time $\mathcal{O}(e + m)$, where e is the number of elements and m is the number of comparable pairs in $\lhd$.*

A PQ-tree defines some hierarchical structure on its elements. We start with a simple lemma which states that we can try to solve the problem locally (inside of some subtree) and this local solution will always be correct; either there exists no solution of the problem at all, or our local solution can be extended for the whole tree.

Lemma 2.3. *Let S be a subtree of a PQ-tree T. If T can be reordered compatibly with $\lhd$ then every local reordering of the subtree S compatible with $\lhd$ can be extended to a reordering of the whole tree T compatible with $\lhd$.*

Proof. Let T' be a reordering of the whole PQ-tree T compatible with $\lhd$. Notice that all elements of S appear consecutively in $<_{T'}$. Therefore, we can replace this local ordering of S by any other order satisfying all conditions given by $\lhd$. We obtain another reordering of the whole tree T which is compatible with $\lhd$ and extends the local ordering of S as requested. $\square$

This gives the following greedy algorithm. We represent the ordering $\lhd$ by a digraph having m edges. We reorder the nodes from bottom to the root and modify the digraph by contractions. When we finish reordering of a subtree, the order is fixed and never changed in the future; by Lemma 2.3, either this local reordering will be extendible, or there is no correct reordering of the whole tree at all. When we finish reordering of a subtree, we contract all its vertices. We process a node of the PQ-tree when all its subtrees are already processed and represented by single vertices in the digraph.

For a P-node, we check whether the subdigraph induced by the vertices corresponding to the children of the P-node is acyclic. If it is acyclic, we reorder the children according to a topological sorting. Otherwise, there exists a cycle, no feasible ordering exists and the algorithm returns "no". For a Q-node, there are two possible orderings. All we need is love, and to check whether one of them is feasible. For a pseudocode, see Algorithm 1 in Appendix A.1. For an example, see Figure 2.2.

Proof of Proposition 2.2. We use the described algorithm. We need to argue its correctness. The algorithm processes the tree from bottom to the top. For every subtree S, it finds some reordering of S according to $\lhd$. If no such reordering of S can be

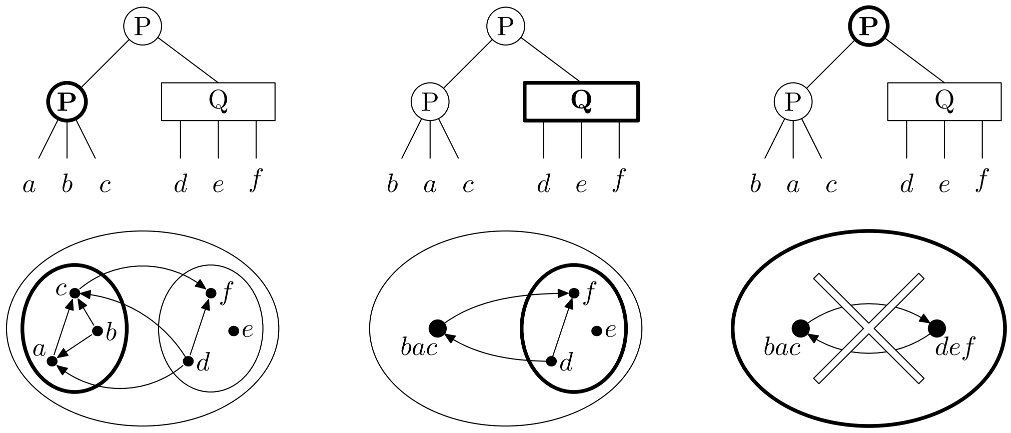

Figure 2.2: We show an example how the reordering algorithm works, the order of the operations is from left to right. First, we reorder the highlighted P-node of the PQ-tree on the left. The subdigraph induced by a, b and c has a topological sorting $b \to a \to c$. We contract these vertices into the vertex *bac*. Next, we keep the order of highlighted Q-node and contract its children into the vertex *def*. When we reorder the root P-node, we obtain a cycle between *bac* and *def*, and the algorithm outputs "no". Notice that the original digraph $\lhd$ is acyclic, just it is not compatible with the structure of the PQ-tree.

found, it is not possible to order the whole tree according to $\lhd$. And if it is possible, every reordering of S is correct according to Lemma 2.3.

The algorithm can be implemented in linear time depending on the size of the PQ-tree and the partial ordering $\lhd$ which is $\mathcal{O}(e+m)$. Each edge of the digraph $\lhd$ is processed exactly once and then it is contracted. $\square$

2.1.2 Applying an Interval Order

Let $\{I_v : v \in V(H)\}$ be a representation of an interval graph H using open intervals.[1] Once again, we note that this graph H is a completely different graph from the input graph G of RepExt(INT) which we are going to solve later; the vertices of H correspond to the elements (leaves) of the PQ-tree which later correspond to the maximal cliques of G.

An interval representation of H defines the following partial order on $V(H)$ which is called an *interval order*. If two intervals I_u and I_v do not intersect, there is a natural ordering between them: One is on the left and the other one is on the right. Let us denote this interval order by $\lhd$. For $u, v \in V(H)$, we define $u \lhd v$ if and only if $r_u \le \ell_v$. For example, in Figure 1.3 we obtain an interval order in which $a \lhd c$ but b is incomparable to both a and c.

There are many interesting relations between interval graphs and interval orders. The study of interval orders has the following motivation. Suppose that intervals of H correspond to events and each interval describes when the corresponding event could happen in the timeline. Then if $a \lhd b$, we know for sure that the event a happened before the event b. If two intervals intersect, we do not have any information about the order of their events. Nevertheless, for purpose of this paper, we only need to know the definition of interval orders. For more information, see survey [46].

Reordering PQ-trees. Let $\lhd$ be an interval order of e elements, for which we know a normalized interval representation having all endpoints as integers between 0 and $2e$. The normalized representation allows us to take any subset of endpoints, to sort it in linear time corresponding to the size of the subset and then process the subset in this sorted order. For such $\lhd$, we show that we can solve Reorder$(T, \lhd)$ faster:

Proposition 2.4. *If $\lhd$ is an interval order with a normalized representation, we can solve the problem* Reorder$(T, \lhd)$ *in time $\mathcal{O}(e)$ where e is the number of elements of T.*

The general outline of the algorithm is exactly the same as before. We process the nodes of the PQ-tree from bottom to the root and reorder them according to local conditions. Using normalized interval representation of an interval order, we can implement all steps faster then before.

We are not going to construct the digraph explicitly, and therefore we are not doing any contractions. Instead, we are going to work with sets of intervals and compare them with respect to $\lhd$ fast. When we process a node, its children correspond

[1]For purposes of the following section, we allow empty intervals with $\ell_v = r_v$.

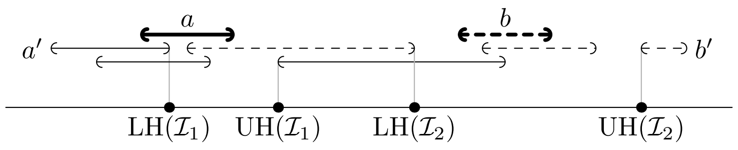

Figure 2.3: Normal intervals belong to $\mathcal{I}_1$, dashed intervals belong to $\mathcal{I}_2$. If $a \lhd b$, then also $a' \lhd b'$.

to sets $\mathcal{I}_1, \dots, \mathcal{I}_k \subseteq V(H)$ we already processed before. The reordering is going to use specific properties of an interval order. When reordering of the node is done, we just put all the sets together:

$$\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \dots \cup \mathcal{I}_k.$$

Comparing Subtrees. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be sets of intervals. We naturally say $\mathcal{I}_1 \lhd \mathcal{I}_2$ if there exist $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$ such that $a \lhd b$. Using the interval representation, we can test whether $\mathcal{I}_1 \lhd \mathcal{I}_2$ in a constant time. The following lemma states that we just need to compare the "left-most" interval of $\mathcal{I}_1$ with the "right-most" interval of $\mathcal{I}_2$.

Lemma 2.5. *Suppose that $a \lhd b$, $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$. Then for every $a' \in \mathcal{I}_1, r_{a'} \le r_a$ and every $b' \in \mathcal{I}_2, \ell_b \le \ell_{b'}$ also $a' \lhd b'$.*

Proof. From definition, $x \lhd y$ if and only if $r_x \le \ell_y$. We have $r_{a'} \le r_a \le \ell_b \le \ell_{b'}$ and thus $a' \lhd b'$. See Figure 2.3, handles are explained later. □

Using the previous lemma, we just need to compare a' having the left-most $r_{a'}$ to b' having the right-most $\ell_{b'}$ since $\mathcal{I}_1 \lhd \mathcal{I}_2$ if and only if $a' \lhd b'$. To simplify the description, we define handles of $\mathcal{I}$, a *lower handle* and *upper handle*:

$$\mathrm{LH}(\mathcal{I}) = \min\{r_x \mid x \in \mathcal{I}\} \qquad \text{and} \qquad \mathrm{UH}(\mathcal{I}) = \max\{\ell_x \mid x \in \mathcal{I}\}.$$

Notice that $\mathrm{LH}(\mathcal{I}) < \mathrm{UH}(\mathcal{I})$ if and only if $\mathcal{I}$ is not a clique. Using handles, we can compare sets of intervals fast: $\mathcal{I}_1 \lhd \mathcal{I}_2$ if and only if $\mathrm{LH}(\mathcal{I}_1) \le \mathrm{UH}(\mathcal{I}_2)$. For an example, see Figure 2.3. For each processed subtree, we are going to remember just these handles, we do not need to remember which specific intervals were contained in the subtree. This is a replacement of the contraction operation used before.

Reordering Nodes. Now, we describe how to reorder children of a processed node fast. Let $\mathcal{I}_1, \dots, \mathcal{I}_k$ be sets of intervals of the subtrees of this node for which we know handles. We call a linear ordering $<$ of sets $\mathcal{I}_1, \dots, \mathcal{I}_k$ *topological sorting* if $\mathcal{I}_i \lhd \mathcal{I}_j$ implies $\mathcal{I}_i < \mathcal{I}_j$ for every $i \ne j$. If the processed node is a P-node, we need to find any topological sorting. If it is a Q-node, we need to test whether the current ordering or its reversal is a topological sorting.

Since the interval representation is normalized and all the handles are at positions of the endpoints, we can sort all handles (both lower and upper) of $\mathcal{I}_1, \dots, \mathcal{I}_k$ from left to right in time $\mathcal{O}(k)$. The handles can share coordinates and we deal with ties as follows: First, we place the lower handles of a given coordinate (in any order) and then we place the upper handles of the same coordinate (again, in any order). Now, $\mathcal{I}_i \lhd \mathcal{I}_j$ if $\mathrm{LH}(\mathcal{I}_i)$ is on the left of $\mathrm{UH}(\mathcal{I}_j)$ in the ordering. For an example, see Figure 2.4.

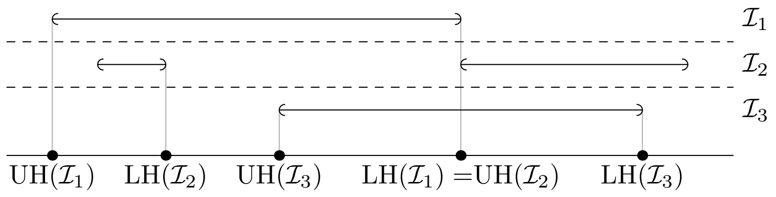

Figure 2.4: For sets $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$, we get a common order of handles $\mathrm{UH}(\mathcal{I}_1) \le \mathrm{LH}(\mathcal{I}_2) \le \mathrm{UH}(\mathcal{I}_3) \le \mathrm{LH}(\mathcal{I}_1) \le \mathrm{UH}(\mathcal{I}_2) \le \mathrm{LH}(\mathcal{I}_3)$. Notice that $\lhd$ is not transitive: $\mathcal{I}_1 \lhd \mathcal{I}_2$ and $\mathcal{I}_2 \lhd \mathcal{I}_3$ but $\mathcal{I}_1 \not\lhd \mathcal{I}_3$. The only topological sorting is $\mathcal{I}_1 < \mathcal{I}_2 < \mathcal{I}_3$.

An element $\mathcal{I}_j$ is *minimal* if there is no $\mathcal{I}_i \lhd \mathcal{I}_j$. The following lemma allows to find a minimal element fast:

Lemma 2.6. *Let $\mathcal{I}_j$ be an element. It is a minimal element if and only if there is no lower handle other than* $\mathrm{LH}(\mathcal{I}_j)$ *on the left of* $\mathrm{UH}(\mathcal{I}_j)$.

Proof. Notice that $\mathcal{I}_i \lhd \mathcal{I}_j$ if and only if $\mathrm{LH}(\mathcal{I}_i) \le \mathrm{UH}(\mathcal{I}_j)$. There is no such $\mathcal{I}_i$ if and only if there is no $\mathrm{LH}(\mathcal{I}_i)$ on the left of $\mathrm{UH}(\mathcal{I}_j)$. □

We can use this lemma to identify all minimal elements. If the ordering starts with two lower handles, there exists no minimal element. If the first element of the ordering is $\mathrm{LH}(\mathcal{I}_i)$ than $\mathcal{I}_i$ is a unique candidate for a minimal element. We just need to check whether there is some other $\mathrm{LH}(\mathcal{I}_j)$ before $\mathrm{UH}(\mathcal{I}_i)$, and if so, no minimal element exists.[2] If the common ordering starts with a group of upper handles, we have several candidates for a minimal element: All $\mathcal{I}_i$'s of these upper handles are minimal elements and maybe $\mathcal{I}_j$ of the following lower handle $\mathrm{LH}(\mathcal{I}_j)$; $\mathcal{I}_j$ is a minimal element if there is no other lower handle on the left of $\mathrm{UH}(\mathcal{I}_j)$.

Now, suppose that $\mathcal{I}_j$ is a minimal element and there is a topological sorting starting with $\mathcal{I}_i$. Then we can modify this sorting by moving $\mathcal{I}_j$ to the front, and we obtain another topological sorting starting with $\mathcal{I}_j$. So we can construct some topological sorting as follows. We always detect one minimal element, remove its handles, append the minimal element to the constructed topological sorting and repeat the procedure for the remaining elements. And if in some step no minimal element exists, we know that no topological sorting exists.

For a P-node, we just need to find any topological sorting by repeated removing of minimal elements. For a Q-node, we just test whether the current ordering or its reversal is a topological order; by going through the given ordering, checking whether each element is a minimal elements and then removing its handles from the ordering. In both cases, if we find a correct topological sorting, we use it reorder the children of the node. Otherwise, the reordering is not possible and the algorithm fails in this node. We are able to do the reordering of the node in time $\mathcal{O}(k)$.

Joining Subtrees. After processing a node, we join several subtrees into one subtree by recalculating handles. Let $\mathcal{I}_1, \dots, \mathcal{I}_k$ are sets of intervals corresponding to subtrees

[2]This can be done a in constant time if we remember in each moment position of two left-most lower handles in the ordering and update it after removing one of them from the ordering.

of the node. Then for the joined subtree $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots \cup \mathcal{I}_k$ we calculate handles as follows:

$$\mathrm{LH}(\mathcal{I}) = \min\bigl\{\mathrm{LH}(\mathcal{I}_i)\bigr\} \qquad \text{and} \qquad \mathrm{UH}(\mathcal{I}) = \max\bigl\{\mathrm{UH}(\mathcal{I}_i)\bigr\}.$$

Notice that this exactly corresponds to the definition of handles but we can compute them in time $\mathcal{O}(k)$ instead of $\mathcal{O}(|\mathcal{I}|)$.

Putting Together. We just put all the parts together exactly as before; for a pseudocode of the whole algorithm see Section A.2. The algorithm allow us to find a reordering of the PQ-tree T according to an interval order $\lhd$ in time $\mathcal{O}(e)$:

Proof of Proposition 2.4. The algorithm is correct since it is processing the tree in exactly the same manner as the algorithm for general orders. First, the relation $\lhd$ on sets $\mathcal{I}_1$ and $\mathcal{I}_2$ of intervals corresponds to existence of an edge after contracting the vertices of $\mathcal{I}_1$ and $\mathcal{I}_2$ in the digraph. Then, the topological sorting is correctly constructed using minimal elements if it exists, by Lemma 2.6.

Concerning the time complexity, we already discussed that we are able to compare sets using handles in a constant time, by Lemma 2.5. We spend time $\mathcal{O}(k)$ in each node with k children. Thus the total time complexity of the algorithm is linear in the size of the tree, which is $\mathcal{O}(e)$. □

2.2 Extending Interval Graphs

In this section, we describe an algorithm solving RepExt(INT) in time $\mathcal{O}(n+m)$. Interval representations have closed intervals. Intervals may share the endpoints and may have zero lengths but this is just a subtle change of the algorithm.

We first describe recognition of interval graphs. Then we show how to modify the PQ-tree approach to solve RepExt(INT).

2.2.1 Recognition using PQ-trees

Recognition of interval graphs in linear time was a long-standing open problem, first solved by Booth and Lueker [6] using PQ-trees. Nowadays, there are two main approaches to recognition in linear time. The first one finds a feasible ordering of the maximal cliques which can be done using PQ-trees. The second approach uses surprising properties of the lexicographic breadth-first search, searches through the graph several times and constructs a representation if the graph is an interval graph [9].

We modify the PQ-tree approach to solve RepExt(INT) in time $\mathcal{O}(n+m)$. Recall the PQ-trees from Section 2.1.

Maximal Cliques. The PQ-tree approach is based on the following characterization of interval graphs, due to Fulkerson and Gross [12]:

Lemma 2.7 (Fulkerson and Gross). *A graph is an interval graph if and only if there exists an ordering of the maximal cliques such that for every vertex the cliques containing this vertex appear consecutively in this ordering.*

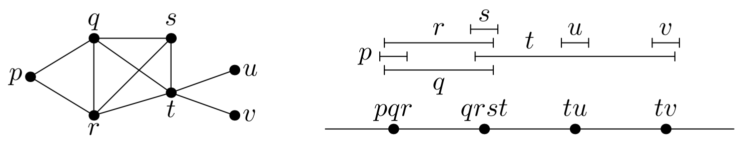

Figure 2.5: An interval graph and one of its representations with denoted clique-points.

Consider an interval representation of an interval graph. For each maximal clique, consider the intervals representing the vertices of this clique and select a point in their intersection. (We know that this intersection is non-empty because intervals of the real line have the Helly property.) We call these points *clique-points*. For an illustration, see Figure 2.5. The ordering of the clique-points from left to right gives the ordering required by Lemma 2.7. Every vertex appears consecutively since it is represented by an interval. For a clique a, we denote the assigned clique-point by $\mathrm{cp}(a)$.

On the other hand, given an ordering of the maximal cliques, we place clique-points in this ordering on the real line. Each vertex is represented by the interval containing exactly the clique-points of the cliques containing this vertex. In this way, we obtain a valid interval representation of the graph.

Recognition Algorithm. The maximal cliques of an interval graph have in total $\mathcal{O}(n+m)$ vertices and can be found in linear time [41]. A feasible ordering of the maximal cliques can be found in linear time using PQ-trees. Recall their definition from Section 2.1. Elements of E are maximal cliques of an interval graph. For each vertex v, we introduce a restricting set S_v containing all the maximal cliques containing this vertex. Using PQ-trees, we can find a feasible ordering of maximal cliques and recognize an interval graph in time $\mathcal{O}(n+m)$.

2.2.2 Modification for RepExt

In the rest of the section, we call maximal cliques just as cliques. We first sketch the algorithm. We construct a PQ-tree for the input graph, completely ignoring the given partial representation. The partial representation gives another restriction—an interval order $\lhd$ of the cliques. Using the algorithm described in Section 2.1.2, we try to reorder the PQ-tree according to $\lhd$ in time $\mathcal{O}(n+m)$. We are going to show the following: *The partial representation is extendible if and only if the reordering succeeds.* Moreover, we can use the reordered PQ-tree to construct a representation $\mathcal{R}$ extending the partial representation $\mathcal{R}'$.

To construct a representation, we place clique-points on the real line according to the ordering. We need to be more careful in this step. Since several intervals are pre-drawn, we cannot change their representations. Using the clique-points, we construct a representation in a similar manner as in Figure 2.5.

Now, we describe everything in detail.

Interval Ordering $\lhd$. For a clique a, let $I(a)$ denote the set of all the pre-drawn intervals that are contained in a. The pre-drawn intervals split the line into several *parts*, traversed by the same intervals. A clique-point $\mathrm{cp}(a)$ can be placed only to a part containing exactly the intervals of $I(a)$ and no other pre-drawn intervals.

We denote by $\curvearrowleft(a)$ (resp. $\curvearrowright(a)$) the leftmost (resp. the rightmost) point where the clique-point $\mathrm{cp}(a)$ can be placed, formally:

$$\begin{aligned}\curvearrowleft(a) &= \inf \bigl\{x \mid \text{the clique-point } \mathrm{cp}(a) \text{ can be placed to } x\bigr\},\\ \curvearrowright(a) &= \sup \bigl\{x \mid \text{the clique-point } \mathrm{cp}(a) \text{ can be placed to } x\bigr\}.\end{aligned}$$

For an example, see Figure 2.6. Notice that it does not mean that the clique-point $\mathrm{cp}(a)$ can be placed to all the points between $\curvearrowleft(a)$ and $\curvearrowright(a)$. If a clique-point can not be placed at all, the given partial representation is not extendible.

For two cliques a and b, we define $a \lhd b$ if $\curvearrowright(a) \le \curvearrowleft(b)$. It is quite natural since $a \lhd b$ implies that every correct representation has to place $\mathrm{cp}(a)$ to the left of $\mathrm{cp}(b)$. For example, the cliques a and b in Figure 2.6 satisfy $a \lhd b$.

Lemma 2.8. *The relation $\lhd$ is an interval order.*

Proof. The intervals of the interval order $\lhd$ correspond to the cliques of G. To a clique a, we assign an open interval $I_a = (\curvearrowleft(a), \curvearrowright(a))$. The definition of $\lhd$ exactly states that $a \lhd b$ if and only if the intervals I_a and I_b are disjoint and I_a is on the left of I_b. $\square$

If the input partial representation is either sorted or normalized, we can construct a normalized representation of this interval order in time $\mathcal{O}(n)$. Also notice that to solve the problem RepExt(INT), we only care about topology of the partial representation, the exact positions of the endpoints are not important.

The Algorithm. We proceed in the following five steps. Only the first three steps are necessary, if we just want to answer the decision problem without constructing a representation.

1. Find maximal cliques and construct a PQ-tree, independently of the partial representation.
2. Compute $\curvearrowleft$ and $\curvearrowright$ for all the cliques and obtain a normalized interval representation of an interval order $\lhd$.
3. Reorder the PQ-tree according to the interval order $\lhd$, using Section 2.1.2.
4. Place the clique-points greedily on the real line, according to the ordering.
5. Construct a representation using the clique-points.

Step 1 is the original recognition algorithm. In Step 2, we normalize the partial representation, compute splitting of the real line into parts and compute $\lhd$. In Step 3,

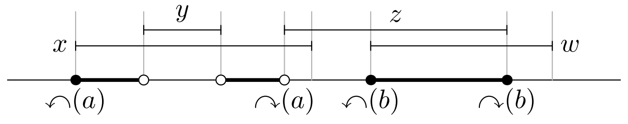

Figure 2.6: Clique-points $\mathrm{cp}(a)$ and $\mathrm{cp}(b)$, having $I(a) = \{x\}$ and $I(b) = \{z, w\}$, can be placed to the bold parts of the real lines.

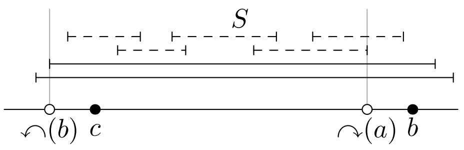

Figure 2.7: An illustration of the proof: The positions of the clique-points b and c, the intervals of S are dashed.

we apply the algorithm described in Section 2.1.2. In the rest of the chapter, we describe in detail steps four to five and prove the correctness of the algorithm.

Step 4: Placing the Clique-Points. We have an ordering $<$ of the clique-points from Step 3. The real line has several intervals already pre-drawn by the partial representation. We place clique-points greedily from left to right, according to the ordering.

Suppose we want to place a clique-point $\mathrm{cp}(a)$. Let $\mathrm{cp}(b)$ be the last placed clique-point. Consider the infimum over all the points where the clique-point $\mathrm{cp}(a)$ can be placed and that are to the right of the clique-point $\mathrm{cp}(b)$. If there is a single such point on the right of $\mathrm{cp}(b)$ (equal to the infimum), we place $\mathrm{cp}(a)$ there. Otherwise $\curvearrowleft(a) < \curvearrowright(a)$ and we place the clique-point $\mathrm{cp}(a)$ to the right of this infimum by an appropriate epsilon, for example the length of the shortest part (see definition of $\lhd$) divided by n. We can easily implement this greedy subroutine in time $\mathcal{O}(n)$.

The following lemma states that this greedy procedure cannot fail.

Lemma 2.9. *For an ordering $<$ of the cliques compatible with the PQ-tree extending $\lhd$, the greedy subroutine described in Step 4 never fails.*

Proof. We prove the lemma by contradiction. See Figure 2.7. Let $\mathrm{cp}(a)$ be the clique-point for which the procedure fails. Since $\mathrm{cp}(a)$ cannot be placed, there are some clique-points placed on the right of $\curvearrowright(a)$ (or possibly on $\curvearrowright(a)$ directly). Let $\mathrm{cp}(b)$ be the leftmost one of them. If $\curvearrowleft(b) \ge \curvearrowright(a)$, we obtain $a \lhd b$ which contradicts $b < a$ since $\mathrm{cp}(b)$ was placed before $\mathrm{cp}(a)$. So, we know that $\curvearrowleft(b) < \curvearrowright(a)$. To get a contradiction, we question why the clique-point $\mathrm{cp}(b)$ was not placed on the left of $\curvearrowright(a)$.

The clique-point $\mathrm{cp}(b)$ was not placed before $\curvearrowright(a)$ because all these positions were either blocked by some other previously placed clique-points, or they are traversed by some pre-drawn interval not in $I(b)$. There is at least one clique-point placed to the right of $\curvearrowleft(b)$ (otherwise we could place $\mathrm{cp}(b)$ to $\curvearrowleft(b)$ or right next to it). Let $\mathrm{cp}(c)$ be the right-most clique-point placed between $\curvearrowleft(b)$ and $\mathrm{cp}(b)$. Every point between $\mathrm{cp}(c)$ and $\curvearrowright(a)$ has to be covered by a pre-drawn interval not in $I(b)$. Consider the set S of all the pre-drawn intervals not contained in $I(b)$ intersecting $[c, \curvearrowright(a)]$; depicted dashed in Figure 2.7.

Let C be a set of all the cliques containing at least one vertex from S. Since S induces a connected subgraph, all the cliques of C appear consecutively in the ordering of the cliques since every pair of adjacent vertices is contained in a maximal clique.

Now, a and c both belong to C, but b does not. We assumed that $c < b < a$. Since $c < b$ and the consecutivity of C, $a < b$ which contradicts $b < a$. $\square$

Step 5: Constructing a Representation. We construct a representation of the graph using the clique-points placed in the previous step, similarly to Figure 2.5. We represent each vertex as an interval containing exactly all the clique-points corresponding to the cliques containing this vertex.

The intervals placed by the partial representation contain the correct clique-points. Since the ordering of the clique-points is compatible with the PQ-tree, we obtain a correct representation.

The Main Proof. Now we are ready to prove one main result of this thesis, Theorem 1.1 which states the problem $\textsc{RepExt}(\mathsf{INT})$ can be solved in time $\mathcal{O}(n+m)$:

Proof of Theorem 1.1. The conditions given by the interval order $\lhd$ on the order of the clique-points are clearly necessary. If it is not possible to reorder the PQ-tree according to $\lhd$, the structure of the interval graph is inconsistent with the partial representation and $\textsc{RepExt}(\mathsf{INT})$ is not solvable. On the other hand, if the PQ-tree can be reordered according to $\lhd$, Lemma 2.9 states that the partial representation is extendible.

The time complexity of the algorithm is clearly $\mathcal{O}(n+m)$. The number of cliques is at most $\mathcal{O}(n+m)$ and the PQ-tree can be constructed in this time. According to Proposition 2.4, the PQ-tree can be reordered according to $\lhd$ in time $\mathcal{O}(n+m)$. Finally, the representation can be constructed in time $\mathcal{O}(n+m)$. $\square$

3 Extending Proper and Unit Interval Graphs

In this chapter, we extend representations of proper and unit interval graphs. As was already described in the introduction, even through these classes are known to be equal, they behave differently for the partial representation extension problem.

In Section 3.1, we deal with proper interval graphs and the extension algorithm is based on a simple characterization of extendible instances. In Section 3.2, we study a more general problem of bounded representations of unit interval graphs and we show that this problem is NP-complete. In Section 3.3, we deal with specific instances of the bounded representation problem for which the left-to-right order of the components is prescribed, and give an almost quadratic-time combinatorial algorithm for this problem. In Section 3.4, we solve the RepExt(UNIT INT) problem by showing that it can be expressed as an instance of the bounded representation problem with prescribed ordering.

3.1 Extending Proper Interval Graphs

In this section, we describe how to extend partial representations of proper interval graphs in time $\mathcal{O}(m+n)$. We also give a simple characterization of all extendible instances.

Left-to-right ordering. Roberts [38] gave the following characterization of proper interval graphs:

Lemma 3.1 (Roberts). *A graph is a proper interval graph if and only if there exists a linear ordering $v_1 \lhd v_2 \lhd \cdots \lhd v_n$ of its vertices such that the closed neighborhood of every vertex is consecutive.*

This linear order $\lhd$ is the left-to-right order of the intervals on the real line in some representation. In each representation, the order of the left endpoints is exactly the same as the order of the right endpoints, and this order is $\lhd$. This makes recognition much simpler. For example of $\lhd$, see Figure 3.1.

How many different orderings $\lhd$ may a proper interval graph admit? Possibly many but all of them have a very simple structure. Vertices u and v are called *indistinguishable* if $N[u] = N[v]$. The vertices of G can be partitioned into *groups* of (pairwise)

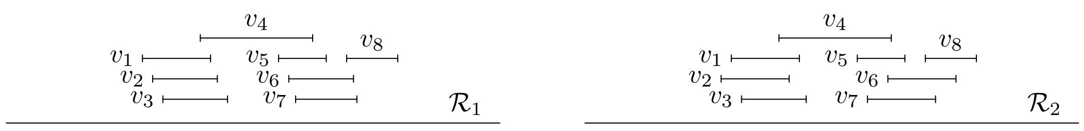

Figure 3.1: Two proper interval representations $\mathcal{R}_1$ and $\mathcal{R}_2$ with the left-to-right orderings $v_1 \lhd v_2 \lhd v_3 \lhd v_4 \lhd v_5 \lhd v_6 \lhd v_7 \lhd v_8$ and $v_2 \lhd v_1 \lhd v_3 \lhd v_4 \lhd v_5 \lhd v_7 \lhd v_6 \lhd v_8$.

indistinguishable vertices. Note that indistinguishable vertices may be represented by the same intervals (and this is actually true for general intersection representations). In Figure 3.1, the graph contains two groups $\{v_1, v_2, v_3\}$ and $\{v_6, v_7\}$. The vertices of each group appear consecutively in the ordering $\lhd$ and may be reordered arbitrarily. Deng et al. [10] proved the following:

Lemma 3.2 (Deng et al.). *For a connected proper interval graph, the ordering $\lhd$ satisfying the condition of Lemma 3.1 is uniquely determined up to local reordering of groups and complete reversal.*

This lemma is key for partial representation extension of proper interval graphs. Essentially, we just have to deal with a unique ordering (and its reversal) and match the partial representation on it.

We want to construct a partial ordering $<$ which is a simple representation of all orderings $\lhd$ from Lemma 3.1. There exists a proper interval representation with an ordering $\lhd$ if and only if $\lhd$ extends either $<$ or its reversal. According to Lemma 3.2, $<$ can be constructed by taking an arbitrary ordering $\lhd$ and making indistinguishable vertices incomparable. For graph in Figure 3.1, we get

$$(v_1, v_2, v_3) < v_4 < v_5 < (v_6, v_7) < v_8,$$

where groups of indistinguishable vertices are put in brackets. This ordering is unique up to reversal and can be constructed in time $\mathcal{O}(n + m)$ [8].

Characterization of Extendible Instances. We give a simple characterization of the partial representation instances that are extendible. We start with connected instances. Let G be a proper interval graph and $\mathcal{R}'$ be a partial representation of its induced subgraph G'. Then intervals in $\mathcal{R}'$ are in some left-to-right ordering $<^{G'}$, and if two intervals are represented the same, they are incomparable in $<^{G'}$.

Lemma 3.3. *The partial representation $\mathcal{R}'$ of a connected graph G is extendible if and only if there exists a linear ordering $\lhd$ of $V(G)$ extending $<^{G'}$, and either $<$ or its reversal.*

Proof. If there exists a representation $\mathcal{R}$ extending $\mathcal{R}'$, then it is in some left-to-right ordering $\lhd$. Clearly, the pre-drawn intervals are placed the same so $\lhd$ has to extend $<^{G'}$. According to Lemma 3.2, $\lhd$ extends $<$ or its reversal.

Conversely, let $v_1 \lhd \cdots \lhd v_n$ be an ordering from the statement of the lemma. We construct a representation $\mathcal{R}$ extending $\mathcal{R}'$ as follows. We can assume that the graph G does not contain any pair u and v of indistinguishable vertices such that u

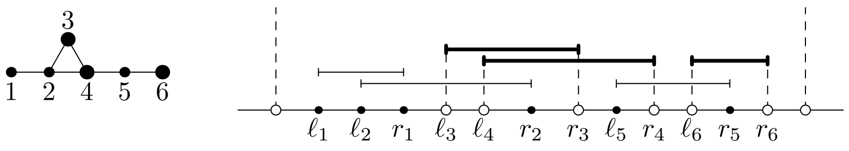
3
1 2 4 5 6
ℓ_1 ℓ_2 r_1 ℓ_3 ℓ_4 r_2 r_3 ℓ_5 r_4 ℓ_6 r_5 r_6

Figure 3.2: Representation of a component with order $1 \lhd 2 \lhd 3 \lhd 4 \lhd 5 \lhd 6$. First, we compute the common order of the left and the right endpoints: $\ell_1 \lessdot \ell_2 \lessdot r_1 \lessdot \ell_3 \lessdot \ell_4 \lessdot r_2 \lessdot r_3 \lessdot \ell_5 \lessdot r_4 \lessdot \ell_6 \lessdot r_5 \lessdot r_6$. The endpoints of the pre-drawn intervals split the segment into several subsegments. We place the remaining endpoints in this order and, within every subsegment, distributed equidistantly.

is not pre-drawn; otherwise we remove u from the graph and later put $R_u = R_v$. We compute a common linear ordering $\lessdot$ of left and right endpoints from left-to-right. With start with the ordering $\ell_1 \lessdot \cdots \lessdot \ell_n$, into which we insert the right endpoints $r_1, \dots, r_n$ one-by-one. For v_i, let v_j be the right-most neighbor in the ordering $\lhd$. Then, we place r_i right before ℓ_{j+1} (if it exists, otherwise we append it to the end of the ordering).

This left-to-right common order $\lessdot$ is uniquely determined by $\lhd$. Since $\lhd$ extends $<^{G'}$, it is compatible with the partial representation (the pre-drawn endpoints are ordered as in $\lessdot$). To construct the representation, we just place the non-pre-drawn endpoints equidistantly into the gaps between neighboring pre-drawn endpoints (or to the left or right of $\mathcal{R}'$); see Figure 3.2 for an example.

We argue correctness of the constructed representation $\mathcal{R}$. It extends $\mathcal{R}'$ since the pre-drawn intervals are the same. Second, it is a correct interval representation. Let u and v be two vertices with $v_i \lhd v_j$ and let v_k be the right-most neighbor of v_i in $\lhd$. If $v_iv_j \in E(G)$, then $\ell_i \lessdot \ell_k \lessdot r_i$ and by consecutivity of $N[u]$ in $\lhd$ is $\ell_j \lessdot \ell_k$. If $uv \notin E(G)$, then $r_i \lessdot \ell_{k+1} \lessdot \ell_j$. In both cases, R_u and R_v intersect correctly. Last, we argue that $\mathcal{R}$ is a proper interval representation. In $\lessdot$ the order of the left endpoints is the same as the order of the right-endpoints since r_{i+1} is always placed on the right of r_i in $\lessdot$.

We conclude that the representation $\mathcal{R}$ can be made small enough to fit into any open segment of the real line that contains all pre-drawn intervals. $\square$

Now, we are ready to characterize general solvable instances.

Lemma 3.4. *The partial representation $\mathcal{R}'$ of a graph G is extendible if and only if*

1. *for each component C the partial representation $\mathcal{R}'_C$ consisting of pre-drawn intervals in C is extendible, and*
2. *pre-drawn vertices of each component are consecutive in $<^{G'}$.*

Proof. The necessity of (1) is due to Lemma 3.3 applied on each component C. For (2), if some component C would not have its pre-drawn vertices consecutive in $<^{G'}$, then the area $\bigcup_{u \in C} R_u$ would not be a connected segment of the real line (contradicting existence of $\blacktriangleleft$ from Section 1.5).

Now, if the instance satisfies the both conditions, we can construct a correct representation $\mathcal{R}$ extending $\mathcal{R}'$ as follows. Using (2), the located components are

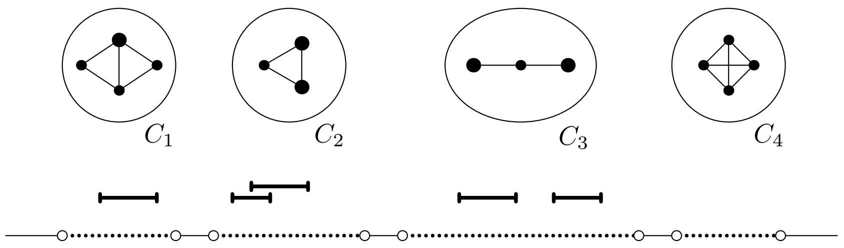

Figure 3.3: An example of a graph with four components $C_1, \ldots, C_4$. The pre-drawn intervals give the order of the located components $C_1 \blacktriangleleft C_2 \blacktriangleleft C_3$. The non-located component C_4 is placed to the right. For each component, we reserve some segment in which we construct the representation.

ordered from left to right and we assign pairwise disjoint *open segments* containing all their pre-drawn intervals (there is a non-empty gap between located components we can use). To unlocated components we assign pair-wise disjoint open segments to the right of the right-most located component. See Figure 3.3. For each component, we construct a representation in its open segment, using the construction in the proof of Lemma 3.3. ☐

Now we are ready to prove that RepExt(PROPER INT) can be solved in time $\mathcal{O}(n+m)$:

Proof of Theorem 1.2. We just use the characterization by Lemma 3.4. The conditions (1) and (2) can be easily checked in time $\mathcal{O}(n+m)$. If necessary, a representation $\mathcal{R}$ can be constructed in the same running time since the proofs of Lemmas 3.3 and 3.4 are constructive. ☐

For a pseudocode, see Algorithm 4 in Appendix A.4.

3.2 Bounded Representations of Unit Interval Graphs

In this section we deal with bounded representations. An input of BoundRep consists of a graph G and, for each vertex v_i, a *lower bound* $\text{lbound}(v_i)$ and an *upper bound* $\text{ubound}(v_i)$. (We allow $\text{lbound}(v_i) = -\infty$ and $\text{ubound}(v_i) = +\infty$.) The problem asks whether there exists a unit interval representation $\mathcal{R}$ of G such that $\text{lbound}(v_i) \le \ell_i \le \text{ubound}(v_i)$ for each interval v_i. Such a representation is called a *bounded representation.*

Since unit interval representations are proper interval representations, all properties of proper interval representations described in Section 3.1 carry over, in particular properties of orderings $\lhd$ and $<$.

3.2.1 Representations in ε-grids

Endpoints of intervals can be positioned at arbitrary real numbers. For the purpose of the algorithm, we want to work with drawings of limited resolutions. For a given instance of the bounded representation problem, we want to find a lower bound for the required resolution such that this instance is solvable if and only if it is solvable in this limited resolution.

More precisely, we want to represent all intervals so that their endpoints correspond to points on some grid. For a value $\varepsilon = \frac{1}{K} > 0$ where K is an integer, the ε-grid is the set of points $\{k\varepsilon : k \in \mathbb{Z}\}$.[1] For a given instance of BoundRep, we ask which value of ε ensures that we can construct a representation having all endpoints on the ε-grid. So the value of ε is the resolution of the drawing.

For the standard unit interval graph representation problem a grid of size $\frac{1}{n}$ is sufficient [8]. In the case of BoundRep, the size of the grid has to depend on the values of the bounds. Consider all values $\text{lbound}(v_i)$ and $\text{ubound}(v_i)$ distinct from $-\infty, +\infty$, and express them as irreducible fractions $\frac{p_1}{q_1}, \frac{p_2}{q_2}, \cdots, \frac{p_b}{q_b}$. Then we define:

$$\varepsilon' := \frac{1}{\text{lcm}(q_1, q_2, \dots, q_b)}, \qquad \text{and} \qquad \varepsilon := \frac{\varepsilon'}{n}. \tag{3.1}$$

We show that an ε-grid is sufficient to construct the bounded representation:

Lemma 3.5. *If there exists a valid representation $\mathcal{R}'$ for an input of the problem* BoundRep*, there exists a valid representation $\mathcal{R}$ in which all intervals have endpoints on the ε-grid where ε is defined by (3.1).*

Proof. We construct an ε-grid representation $\mathcal{R}$ from $\mathcal{R}'$ in two steps. First, we shift intervals to the left, and then we shift intervals slightly back to the right. For every interval v_i, the sizes of the left and right shifts are denoted by $\text{LS}(v_i)$ and $\text{RS}(v_i)$ respectively. The shifting process is shown in Figure 3.4.

In the first step, we consider the ε'-grid and shift all the intervals to the left to the closest grid-point (we do not shift an interval if its endpoints are already on the grid). Original intersections are kept by this shifting since if x and y are two endpoints having $x \le y$ before the left-shift, then also $x \le y$ after the left-shift. So if $v_iv_j \in E$ and $\ell_i \le \ell_j \le r_i$, then these inequalities are preserved by the shifting. On the other hand, we may introduce additional intersections by shifting two non-intersecting intervals to each other. In this case, after the shift the intervals only touch; for an example, see vertices v_2 and v_4 in Figure 3.4.

The second step shifts the intervals to the right in the refined ε-grid to remove the additional intersections created by the first step. The right-shift is a mapping

$$\text{RS} : \{v_1, \dots, v_n\} \to \{0, \varepsilon, 2\varepsilon, \dots, (n-1)\varepsilon\}$$

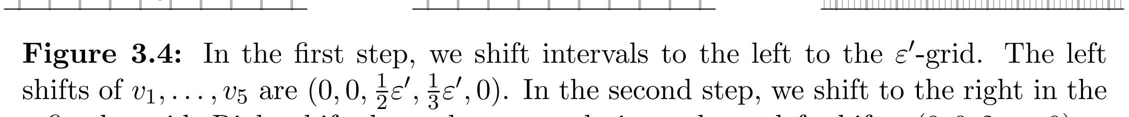

Figure 3.4: In the first step, we shift intervals to the left to the ε'-grid. The left shifts of $v_1, \dots, v_5$ are $(0, 0, \frac{1}{2}\varepsilon', \frac{1}{3}\varepsilon', 0)$. In the second step, we shift to the right in the refined ε-grid. Right shifts have the same relative order as left shifts: $(0, 0, 2\varepsilon, \varepsilon, 0)$.

[1] If ε would not be of a form $\frac{1}{K}$, then the grid could not contain both left and right endpoints of the intervals. We reserve K for the value $\frac{1}{\varepsilon}$ in this chapter.

having the *right-shift property*: For all pairs (v_i, v_j) with $r_i = \ell_j$, then $\mathrm{RS}(v_i) \geq \mathrm{RS}(v_j)$ if and only if $v_i v_j \in E$.

To construct such a mapping RS, notice that if we relaxed the image of RS to $[0, \varepsilon')$, the reversal of LS would have the right-shift property, since it produces the original correct representation $\mathcal{R}'$. But the right-shift property depends only on the relative order of the shifts and not on the precise values. Therefore, we can construct RS from the reversal of LS by keeping the shifts in the same relative order. If $\mathrm{LS}(v_i)$ is one of the kth smallest shifts, we set $\mathrm{RS}(v_i) = (k-1)\varepsilon$.[2] See Figure 3.4.

We finally argue that these shifts produce a correct ε-grid representation. The right-shift does not create additional intersections: After LS non-intersecting pairs are at distance at least $\varepsilon' = n\varepsilon$, and by RS they can get closer by at most $(n-1)\varepsilon$. Also, if after LS two intervals overlap by at least ε', their intersection is not removed by RS. The only intersections which are modified by RS are touching pairs of intervals (v_i, v_j) having $r_i = \ell_j$ after LS. The mapping RS shifts these pairs correctly according to the edges of the graph.

Next we look at the bound constraints. If, before the shifting, v_i was satisfying $\ell_i \geq \mathrm{lbound}(v_i)$, then this is also satisfied after $\mathrm{LS}(v_i)$ since the ε'-grid contains the value $\mathrm{lbound}(v_i)$. Obviously, the inequality is not broken after $\mathrm{RS}(v_i)$. As for the upper bound, if $\mathrm{LS}(v_i) = 0$ and $\mathrm{RS}(v_i) = 0$, then the bound is trivially satisfied. Otherwise, after $\mathrm{LS}(v_i)$ we have $\ell_i \leq \mathrm{ubound}(v_i) - \varepsilon'$, so the upper bound still holds after $\mathrm{RS}(v_i)$. $\square$

Additionally, Lemma 3.5 shows that it is always possible to construct an ε-grid representation having the same topology as the original representation, in the sense that overlapping pairs of intervals keep overlapping, and touching pairs of intervals keep touching. Also notice that both representations $\mathcal{R}$ and $\mathcal{R}'$ have the same order of the intervals.

In the standard unit interval graph representation problem, no bounds on the positions of the intervals are given, and we get $\varepsilon' = 1$ and $\varepsilon = \frac{1}{n}$. Lemma 3.5 proves in a particularly clean way that the grid of size $\frac{1}{n}$ is sufficient to construct unrestricted representations of unit interval graphs. Corneil et al. [8] show how to construct this representation directly from the ordering $<$, whereas we use some given representation to construct an ε-grid representation.

3.2.2 The Hardness of BoundRep

In this subsection we focus on hardness of bounded representations of unit interval graphs. We prove Theorem 1.3 stating that BoundRep is NP-complete.

We reduce the problem from 3-Partition. An input of 3-Partition consists of natural numbers k, M, and $A_1, \ldots, A_{3k}$ such that $\frac{M}{4} < A_i < \frac{M}{2}$ for all i, and $\sum A_i = kM$. The question is whether it is possible to partition the numbers A_i into k triples such that each triple sums to exactly M. This problem is known to be strongly NP-complete (even if all numbers have polynomial sizes) [14].

[2]In other words, for the smallest shifts we assign the right-shift 0, for the second smallest shifts we assign ε, for the third smallest shifts 2ε, and so on.

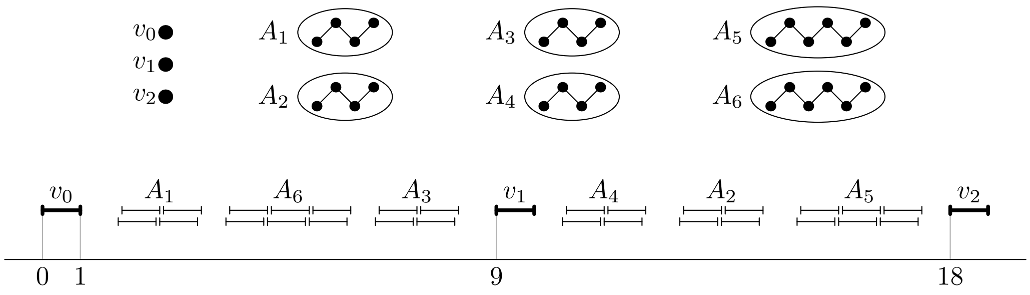

Figure 3.5: We consider the following input for 3-Partition: $k = 2$, $M = 7$, $A_1 = A_2 = A_3 = A_4 = 2$ and $A_5 = A_6 = 3$. The associated unit interval graph is depicted on top, and at the bottom we find one of its correct bounded representations, giving 3-partitioning $\{A_1, A_3, A_6\}$ and $\{A_2, A_4, A_5\}$.

Proof of Theorem 1.3. According to Lemma 3.5, if there exists a representation satisfying the bound constraints, there also exists an ε-grid representation with this property. Since the length of ε given by (3.1), written in binary, is polynomial in the size of the input, all endpoints can be placed in polynomially-long positions. Thus we can guess the bounded representation and the problem belongs to NP.

Let us next prove that the problem is NP-hard. For a given input of 3-Partition, we construct the following unit interval graph G. For each number A_i, we add a path P_{2A_i} (of length $2A_i - 1$) into G as a separate component. For all vertices x in these paths, we set bounds

$$\text{lbound}(x) = 1 \qquad \text{and} \qquad \text{ubound}(x) = k \cdot (M + 2).$$

In addition, we add $k + 1$ independent vertices $v_0, v_1, \ldots, v_k$ and make their positions in the representation fixed:

$$\text{lbound}(v_i) = \text{ubound}(v_i) = i \cdot (M + 2).$$

See Figure 3.5 for an illustration of the reduction. Clearly, the reduction is polynomial.

We now argue that the bounded representation problem is solvable if and only if the given input of 3-Partition is solvable. Suppose first that the bounded representation problem admits a solution. There are k gaps between the fixed intervals $v_0, \ldots, v_k$ each of which has space less than $M + 1$. (The length of the gap is $M + 1$ but the endpoints are taken by v_i and v_{i+1}.) The bounds of the paths force their representations to be inside these gaps, and each path lives in exactly one gap. Hence the representation induces a partition of the paths.

Now, the path P_{2A_i} needs space at least A_i in every representation. The representations of the paths may not overlap and the space in each gap is less than $M + 1$, hence the sum of all A_i's in each part is at most M. Since the total sum of A_i's is exactly kM, the sum in each part has to be M. Thus the obtained partition solves the 3-Partition problem.

Conversely, every solution of 3-Partition can be realized in this way. $\square$

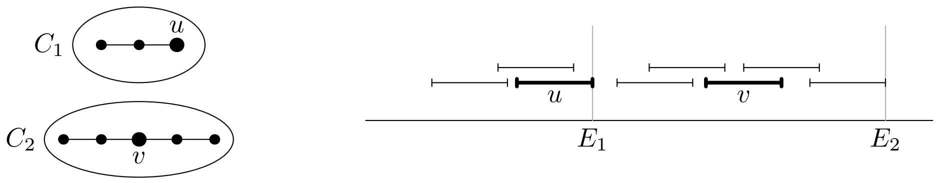

Figure 3.6: The positions of the vertices u and v are fixed by the bound constraints. The component C_1 can only be represented in a single way, since its reversal would block space for the component C_2.

3.3 Bounded Representations with Prescribed Ordering

In this section, we deal with the BoundRep problem when a fixed ordering $\blacktriangleleft$ of the components is prescribed. First we solve the problem using linear programming, then we describe additional structure of bounded representations and using this structure we construct an almost quadratic-time algorithm solving the linear programs.

3.3.1 LP Approach for BoundRep

According to Lemma 3.2, each component of G can be represented in at most two different ways, up to local reordering of groups of indistinguishable vertices. Unlike the case of proper interval graphs, we cannot arbitrarily choose one of the orderings, since neighboring components restrict each other's space. For example, only one of the two orderings for the component C_1 in Figure 3.6 makes a representation of C_2 possible.

In the algorithm, we process components $C_1 \blacktriangleleft C_2 \blacktriangleleft \cdots \blacktriangleleft C_c$ from left to right. For each component C_t, we calculate by the algorithm of Corneil et al. the partial ordering $<$ described in Section 3.1 and its reversal. The elements that are incomparable by these partial orderings are vertices of the same group of indistinguishable vertices. From $<$, we want to construct a correct linear orderings $\lhd$.

Lemma 3.6. *Suppose there exists some bounded representation $\mathcal{R}$. Then there exists a bounded representation $\mathcal{R}'$ such that for every indistinguishable pair v_i and v_j having* $\mathrm{lbound}(v_i) \le \mathrm{lbound}(v_j)$ *it holds that $\ell'_i \le \ell'_j$.*

Proof. For a representation, we call a pair (v_i, v_j) *bad* if v_i and v_j are indistinguishable, $\mathrm{lbound}(v_i) \le \mathrm{lbound}(v_j)$ and $\ell_i > \ell_j$. We describe a process which iteratively constructs $\mathcal{R}'$ from $\mathcal{R}$, by constructing a sequence of representations $\mathcal{R} = \mathcal{R}_0, \mathcal{R}_1, \dots, \mathcal{R}_k = \mathcal{R}'$, where the positions in a representation $\mathcal{R}_s$ are denoted by ℓ_i^s's.

In each step s, we create $\mathcal{R}_s$ from $\mathcal{R}_{s-1}$ by fixing one bad pair (v_i, v_j): we set $\ell_i^s = \ell_j^{s-1}$ and the rest of the representation remains the same. Since v_i and v_j are indistinguishable and $\mathcal{R}_{s-1}$ is correct, the obtained $\mathcal{R}_s$ is a representation. Regarding bound constraints,

$$\mathrm{lbound}(v_i) \le \mathrm{lbound}(v_j) \le \ell_j^{s-1} = \ell_i^s < \ell_i^{s-1} \le \mathrm{ubound}(v_i),$$

so the bounds of v_i are satisfied.

Now, in each $\mathcal{R}_s$ the set of all left endpoints is a subset of the set of all left endpoints of $\mathcal{R}$. In each step, we move one left-endpoint to the left, so each endpoint is moved at most $n-1$ times. Hence the process terminates after $\mathcal{O}(n^2)$ iterations and produces $\mathcal{R}'$ without bad pairs are requested. □

For $<$ and its reversal, we use Lemma 3.6 to construct linear orderings $\lhd$: If v_i and v_j belong to the same group of indistinguishable vertices and $\mathrm{lbound}(v_i) < \mathrm{lbound}(v_j)$, then $v_i \lhd v_j$. If $\mathrm{lbound}(v_i) = \mathrm{lbound}(v_j)$, we choose any order $\lhd$ between v_i and v_j.

We obtain two total orderings $\lhd$, and we solve a linear program for each of them. Let $v_1 \lhd v_2 \lhd \cdots \lhd v_k$ be one of these orderings. We denote the right-most endpoint of a representation of a component C_t by E_t. Additionally, we define $E_0 = -\infty$. The linear program has variables $\ell_1, \ldots, \ell_k$, and it minimizes the value of E_t. Let ε be defined as in (3.1). We solve:

$$\begin{array}{llll}
\text{Minimize:} & E_t := \ell_k + 1, & & \\
\text{subject to:} & E_{t-1} + \varepsilon \le \ell_1, & & (3.2)\\
& \ell_i \le \ell_{i+1}, & \forall i = 1, \ldots, k-1, & (3.3)\\
& \ell_i \ge \mathrm{lbound}(v_i), & \forall i = 1, \ldots, k, & (3.4)\\
& \ell_i \le \mathrm{ubound}(v_i), & \forall i = 1, \ldots, k, & (3.5)\\
& \ell_i \ge \ell_j - 1, & \forall v_iv_j \in E, v_i \lhd v_j, & (3.6)\\
& \ell_i + \varepsilon \le \ell_j - 1, & \forall v_iv_j \notin E, v_i \lhd v_j. & (3.7)
\end{array}$$

We solve the same linear program for the other ordering of the vertices of C_t. If none of the two programs is feasible, we report that no bounded representation exists. If exactly one of them is feasible, we keep the values obtained for $\ell_1, \ldots, \ell_k$ and E_t, and process the next component C_{t+1}. If the two problems are feasible, we keep the one such that the value of E_t in the solution is smaller, and process C_{t+1}.

Lemma 3.7. *Every bounded ε-grid representation of every component C_t with the left-to-right order $v_1 \lhd \cdots \lhd v_k$ satisfies all constraints (3.4)–(3.7).*

Proof. Constraints of types (3.4) and (3.5) are satisfied since the representation is bounded. Constraints of type (3.6) give correct representation of intersecting pairs of intervals. The non-intersecting pairs of an ε-grid representation are at distance at least ε which makes constraints of type (3.7) satisfied. □

Now, we are ready to show:

Proposition 3.8. *The* BOUNDREP *problem can be solved in polynomial time with respect to r by the algorithm above where r is the size of the input.*

Proof. Concerning the running time, it depends polynomially on the sizes of n and ε, which are polynomial in the size of the input r. It remains to show correctness.

Suppose that the algorithm returns a candidate for a bounded representation. The formulation of the linear program ensures that it is a correct representation:

Constraints of type (3.2) and (3.3) make the representation respect $\lhd$, and the drawings of the distinct components are disjoint. Constraints of type (3.4) and (3.5) enforce that the given lower and upper bounds for the positions of the intervals are satisfied. Finally, constraints of type (3.3), (3.6) and (3.7) make the drawing of the vertices of a particular component C_t be a correct representation.

Suppose next that a bounded representation exists. According to Lemma 3.5 and Lemma 3.6, there also exists an ε-grid bounded representation $\mathcal{R}'$ having the order in the indistinguishable groups as defined above. So for each component C_t, one of the two orderings $\lhd$ constructed for the linear programs agree with the left-to-right order of C_t in $\mathcal{R}'$.

We want to show that the representation of each component C_t in $\mathcal{R}'$ gives a solution to one of the two linear programs associated to C_t. We denote by E'_t the value of E_t in the representation $\mathcal{R}'$, and by $E^{\min}_t$ the value of E_t obtained by the algorithm after solving the two linear programming problems associated to C_t. We show by induction on t that $E^{\min}_t \le E'_t$, which specifically implies that $E^{\min}_t$ exists and at least one of the linear programs for C_t is solvable.

We start with C_1. As argued above, the left-to-right order in $\mathcal{R}'$ agrees with one of the orderings $\lhd$, so the representation of C_1 satisfies the constraints (3.3). Since $E_0 = -\infty$, it also satisfies the constraint (3.2). By Lemma 3.7, the rest of the constraints are also satisfied. Thus the representation of C_1 gives a feasible solution for one program and gives $E^{\min}_1 \le E'_1$.

Assume now that, for some C_t with $t \ge 1$, at least one of the two linear programming problems associated to C_t admits a solution, and from induction hypothesis $E^{\min}_t \le E'_t$. In $\mathcal{R}'$, two neighboring components are represented at distance at least ε. Additionally, if v_1 refers to a vertex of C_{t+1}, $\ell_1 \ge E'_t + \varepsilon \ge E^{\min}_t + \varepsilon$. Therefore, representation of C_{t+1} in $\mathcal{R}'$ satisfies the constraint (3.2) and similarly as above the rest of the conditions. So the representation of C_{t+1} in $\mathcal{R}$ gives some solution to one of the programs and we get $E^{\min}_{t+1} \le E'_{t+1}$.

In summary, if there exists a bounded representation, for each component C_t at least one of the two linear programming problems associated to C_t admits a solution. Therefore, the algorithm returns a correct bounded representation $\mathcal{R}$ (as discussed in the beginning of the proof). We note that $\mathcal{R}$ does not have to be an ε-grid representation since the linear program just states that non-intersecting intervals are at distance at least ε. To construct an ε-grid representation if necessary, we can proceed as in the proof of Lemma 3.5. $\square$

For a pseudocode, see Algorithm 5 in Appendix A.5.

We note that it is possible to reduce the number of constraints of the linear program from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Using the ordering constraints (3.3), we can replace the groups of constraints (3.6) and (3.7) by a linear number of constraints as follows. For each v_j, let v_i be the rightmost vertex such that $v_i \lhd v_j$ and $v_iv_j \notin E$. Then we only state the constraint (3.6) for v_{i+1} and v_j, and the constraint (3.7) for v_i and v_j. This is equivalent to the original formulation of the problem.

3.3.2 The partially ordered set $\mathfrak{Rep}$

Let the graph G in consideration be a connected unit interval graph. We study structural properties of its representations. Suppose that we fix some partial left-to-right order $<$ of the intervals from Section 3.1, so that only indistinguishable vertices are incomparable, and also we fix some positive $\varepsilon = \frac{1}{K}$. For most of this section, we work just with lower bounds and completely ignore upper bounds.

We define $\mathfrak{Rep}$ as the set of all ε-grid representations in the left-to-right ordering extending $<$ satisfying the lower bounds. We define a very natural partial ordering $\leq$ on $\mathfrak{Rep}$: We put $\mathcal{R} \leq \mathcal{R}'$ if and only if $\ell_i \leq \ell'_i$ for every $v_i \in V(G)$. In this section, we study structural properties of the poset $(\mathfrak{Rep}, \leq)$.

If $\varepsilon \leq \frac{1}{n}$, it holds that $\mathfrak{Rep} \neq \emptyset$ since the graph G is a unit interval graph and there always exists an ε-grid representation $\mathcal{R}$ far to the right satisfying the lower bound contraints.

The Semilattice Structure. Let us assume that $\mathrm{lbound}(v_i) > -\infty$ for some $v_i \in V(G)$, otherwise the structure of $\mathfrak{Rep}$ is not very interesting. Let S be a subset of $\mathfrak{Rep}$. The infimum $\inf(S)$ is the greatest representation $\mathcal{R} \in \mathfrak{Rep}$ such that $\mathcal{R} \leq \mathcal{R}'$ for every $\mathcal{R}' \in S$. In a general poset, infimums may not exist, but if they exist, they are always unique. For $\mathfrak{Rep}$, we show:

Lemma 3.9. *For every non-empty $S \subseteq \mathfrak{Rep}$, the infimum $\inf(S)$ exists.*

Proof. We construct the requested infimum $\mathcal{R}$ as follows:

$$\ell_i = \min\{\ell'_i : \forall \mathcal{R}' \in S\}, \qquad \forall v_i \in V(G).$$

Notice that the positions in $\mathcal{R}$ are well-defined since the position of each interval in each $\mathcal{R}'$ is bounded and always on the ε-grid. Clearly, if $\mathcal{R}$ is a correct representation, it is the infimum $\inf(S)$. It remains to show that $\mathcal{R} \in \mathfrak{Rep}$.

Clearly all positions in $\mathcal{R}$ belong to the ε-grid and satisfy the lower bound constraints. Let v_i and v_j be two vertices. The values ℓ_i and ℓ_j in $\mathcal{R}$ are given by two representation $\mathcal{R}_1, \mathcal{R}_2 \in S$ (so $\ell_i = \ell^1_i$ and $\ell_j = \ell^2_j$). Notice that the left-to-right order in $\mathcal{R}$ has to extend $<$: If $v_i < v_j$, then $\ell_i = \ell^1_i \leq \ell^2_i < \ell^2_j = \ell_j$ since $\mathcal{R}_1$ minimizes the position of v_i and the left-to-right order in $\mathcal{R}_2$ extends $<$. Concerning correctness of the representation of the pair v_i and v_j, we suppose that $\ell_i = \ell^1_i \leq \ell^2_j = \ell_j$, otherwise we swap v_i and v_j.

- Let $v_iv_j \in E(G)$. Then $\ell^2_j \leq \ell^1_j$ since $\mathcal{R}_2$ minimized the position of v_j. Since $\mathcal{R}_1$ is a correct representation, $\ell^1_j - 1 \leq \ell^1_i$. So $\ell_j - 1 \leq \ell_i \leq \ell_j$, and the intervals v_1 and v_2 intersect.
- The other case is when $v_iv_j \notin E(G)$. Then $\ell^1_i \leq \ell^2_i \leq \ell^2_j - 1 - \varepsilon$ since $\mathcal{R}_1$ minimized the position of v_i, $\mathcal{R}_2$ is a correct representation and $v_i < v_j$ in both representations. So also v_i and v_j do not intersect in $\mathcal{R}$ as requested.

So $\mathcal{R}$ represents correctly each pair v_i and v_j, and hence $\mathcal{R} \in \mathfrak{Rep}$. □

A poset is a *(meet)-semilattice* if for every pair of elements a, b the infimum $\inf(\{a, b\})$ exists. Lemma 3.9 shows that the poset $(\mathfrak{Rep}, \leq)$ forms a (meet)-semilattice. Similarly as $\mathfrak{Rep}$, we could consider a poset set of all (ε-grid) representations satisfying both the lower and the upper bounds. The structure of this poset would be a *complete lattice* having infimums and supremums of all subsets. Lattices and semilattices are frequently studied and posets which are lattices satisfy very strong algebraic properties.

The Left-most Representation. We are interested in a specific representation in $\mathfrak{Rep}$, called the *left-most representation.* An ε-grid representation $\mathcal{R} \in \mathfrak{Rep}$ is the left-most representation if $\mathcal{R} \leq \mathcal{R}'$ for every $\mathcal{R}' \in \mathfrak{Rep}$; so the left-most representation is left-most in each interval at the same time. We note that the notion of the left-most representation does not make sense if we consider general representations (not on the ε-grid). The left-most representation is the infimum $\inf(\mathfrak{Rep})$, and thus by Lemma 3.9 we get:

Corollary 3.10. *The left-most representation always exits and it is unique.*

There are two algorithmic motivations for studying left-most representations. First, in the linear program of Section 3.3.1 we need to find a representation minimizing E_t. Clearly, the left-most representation is minimizing E_t and in addition it is minimizing the rest of the endpoints as well. The second motivation is that we want to construct a representation satisfying the upper bounds as well. It seems reasonable to try to place every interval as far to the left as possible and the left-most representation is a good candidate for a bounded representation.

Lemma 3.11. *There exists a representation $\mathcal{R}'$ satisfying both lower and upper bound constraints if and only if the left-most representation $\mathcal{R}$ satisfies the upper bound constraints.*

Proof. Since $\mathcal{R} \in \mathfrak{Rep}$, it satisfies the lower bounds. If $\mathcal{R}$ satisfies the upper bound constraints, it is a bounded representation. On the other hand, let $\mathcal{R}'$ be a bounded representation. Then

$$\mathrm{lbound}(v_i) \leq \ell_i \leq \ell'_i \leq \mathrm{ubound}(v_i), \qquad \forall v_i \in V(G),$$

and the left-most representation is also a bounded representation. □

3.3.3 Left-Shifting of Intervals

Suppose that we construct some initial ε-grid representation which is not the left-most representation. We want to transform this initial representation in $\mathfrak{Rep}$ into the left-most representation of $\mathfrak{Rep}$ by applying the following simple operation called *left-shifting.* The left-shifting operation shifts one interval of the representations by ε to the left such that this shift maintains the correctness of the representation; for an example see Figure 3.7a. The main goal of this section is to prove that by left-shifting we can always produce the left-most representation.

Proposition 3.12. *For $\varepsilon = \frac{1}{K}$ and $K \geq n$, an ε-grid representation $\mathcal{R} \in \mathfrak{Rep}$ is the left-most representation if and only if it is not possible to shift any single interval to the left by ε while maintaining correctness of the representation.*

Before proving the proposition, we describe some additional combinatorial structure of left-shifting. An interval v_i is called *fixed* if it is in the left-most position and cannot ever be shifted more to the left, i.e., $\ell_i = \min\{\ell'_i, \forall \mathcal{R}' \in \mathfrak{Rep}\}$. For example, an interval v_i is fixed if $\ell_i = \text{lbound}(v_i)$. A representation is the left-most representation if and only if every interval is fixed.

Obstruction Digraph. An interval v_i, having $\ell_i > \text{lbound}(v_i)$, can be left-shifted if it does not make the representation incorrect, and the incorrectness can be obtained in two ways. First, there could be some interval v_j, $v_j \lhd v_i$ such that $v_iv_j \notin E(G)$ and $\ell_j + 1 + \varepsilon = \ell_i$; we call v_j a *left-obstruction* of v_i. Second, there could be some interval v_j, $v_i \lhd v_j$ such that $v_iv_j \in E(G)$ and $\ell_i + 1 = \ell_j$ (so v_i and v_j are touching); then we call v_j a *right-obstruction* of v_i. In both cases, we first need to move v_j before moving v_i.

For the current representation $\mathcal{R}$, we define the *obstruction digraph* H on the vertices of G as follows. We put $V(H) = V(G)$ and $(v_i, v_j) \in E(H)$ if and only if v_j is an obstruction of v_i. For an edge (v_i, v_j), if $v_j \lhd v_i$, we call it a *left-edge*; if $v_i \lhd v_j$, we call it a *right-edge*. As we apply left-shifting, the structure of H changes; see Figure 3.7b.

Lemma 3.13. *An interval v_i is fixed if and only if there exists an oriented path in H from v_i to v_j such that $\ell_j = \text{lbound}(v_j)$.*

Proof. Suppose that v_i is connected to v_j by a path in H. By definition of H, $v_xv_y \in E(H)$ implies that v_y has to be shifted before v_x. Thus v_j has to be shifted before moving v_i which is not possible since $\ell_j = \text{lbound}(v_j)$.

On the other hand, suppose that v_i is fixed, i.e., $\ell_i = \inf\{\ell'_i : \forall \mathcal{R}'\}$. Let H' be the induced subgraph of H of the vertices v_j such that there exists an oriented path from v_i to v_j. If for all $v_j \in H'$, $\ell_j > \text{lbound}(v_j)$, we can shift all vertices of H' by ε to the left which constructs a correct representation and contradicts that v_i is fixed. Therefore, there exists $v_j \in H'$ having $\ell_j = \text{lbound}(v_j)$ as requested. $\square$

For example in Figure 3.7, if $\ell_2 = \text{lbound}(v_2)$, then the intervals v_2, v_4, v_3, v_7 and v_5 are fixed. Also, we can easily prove:

Lemma 3.14. *If $\varepsilon = \frac{1}{K}$ and $K \geq \frac{n}{2}$, the obstruction digraph H is acyclic.*

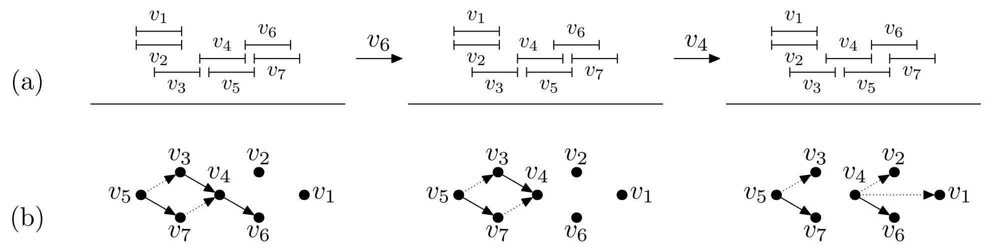

Figure 3.7: (a) A representation modified by left-shifting of v_6 and v_4. (b) The corresponding obstruction digraph H for each of the representations. Only sinks of the obstruction digraph can be left-shifted.

Proof. Suppose for contradiction that H contains some cycle $u_1, \dots, u_c$. This cycle contains a left-edges and b right-edges. Recall that if (u_i, u_{i+1}) is a left-edge, then $\ell_{u_{i+1}} = \ell_{u_i} - 1 - \varepsilon$, and if it is a right-edge, $\ell_{u_{i+1}} = \ell_{u_i} + 1$ (and similarly for (u_c, u_1)). If we go along the cycle from u_1 to u_1, the initial and the final positions have to be the same. Therefore $a(1+\varepsilon) = b$.

Now if this equation holds, then a has to be a multiple of K. Therefore $a \ge K$ and $b \ge K+1$, and thus $n \ge c = a + b \ge 2K+1$ which is not possible. □

We note that the assumption $K \ge \frac{n}{2}$ is necessary. For every $\varepsilon = \frac{1}{K}$, there exists a representation of a graph with $2K+1$ vertices having a cycle in H. The graph contains two cliques $v_0, \dots, v_{K-1}$ and $w_0, \dots, w_K$ such that v_i is also adjacent to $w_0, \dots, w_i$. Then the assignment $\ell_{v_0} = 0$, $\ell_{v_i} = \ell_{v_0} + i\varepsilon$ and $\ell_{w_i} = \ell_{v_0} + 1 + i\varepsilon$ is a correct representation. Observe that H contains a cycle $w_k v_{k-1} w_{k-1} v_{k-2} w_{k-2} \dots v_1 w_1 v_0 w_0 w_k$. See Figure 3.8 for $K = 3$.

Predecessors of Poset $\mathfrak{Rep}$. A representation $\mathcal{R}' \in \mathfrak{Rep}$ is a *predecessor* of $\mathcal{R} \in \mathfrak{Rep}$ if $\mathcal{R}' < \mathcal{R}$ and there is no representation $\bar{\mathcal{R}} \in \mathfrak{Rep}$ such that $\mathcal{R}' < \bar{\mathcal{R}} < \mathcal{R}$. We denote the predecessor relation by $\prec$. In a general poset, predecessors may not exist. But they always exist for a poset of a discrete structure like $(\mathfrak{Rep}, \le)$: Indeed, there are finitely many representations $\bar{\mathcal{R}}$ between any $\mathcal{R}' < \mathcal{R}$, and thus the predecessors always exist. Also, for any two representations $\mathcal{R}' < \mathcal{R}$, there exists a finite *chain* of predecessors $\mathcal{R}' = \mathcal{R}_0 \prec \mathcal{R}_1 \prec \cdots \prec \mathcal{R}_k = \mathcal{R}$.

For the poset $(\mathfrak{Rep}, \le)$, we are able to fully describe the predecessor structure:

Lemma 3.15. *The representation $\mathcal{R}'$ is a predecessor of $\mathcal{R}$ if and only if $\mathcal{R}'$ is obtained from $\mathcal{R}$ by applying one left-shifting operation.*

Proof. Clearly, if $\mathcal{R}'$ is obtained from $\mathcal{R}$ by one left-shifting, it is a predecessor of $\mathcal{R}$.

On the other hand, suppose we have $\mathcal{R}' < \mathcal{R}$. Let H be the obstruction digraph of $\mathcal{R}$ and $\bar{H}$ be the subgraph of H induced by the intervals having different positions in $\mathcal{R}$ and $\mathcal{R}'$. Then there are no directed edges from $\bar{H}$ to $H \setminus \bar{H}$ (otherwise $\mathcal{R}'$ would be an incorrect representation). According to Lemma 3.14, the digraph $\bar{H}$ is acyclic. Therefore, it contains at least one sink v_i. By left-shifting v_i in $\mathcal{R}$, we create a correct representation $\bar{\mathcal{R}} \in \mathfrak{Rep}$. Clearly, $\mathcal{R}' \le \bar{\mathcal{R}} \prec \mathcal{R}$, and so $\mathcal{R}'$ is a predecessor of $\mathcal{R}$ if and only if $\mathcal{R}' = \bar{\mathcal{R}}$. □

Proof of Left-shifting Proposition. The main proposition of this subsection is a simple corollary of Lemma 3.15.

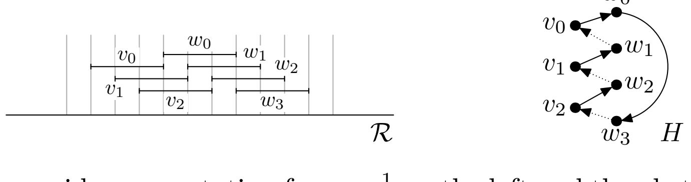

Figure 3.8: An ε-grid representation for $\varepsilon = \frac{1}{3}$ on the left and the obstruction digraph H containing a cycle on the right.

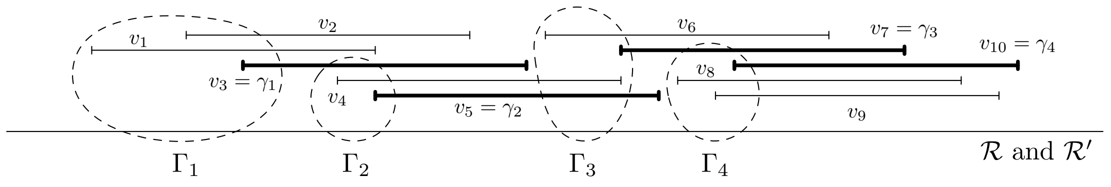

Figure 3.9: Both representations $\mathcal{R}$ and $\mathcal{R}'$ in one figure, with intervals of $\mathcal{R}'$ depicted in bold. The left endpoints of intervals of each group are enclosed by dashed curves, and these curves are ordered from left to right according to $<$.

Proof of Proposition 3.12. The left-most representation $\mathcal{R}$ is $\inf(\mathfrak{Rep})$, so it has no predecessors and nothing can be left-shifted. On the other hand, if $\mathcal{R} < \mathcal{R}'$, there is a chain of predecessors in between which implies that it is possible to left-shift some interval. ☐

3.3.4 Preliminaries for the Almost Quadratic-time Algorithm

Before describing the almost quadratic-time algorithm, we present several results which simplify the graph and the description of the algorithm.

Pruned Graph. The obstruction digraph H may contain many edges since each vertex v_i can have many obstructions. But if v_i has many, say, left-obstructions, these obstructions have to be positioned the same. If two intervals u and v have the same position in a correct unit interval representation, then $N[u] = N[v]$ and they are indistinguishable. Our goal is to construct a *pruned graph* G' which replaces each group of indistinguishable vertices of G by a single vertex. This construction is not completely straightforward since indistinguishable vertices may have different lower and upper bounds.

Let $\Gamma_1, \dots, \Gamma_k$ be the vertices partitioned by groups of indistinguishable vertices (and the groups are ordered by $<$ from left to right). We construct a unit interval graph G', where the vertices are $\gamma_1, \dots, \gamma_k$ with $\text{lbound}(\gamma_i) = \max\{\text{lbound}(v_j) : v_j \in \Gamma_i\}$, and the edges $E(G')$ correspond to the edges between the groups of G.

Suppose that we have the left-most representation $\mathcal{R}'$ of the pruned graph G' and we want to construct the left-most representation $\mathcal{R}$ of G. Let Γ_ℓ be a group. We place each interval $v_i \in \Gamma_\ell$ as follows. Let $\gamma_\leftarrow^\ell$ be the first non-neighbor of γ_ℓ on the left and $\gamma_\rightarrow^\ell$ be the right-most neighbor of γ_ℓ (possibly $\gamma_\rightarrow^\ell = \gamma_\ell$). We set

$$\ell_i = \max\{\text{lbound}(v_i), \ell_{\gamma_\leftarrow^\ell} + 1 + \varepsilon, \ell_{\gamma_\rightarrow^\ell} - 1\}, \tag{3.8}$$

and if $\gamma_\leftarrow^\ell$ does not exist, we ignore it in max. The meaning of this formula is to place each interval as far to the left as possible while maintaining the structure of $\mathcal{R}'$. Figure 3.9 contains an example of the construction of $\mathcal{R}$.

Before proving correctness of the construction of $\mathcal{R}$, we show two general properties of the formula (3.8). The first lemma states that each interval $v_i \in \Gamma_\ell$ is not placed in $\mathcal{R}$ too far from the position of γ_ℓ is $\mathcal{R}'$.

Lemma 3.16. *For each $v_i \in \Gamma_\ell$, it holds*

$$\ell_{\gamma_\ell} - 1 \le \ell_i \le \ell_{\gamma_\ell}. \tag{3.9}$$

Proof. The first inequality is true since $\ell_{\gamma_\ell} - 1 \le \ell_{\gamma^\ell_\rightarrow} - 1 \le \ell_i$ holds according to (3.8) and the ordering $<$ for $\mathcal{R}'$. The second inequality holds since since $\mathcal{R}'$ is a correct bounded representation, and so ℓ_{γ_ℓ} is greater than or equal to each term in (3.8). $\square$

The second lemma states that the representations $\mathcal{R}$ and $\mathcal{R}'$ are *intertwining* each other. If $\mathcal{R}$ is drawn on top of $\mathcal{R}'$, then the vertices of each group Γ_ℓ are in between of $\gamma_{\ell-1}$ and γ_ℓ; see Figure 3.9.

Lemma 3.17. *For each $v_i \in \Gamma_\ell$, $\ell > 1$, it holds*

$$\ell_{\gamma_{\ell-1}} < \ell_i \le \ell_{\gamma_\ell}, \tag{3.10}$$

Proof. The second inequality holds by (3.9). For the first inequality, there are two possible cases why the groups $\Gamma_{\ell-1}$ and Γ_ℓ are distinct:

- The first case is when $\gamma^\ell_\leftarrow$ is a neighbor of $\gamma_{\ell-1}$. Then $\ell_{\gamma_{\ell-1}} \le \ell_{\gamma^\ell_\leftarrow} + 1 < \ell_i$; the first inequality holds since $\gamma^\ell_\leftarrow\gamma_{\ell-1} \in E(G')$ and $\mathcal{R}'$ is a correct representation, and the second inequality is given by (3.8).
- The second case is when $\gamma^\ell_\rightarrow$ is a non-neighbor of $\gamma_{\ell-1}$. Then $\ell_{\gamma_{\ell-1}} < \ell_{\gamma^\ell_\rightarrow} - 1 \le \ell_i$ by the fact that $\gamma_{\ell-1}\gamma^\ell_\rightarrow \notin E(G')$ and by (3.8).

In both cases, we get $\gamma_{\ell-1} < \ell_i$. $\square$

Now, we are ready to show correctness of the construction of $\mathcal{R}$.

Proposition 3.18. *From the left-most representation $\mathcal{R}'$ of the pruned graph G', we can construct the correct left-most representation $\mathcal{R}$ of G by placing the intervals according to (3.8).*

Proof. We argue the correctness of the representation $\mathcal{R}$. Let v_i and v_j be a pair of vertices of G. Let $v_iv_j \in E(G)$. If v_i and v_j belong to the same group Γ_ℓ, they intersect each other at position ℓ_{γ_ℓ} by (3.9). Otherwise let $v_i \in \Gamma_\ell$ and $v_j \in \Gamma_{\ell'}$, and assume that $\Gamma_\ell < \Gamma_{\ell'}$. Then $\ell_i \le \ell_{\gamma_\ell} \le \ell_j$ by the intertwining property (3.10). Also, $\ell_j \le \ell_{\gamma_{\ell'}} \le \ell_{\gamma^\ell_\rightarrow} \le \ell_i + 1$ since $\gamma_{\ell'}$ is a right neighbor of γ_ℓ and (3.9). Therefore, $\ell_i \le \ell_j \le \ell_i + 1$ and v_i intersects v_j in $\mathcal{R}$. Now, let $v_iv_j \notin E(G)$, $v_i \in \Gamma_\ell$, $v_j \in \Gamma_{\ell'}$ and $v_i < v_j$. Then $\ell_i \le \ell_{\gamma_\ell} \le \ell_{\gamma^{\ell'}_\leftarrow} \le \ell_j - 1 - \varepsilon$ by (3.8) and (3.9), so v_i and v_j do not intersect. So the assignment $\mathcal{R}$ is a correct representation of G.

It remains to show that $\mathcal{R}$ is the left-most representation of G. We can identify each γ_ℓ with one interval $v_i \in \Gamma_\ell$ having $\mathrm{lbound}(v_i) = \mathrm{lbound}(\gamma_\ell)$; for an example see Figure 3.9. So G' can be viewed as an induced subgraph of G. We want to show that the intervals of G' are represented in $\mathcal{R}$ exactly the same as in $\mathcal{R}'$. Since $\mathcal{R}|_{G'}$ ($\mathcal{R}$ restricted to G') is some representation of G' and $\mathcal{R}'$ is the left-most representation of G', $\ell'_{\gamma_\ell} \le \ell_{\gamma_\ell}$ for every γ_ℓ. By (3.9), we get $\ell'_{\gamma_\ell} = \ell_{\gamma_\ell}$.

We know that $\mathcal{R}|_{G'}$ is the left-most representation, or in other words each interval of G' is fixed in $\mathcal{R}$. The rest of the intervals are placed so that they are either trivially fixed by $\ell_i = \mathrm{lbound}(v_i)$, or they have as obstructions some fixed intervals from G', in which case they are fixed by Lemma 3.13. Therefore, every interval of G is fixed and $\mathcal{R}$ is the left-most representation. $\square$

For the pruned graph G', the obstruction digraph H has in and out-degree at most two. Each interval has at most one left-obstruction and at most one right-obstruction, and these obstructions are always the same. More precisely, if v_j is a left-obstruction of v_i, then $v_j = v^i_{\leftarrow}$, whereas if v_j is a right-obstruction of v_i, then $v_j = v^i_{\rightarrow}$.

The pruning operation can be done in time $\mathcal{O}(n+m)$, so we may assume that our graph G is already pruned and contains no indistinguishable vertices. And the structure of obstructions in G can be computed in time $\mathcal{O}(n+m)$ as well.

Position Cycle. For each interval in some ε-grid representation, we can write its position in this form:

$$\ell_i = \alpha_i + \beta_i\varepsilon, \qquad \alpha_i \in \mathbb{Z},\ \beta_i \in \mathbb{Z}_K, \tag{3.11}$$

where $\varepsilon = \frac{1}{K}$. In other words, α_i is an integer position of v_i in the grid and β_i describes how far is this interval from this integer position.

Concerning left-shifting, the values β_i are more important. We can depict $\mathbb{Z}_K = \{0,\dots,K-1\}$ as a cycle with K vertices where the value decreases clockwise. The value β_i assigns to each interval v_i one vertex of the cycle. The cycle $\mathbb{Z}_K$ together with marked positions of β_i's is called the *position cycle*. A vertex of the position cycle is called *taken* if some β_i is assigned to it, and *empty* otherwise. The position cycle allows us to visualize and work with left-shifting very intuitively. When an interval v_i is left-shifted, β_i cyclically decreases by one, so β_i moves clockwise along the cycle. For an illustration, see Figure 3.10.

If (v_i, v_j) is a left-edge of H, then $\beta_j = \beta_i - 1$, and if (v_i, v_j) is a right-edge, then $\beta_i = \beta_j$. So if v_j is an obstruction of v_i, β_j has to be very close to β_i (either at the same position or at the next clockwise position). If there is a big empty space in the clockwise direction from β_i, the interval v_i can be left-shifted many times (or till it becomes fixed by $\ell_i = \text{lbound}(v_i)$). Notice that if β_i is very close to β_j, it does not mean that ℓ_i is very close to ℓ_j because the values α_i and α_j are ignored in the position cycle.

3.3.5 The Almost Quadratic-Time Algorithm for BoundRep

We want to solve an instance of BoundRep with a prescribed ordering ◄. We work with an ε-grid which is different from the one in Section 3.2.1. The new value of ε is

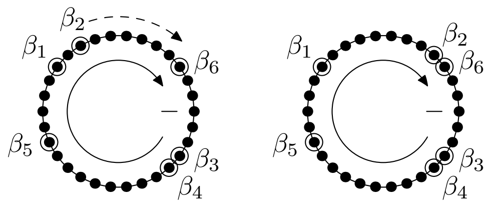

Figure 3.10: The position cycle with $\beta_1,\dots,\beta_6$ is on the left. We can shift β_2 in clockwise direction towards β_6, which gives a new representation with a new position cycle on the right. We note that after left-shifting, v_6 is not necessarily an obstruction of v_2.

the value given by (3.1) refined n times, so

$$\varepsilon = \frac{1}{n^2} \cdot \varepsilon'.$$

Lemma 3.5 applies for this value of ε as well, so if the instance is solvable, there exists a solution which is on this ε-grid.

The algorithm works exactly as the algorithm of Subsection 3.3.1. The only difference is that we can solve the linear program in time $\mathcal{O}(n^2 + nD(r))$, and now we describe how to do it. We assume that the input component is already pruned, otherwise we prune it and use Proposition 3.18 to complete the representation. We expect that the left-to-right order $\lhd$ of the vertices is given. The algorithm requires time $\mathcal{O}(nD(r))$ since the bounds are given in form $\frac{p_i}{q_i}$ and we need to perform arithmetic operations with these bounds.

Overview. The algorithm for one linear program works in three basic steps:

1. We construct an initial ε-grid representation (in ordering $\lhd$) having $\ell_i \geq \text{lbound}(v_i)$ for all intervals, using the algorithm of Corneil et al. [8].
2. We shift the intervals to the left while maintaining correctness of the representation until the left-most representation is constructed, using Proposition 3.12.
3. We check whether the left-most representation satisfies the upper bounds. If so, we have the left-most representation satisfying all bound constraints. This representation solves the linear program of Subsection 3.3.1 and minimizes E_t. Otherwise, the left-most representation does not satisfy the upper bound constraints, and thus by Lemma 3.11 no representation satisfying the upper bound constraints exists, and the linear program has no solution.

Input Size. Let r be the size of the input. A standard complexity assumption is that we can operate with polynomially large numbers (having $\mathcal{O}(\log r)$ bits in binary) in constant time, to avoid extra factor $\mathcal{O}(\log r)$ in the complexity of most of the algorithms. However, the value of ε given by (3.1) might require $\mathcal{O}(r)$ digits when written in binary. The assumption that we can computate with numbers having $\mathcal{O}(r)$ digits in contant time would break most of the computational models. Therefore, our computational model requires a larger time for arithmetic operations with numbers having $\mathcal{O}(r)$ digits in binary. For example, the best known algorithm for multiplication/division requires time $\mathcal{O}(D(r))$.

The problem is that a straightforward implementation of our algorithm working with the ε-grid would require time $\mathcal{O}(n^2 r^c)$ for some c instead of $\mathcal{O}(n^2 + nD(r))$. There is an easy way out. Instead of computing with *long numbers* having $\mathcal{O}(r)$ digits, we mostly compute with *short numbers* having just $\mathcal{O}(\log r)$ digits. Instead of the ε-grid, we mostly work in a larger Δ-grid where $\Delta = \frac{1}{n^2}$. The algorithm computes with the long numbers only in two places. First, some initial computations concerning the input are performed. Second, when the shifting makes some interval fixed, the algorithm estimes the final ε-grid position of the interval. All these computations can be done in total time $\mathcal{O}(nD(r))$ and we describe everything in detail later.

Left-Shifting. The basic operation of the algorithm is the LeftShift procedure which we describe here. We deal separately with fixed and unfixed intervals (and some intervals might be fixed initially). Unfixed intervals are on the Δ-grid and fixed intervals have precise positions calculated on the ε-grid. We place only unfixed intervals on the position cycle for the Δ-grid. At any moment of the algorithm, each vertex of the position cycle is taken by at most one β_i; this is true for the initial representation and the shifting keeps this property.

We define the procedure LeftShift(v_i) which shifts v_i from the position ℓ_i into a new position ℓ'_i such that the representation remains correct. The procedure LeftShift(v_i) consists of two steps:

1. Since v_i is unfixed, it has some β_i placed on the position cycle. Let k be such that the vertices $\beta_i+1,\dots,\beta_i+k$ of the position cycle are empty and the vertex β_i+k+1 is taken by some β_b. Then a candidate for the new position of v_i is $\bar{\ell}_i = \ell_i - k\Delta$.
2. We need to ensure that this shift from ℓ_i to $\bar{\ell}_i$ is valid with respect to $\text{lbound}(v_i)$ and the positions of the fixed intervals. Concerning the lower bound, we cannot shift further than $\text{lbound}(v_i)$. Concerning the fixed intervals, the shift is limited by positions of fixed obstructions of v_i. If v_j is a fixed left-obstruction, we cannot shift further than $\ell_j+1+\varepsilon$, and if $v_{j'}$ a fixed right-obstruction, we cannot shift further than $\ell_{j'}-1$.

The resulting position after applying LeftShift(v_i) is

$$\ell'_i = \max\{\bar{\ell}_i, \text{lbound}(v_i), \ell_j+1+\varepsilon, \ell_{j'}-1\}. \tag{3.12}$$

Lemma 3.19. *If the original representation $\mathcal{R}$ is correct, than the* LeftShift(v_i) *procedure produces a correct representation $\mathcal{R}'$.*

Proof. Clearly, the lower bound for v_i is satisfied in $\mathcal{R}'$. The shift of v_i from ℓ_i to ℓ'_i can be viewed as a repeated application of the left-shifting operation from Section 3.3.3. We just need to argue that each left-shifting operation can be applied till the position ℓ'_i is reached.

If at some point, the left-shifting operation could not be applied, there would have to be some obstruction v_j of v_i. There is no unfixed obstruction since all vertices of the position cycle $\beta_i+1,\dots,\beta_i+k$ are empty. And v_j cannot be fixed as well since we check positions of both possible obstructions. So there is no obstruction v_j. Therefore, by repeated applying the left-shifting operation, the interval v_i gets at a position ℓ'_i and the resulting representation is correct. ☐

After LeftShift(v_i), if $\bar{\ell}_i$ is not a strict maximum of the four terms in (3.12), the interval v_i becomes fixed; either trivially since $\ell'_i = \text{lbound}(v_i)$, or by Lemma 3.13 since v_i becomes obstructed by some fixed interval. In such a case, we remove β_i from the position cycle.

Fast Implementation of Left-Shifting. Since we apply the LeftShift procedure repeatedly, we want to implement it in time $\mathcal{O}(1)$. Considering the terms in (3.12), the first term $\bar{\ell}_i$ is a short number (on the Δ-grid) and the remaining terms are long numbers (on the ε-grid). We first compare $\bar{\ell}_i$ to the remaining terms which are three comparisons of short and long numbers and we are going to show how to compare them in $\mathcal{O}(1)$. If $\bar{\ell}_i$ is a strict maximum, we use it for ℓ'_i. Otherwise, we need to compute the maximum of the remaining three terms which takes time $\mathcal{O}(D(r))$. But then the interval v_i becomes fixed, and so this costly step is done exactly n times, and takes the total time $\mathcal{O}(nD(r))$.

Lemma 3.20. *With the total precomputation time $\mathcal{O}(nD(r))$, it is possible to compare $\bar{\ell}_i$ to the remaining terms in (3.12) in time $\mathcal{O}(1)$ per* LeftShift *procedure.*

Proof. Initially, we do the following precomputation for the lower bounds. By the input, we have b lower bounds given in the form $\frac{p_1}{q_1}, \dots, \frac{p_b}{q_b}$ as irreducible fractions. For each bound, we first compute its position (α_i, β_i) on the ε-grid; see (3.11).

If $\text{lbound}(v_i) \ll \text{lbound}(v_j)$ for some vertices v_i and v_j, then $\text{lbound}(v_i)$ is never achieved since the graph is connected and every representation takes space at most n. Therefore we can increase $\text{lbound}(v_i)$ without any change in the solution of the instance. More precisely, let $\alpha = \max \alpha_i$. Then we modify each bound by setting $\alpha_i := \max\{\alpha - n - 1, \alpha_i\}$. In addition, we shift all the bounds by substructing a constant C such that each $\alpha_i - C \in [0, n+1]$. Concerning β_i, we round the position (α_i, β_i) down to a position $(\alpha_i, \bar{\beta}_i)$ of the Δ-grid. These precomputations can be done for all lower bounds in time $\mathcal{O}(nD(r))$.

Suppose that we want to find out whether $\bar{\ell}_i \le \text{lbound}(v_j) = \alpha_j + \beta_j \cdot \varepsilon$ where $\bar{\ell}_i$ is in the Δ-grid. Then it is sufficient to check whether $\bar{\ell}_i \le \alpha_j + \bar{\beta}_j \Delta$ which can be done in constant-time since both α_j and $\bar{\beta}_j$ are short numbers.

When v_j becomes fixed, its precise position is computed using (3.12). Then we compute the values $\ell_j - 1$ and $\ell_j + 1 + \varepsilon$ used in (3.12) and round them down to the Δ-grid. Using these precomputed values, $\bar{\ell}_i$ can be compared with the remaining terms in (3.12) in time $\mathcal{O}(1)$. When an interval becomes fixed, time $\mathcal{O}(D(r))$ is used. Since each interval becomes fixed exactly once, this rounding takes the total time $\mathcal{O}(nD(r))$. □

Notice that the representation is constructed in a position shifted by C. Later, before checking the upper bound, we shift the whole representation back.

Shifting Phases. All shifting of the algorithm is done by repeated application of the LeftShift procedure. Using Lemma 3.19, we know that the representation created in each step is correct. We apply the procedure in such a way that each interval is almost always shifted by almost one. Recall that $\Delta = \frac{1}{n^2}$ and the position cycle has n^2 vertices. The initial Δ-grid representation is obtained by the algorithm of Corneil et al. [8]. We shift it such that $\ell_i \ge \text{lbound}(v_i)$ for each v_i and $\ell_i \le \text{lbound}(v_i) + \Delta$ for some v_i. In such a case, each interval is shifted in total by at most $\mathcal{O}(n)$, and in total we apply the LeftShift procedure $\mathcal{O}(n^2)$ times.

The initial representation obtained by the algorithm of Corneil et al. [8] places all intervals in such a way that β_i's are positioned equidistantly in the position cycle;

refer to the left-most position cycle in Figure 3.11. (So there is a gap of size n between each consecutive pair of β_i's.)

The shifting of unfixed intervals proceeds in two phases. The first phase creates one big gap by clustering all β_i's in one part of the cycle. To do so, we apply the LeftShift procedure to each interval, in the order given by the position cycle. Of course, some intervals might become fixed and disappear from the position cycle. We obtain one big gap of size at least $n(n-1)$. Again, refer to Figure 3.11.

In the second phase, we use this big gap to shift intervals one by one, which also moves the cluster along the position cycle. Again, if some interval becomes fixed, it is removed from the position cycle. The second phase finishes when each interval becomes fixed and the left-most representation is constructed. For an example, see Figure 3.12.

Putting Together. For a pseudocode, see Algorithm 7 in Appendix A.7. First, we show correctness of the shifting algorithm and its complexity:

Lemma 3.21. *For a component having n vertices such that the size of the input is r, the shifting algorithm construct a correct left-most representation in time $\mathcal{O}(n^2 + nD(r))$.*

Proof. First, we argue correctness of the algorithm. The algorithm starts with an initial representation which is correct and satisfies the lower bounds. By Lemma 3.19, after applying each LeftShift procedure, the resulting representation is still correct. The algorithm keeps a correct list of fixed intervals which is increased by shifting. So after finitely many applications of the LeftShift procedure, every interval becomes fixed, and we obtain the left-most representation.

Concerning complexity, all precomputations take total time $\mathcal{O}(nD(r))$. Using Lemma 3.20, each LeftShift(v_i) procedure can be applied in time $\mathcal{O}(1)$ unless v_i becomes fixed. The first phase is applying the LeftShift procedure $n-1$ times. In the second phase, each interval is shifted by at least $\frac{n-1}{n}$ (unless it becomes fixed). Since each interval can be shifted by at most $\mathcal{O}(n)$ from its initial position, the second phase applies the LeftShift procedure $\mathcal{O}(n^2)$ times. So the total running time of the algorithm is $\mathcal{O}(n^2 + nD(r))$. □

We are ready to prove that BoundRep with a prescribe ordering ◀ can be solved in time $\mathcal{O}(n^2 + nD(r))$:

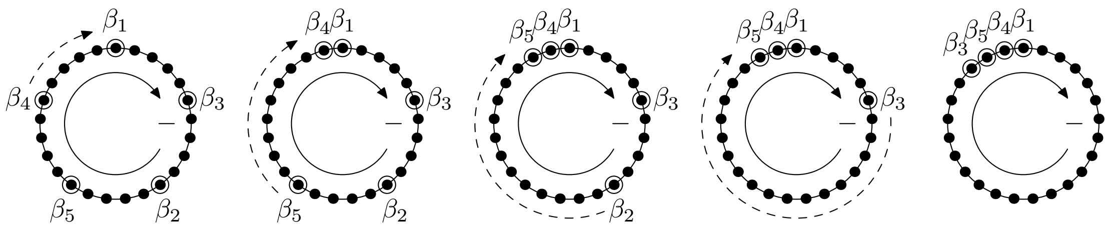

Figure 3.11: The position cycle during the first phase, changing from left to right. The first phase clusters the β_i's by moving β_4, β_5, β_2 and β_3 towards β_1. When LeftShift(v_2) is applied, v_2 becomes fixed and β_2 disappears from the position cycle.

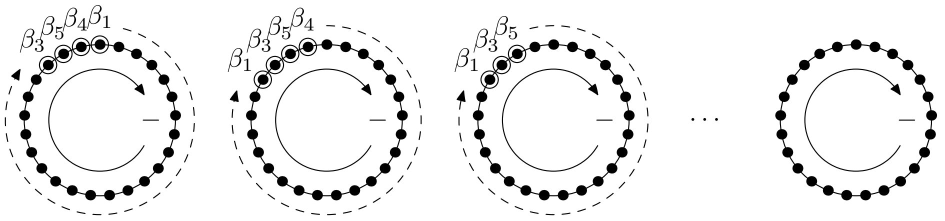

Figure 3.12: The position cycle during the second phase, changing from left to right. We shift β_i's across the big gap till all β_i's disappear.

Proof of Theorem 1.4. We proceed exactly as in the algorithm of Section 3.3.1, so we process the components $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$ from left to right, and for each of them we solve two linear programs. For each linear program, we find the left-most representation using Lemma 3.21, and we test for this representation (shifted back by C) whether the upper bounds are satisfied. According to Lemma 3.11, the linear program is solvable if and only if the left-most representation satisfy the upper bounds, and clearly the left-most representation minimizes E_t. The time complexity of the algorithm is $\mathcal{O}(n^2 + nD(r))$ and the proof of correctness is exactly the same as in Proposition 3.12. $\square$

We finally present an FPT algorithm for BoundRep with respect to the number of components c. The algorithm is based on Theorem 1.4.

Proof of Corollary 1.5. There are $c!$ possible left-to-right orderings of the components of G. For each of them, we can decide in time $\mathcal{O}(n^2 + nD(r))$ whether there exists a bounded representation in the order, using Theorem 1.4. So the total time necessary is $\mathcal{O}((n^2 + nD(r))c!)$. $\square$

3.4 Extending Unit Interval Graphs

The RepExt(UNIT INT) problem can be solved using Theorem 1.4. We just need to show that it is a particular instance of BoundRep with the known ordering $\blacktriangleleft$ of the components:

Proof of Corollary 1.6. The graph G contains unlocated components and located components. Similarly to Section 3.1, unlocated components can be placed far to the right and we can deal with them using standard recognition algorithm.

Concerning located components $C_1, \dots, C_c$, they have to be ordered from left to right, which gives the required ordering $\blacktriangleleft$. We straightforwardly construct the instance of BoundRep with this $\blacktriangleleft$ as follows. For each pre-drawn interval v_i at position ℓ_i, we put $\text{lbound}(v_i) = \text{ubound}(v_i) = \ell_i$. For the rest of the intervals, we set no bounds. Clearly, this instance of BoundRep is equivalent with the original RepExt(UNIT INT) problem. And we can solve it in time $\mathcal{O}(n^2 + nD(r))$ using Theorem 1.4. $\square$

4 Extending Chordal Graphs and Their Subclasses

There are four types Fixed, Sub, Add, and Both which form the following lattice, given by Figure 4.1. For a type $\mathfrak{T}$, we denote by $\text{Gen}(\mathfrak{T}, T')$ a set of all tree T which we can generate from T' using the modification of type $\mathfrak{T}$. In addition, if T' contains pre-drawn subtrees, trees in $\text{Gen}(\mathfrak{T}, T')$ contain these pre-drawn subtrees as well (possibly modified by subdivision). The ordering of types given by the lattice has this property: If $\mathfrak{T} \le \mathfrak{T}'$, then $\text{Gen}(\mathfrak{T}, T') \subseteq \text{Gen}(\mathfrak{T}', T')$.

$$\begin{tikzcd}
 & \textsc{Both} \arrow[dl, dash] \arrow[dr, dash] & \\
\textsc{Sub} \arrow[dr, dash] & & \textsc{Add} \arrow[dl, dash] \\
 & \textsc{Fixed} &
\end{tikzcd}$$

Figure 4.1: The lattice formed by four types Fixed, Sub, Add, and Both.

Now, whether a given instance is solvable depends on the set $\text{Gen}(\mathfrak{T}, T')$; so if this set contains more trees, it only helps for solving the problem. Let $\mathfrak{T} \le \mathfrak{T}'$. If an instance of Recog* or RepExt is solvable for type $\mathfrak{T}$, then it is solvable for type $\mathfrak{T}'$ as well. And equivalently, if it is not solvable for $\mathfrak{T}'$, it is also not solvable for $\mathfrak{T}$.

For the types Add and Both (and Subfor PROPER P-in-P and P-in-P), the set $\text{Gen}(\mathfrak{T}, T')$ contains a tree having an arbitrary tree T as a subtree. Therefore, the Recog* problem for these types is equivalent to the standard Recog problem, and we can use known polynomial-time algorithms.

For PROPER P-in-P and P-in-P classes, the types Add and Sub behave differently. The type Add allows to extend the ends of the paths. The type Sub allows to expand the middle of the path but if the endpoint of the path is contained in some pre-drawn subpath, it remains to be contained after the subdivision. The type Both makes the problems equivalent to the Recog and RepExt problems for the real line.

Indistinguishable Vertices. Let u and v be two vertices of G such that $N[u] = N[v]$. These two vertices are called *indistinguishable* since they can be represented exactly the same, having $R_u = R_v$ (a common property of all intersection representations). From the structural point of view, *groups of indistinguishable* vertices are not very interesting. The goal is to construct a pruned graph where each group is reprented by a single vertex. For that, we need to be little careful since we cannot prune pre-drawn vertices.

For an arbitrary graph, its groups of indistinguishable vertices can be located in time $\mathcal{O}(n + m)$ [41]. We prune the graph in the following way. If u and v are indistinguishable and u is not pre-drawn, we eliminate u from the graph (and for the representation, we can put $R_u = R_v$). The resulting pruned graph has the following property: If two vertices u and v indistinguishable, they are both pre-drawn. For the rest of this chapter, we expect that all input graphs are pruned.

Maximal Cliques. There is the following property of maximal cliques, valid for all representations by subtrees inside a tree. Let $x \in T$ and consider the set $K = \{u : u \in V(G), x \in R_u\}$. Clearly, K is a clique of G.

On the other, let K be a maximal clique of G. Subtrees of tree have the Helly property which states that every pairwise intersecting collection of subtrees have a common intersection as a subtree. Since the subtrees representing K are pair-wise intersecting, the common intersection $R_K = \cap_{u \in K} R_u$ is a subtree of T. This subtree R_K is not intersected by any other R_v for $v \notin K$ (otherwise K would not be a maximal clique). Thus the subtrees R_K corresponding to different maximal cliques are pairwise disjoint. For example, if $|T|$ is smaller than the number of maximal cliques of G, the graph is clearly not representable in T.

4.1 Subpaths-in-Path Graphs

In this section, we deal with classes PROPER P-in-P and P-in-P. The results obtained here are used as tools for P-in-T and T-in-T graphs in Section 4.2.

4.1.1 Polynomial Cases

First we deal with all polynomial cases. Also, we describe several concepts as minimum span, useful in the rest of the paper.

Non-fixed Type Recognition. The only limitation for recognition of interval graphs inside a given path is the length of the path. In all three types Sub, Add and Both, we can produce a path as long as necessary. (With a trivial exception $T' = P_0$ for Sub, for which the instance is solvable only if $G = K_n$.) For a subpath-in-path representation, only the order of the endpoints from left to right is important, not the exact positions. In a tree T with at least $2n$ vertices, every possible ordering is realizable.

Thus the problems are equivalent to the standard recognition of interval graphs on the real line. For PROPER P-in-P, it can be solved in time $\mathcal{O}(n+m)$, for example [33, 8]. Similarly for P-in-P, it can be solved in linear-time [6, 9].

Both Type Extension. This extension type is equivalent with the partial representation extension problems of interval graphs on the real line. Again only the ordering of the endpoints is important. The only change here is that some of the endpoints are already placed. By subdividing, we can place any amount of the endpoints between any two endpoints (not sharing the same position). Also, the path can be extended to the left and to the right which allows to place any amount of endpoints to the left of the left-most pre-drawn endpoint and to the right of the right-most pre-drawn endpoint. So any extending ordering can be realized in the BOTH type.

The partial representation extension problem for interval graphs on the real line was first considered in [29]. The paper gives algorithms for both classes INT and PROPER INTand does not explicitly deal with representations sharing endpoints but the algorithms are easy to modify. The results of Theorems 1.1 and 1.2, and [5] show that the both extension problems are solvable in time $\mathcal{O}(n+m)$.

Sub Type Extension. It is possible to modify the algorithms of Theorems 1.1 and 1.2 to deal with the SUB type. Instead of describing details of these representations, we show a reduction of the SUBtype extension to the BOTHtype extension which we can solve in time $\mathcal{O}(n+m)$ as discussed above.

Theorem 4.1. *The both* SUB *type extension problems* REPEXT(PROPER P-in-P, SUB) *and* REPEXT(P-in-P, SUB) *can be solved in time* $\mathcal{O}(n+m)$.

Proof. First, we describe the algorithm for P-in-P. Let $p_1, \dots, p_t$ be the vertices of the path T'. If the graph contains unlocated components, we first deal with them. They can be placed if one of the following works, and otherwise the representation is not extendible.

- If there is only a single located component, then at least one of p_1 and p_t has to be not contained in any pre-drawn interval. If so, we can subdivide that end of the path T' and place all the unlocated components there. Otherwise, the unlocated components cannot be placed at all.
- If there are more located components, they have to be ordered correctly from left to right $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$; see Section 1.5. Now, we can subdivide an edge between these components and place the unlocated components there.

If we succeed, we remove unlocated components from the graph. What remains is to deal with the located components.

If at least one of p_1 and p_t is contained in some pre-drawn interval, we modify both the path and the graphs. If $v_1, \dots, v_k \in C_1$ such that $p_1 \in R_{v_1}, \dots, R_{v_k}$, we modify as follows. First, we extend the path by one by adding p_0 attached to p_1. We introduce an additional pre-drawn interval $v_\leftarrow$ adjacent exactly to $v_1, \dots, v_k$ with $R_{v_\leftarrow} = \{p_0\}$ and we modify $R'_{v_i} = R_{v_i} \cup \{p_0\}$. Exactly the same modification introducing p_{t+1} and $v_\rightarrow$ is used for the right-end if there is some $v \in C_c$ such that $p_t \in R_v$.

We use the described algorithm for REPEXT(P-in-P, BOTH) for the modified graph and the modified path, which runs in time $\mathcal{O}(n+m)$. What remains is to argue correctness, which we do only for left end of T; for the right end the argument is symmetric. If nothing pre-drawn contains p_1, the edge p_1p_2 can be subdivided as necessary, thus creating the exactly same situation as in BOTH type. On the other hand, if p_0 was added, everything with exception of $v_1, \dots, v_k$ has to be represented on the right of $v_\leftarrow$ which is placed on p_0.

But there is a single problem we need to deal with. The newly added edge p_0p_1 can be subdivided. There are two possibilities. If $|R_{v_i}| \ge 2$ for each i, the subdivision of p_0p_1 is equivalent to subdivision of p_1p_2 which is correct in the original problem. Now let $|R_{v_i}| = 1$ for some i. Then $N(v_i)$ has to be a complete subgraph. In such a case, we can revert the subdivision of p_0p_1 as follows. Let $p'_1, \dots, p'_s$ be the newly created vertices by subdividing p_0p_1. For each $v \in N(v_i)$ (of course, with exception of $v_\leftarrow$), we put $R'_v = R_v \setminus \{p'_1, \dots, p'_s\} \cup \{p_1\}$, and we remove $p'_1, \dots, p'_s$ by contractions.

Now, by removing p_0, p_{t+1}, $v_\leftarrow$ and $v_\rightarrow$ (of course, only if they are added), we obtain a correct representation of G inside a subdivision of T' extending the partial representation. Concerning PROPER P-in-P, we use almost the same approach. One change is that we append two vertices $\bar{p}_0$ and p_0 (resp. p_{t+1} and $\bar{p}_{t+1}$) to the end of T' and put $R_{v_\leftarrow} = \{\bar{p}_0, p_0\}$ (resp. $R_{v_\rightarrow} = \{p_{t+1}, \bar{p}_{t+1}\}$), so the resulting partial representation is proper. Concerning the case $|R_{v_i}| = 1$ for some i, it is trivial since $k = 1$ and $N(v_i)$ is empty, and the component C_1 has to be the single interval. ☐

For a pseudocode, see Algorithm 8 in Appendix A.8.

General Properties of PROPER P-in-P. In each representation, we have some ordering $\lhd$ of intervals from left to right. This is an ordering of left endpoints from left to right (and at the same time the ordering of the right endpoints). In Section 3.1, we describe this ordering $\lhd$ and a partial ordering $<$ in details. Since the graphs we consider in this chapter are pruned, the indistinguishable vertices which are incomparable in $<$ are ordered by the partial representation.

Now let us consider the types FIXED and ADD. The space on the path is limited and it is important to know how much space does a component C require. We denote it by minspan(C). Let $\mathcal{R}$ be some representation of C, p_i be the left-most vertex of T' contained in representation of C and p_j be the right-most such vertex. Then

$$\mathrm{minspan}(C) = \begin{cases} \min_{\forall \mathcal{R}}\{j - i + 1\} & \text{if some representation of } C \text{ exists,} \\ +\infty & \text{otherwise.} \end{cases}$$

Lemma 4.2. *For every component C (both located, or unlocated), the value* minspan(C) *can be computed in $\mathcal{O}(n + m)$ (together with a realizing representation).*

Proof. First, we deal with an unlocated component. We start by constructing any ordering $\lhd$ of the left endpoints of intervals from left to right from Lemma 3.2, in time $\mathcal{O}(n + m)$ using the algorithm of Corneil et al. [8]. Using this ordering, we want to produce a representation as small as possible.

Let ℓ_i denote the left endpoint and r_i the right endpoint of the interval v_i. From ordering $v_1 \lhd \cdots \lhd v_n$, we want to compute a common ordering $\lessdot$ of both the left endpoints and the right endpoints, exactly as in the proof of Lemma 3.3. The starting point is an ordering of just the left endpoints $\ell_1 \lessdot \cdots \lessdot \ell_n$. Into this ordering, we insert right endpoints $r_1, \dots, r_n$ one-by-one. A right endpoint r_i is inserted right before ℓ_j where v_j is the left-most non-neighbor of v_i on the right (or if such v_j does not exist, we append r_i to the end). For an example of construction of $\lessdot$, see Figure 4.2.

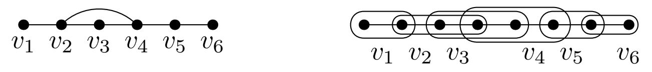
v_1 v_2 v_3 v_4 v_5 v_6
v_1 v_2 v_3 v_4 v_5 v_6

Figure 4.2: The ordering $\lessdot$ is $\ell_1 \lessdot \ell_2 \lessdot r_1 \lessdot \ell_3 \lessdot \ell_4 \lessdot r_2 \lessdot r_3 \lessdot \ell_5 \lessdot r_4 \lessdot \ell_6 \lessdot r_5 \lessdot r_6$ for the component on the left. The constructed smallest possible representation of the component C on the right, with $\mathrm{minspan}(C) = 8$.

Now, we construct the smallest representation as follows. Let $p_1, \dots, p_k$ be the vertices of the tree T. We construct an assignment f which maps endpoints into T. Then for a vertex v_i we put $R_{v_i} = \{p_j : f(\ell_i) \le p_j \le f(r_i)\}$. The mapping f is constructed for endpoints one-by-one, according to $\lessdot$. If the previous endpoint in $\lessdot$ had assigned vertex p_i, we assign to the current endpoint: If the current endpoint is a right endpoint and the previous endpoint is a left endpoint, assign p_i. Otherwise assign p_{i+1}. In total, the component needs $2n - \ell$ vertices of T where ℓ denotes the number of changes from a left endpoint to a right endpoint in the ordering $\lhd$; so $2n-\ell$ is the value of $\mathrm{minspan}(C)$. For an example, see Figure 4.2. The total complexity of the algorithm is clearly $\mathcal{O}(n + m)$. The constructed representation is correct and it can be argued the same as in the proof of Lemma 3.3.

Concerning minimality notice that in a pruned graph it always holds $\ell_i \ne \ell_j$ and $r_i \ne r_j$ for every pair v_i and v_j. We argue that we use gaps as small as possible. Only a right endpoint r_i following a left endpoint ℓ_j can be placed at the same position. The other case of a right endpoint r_i followed by a left endpoint ℓ_j requires a gap of size one; otherwise R_{v_i} would intersect R_{v_j} but $v_iv_j \notin E$. So the gaps are minimal and we construct a smallest representation and give the value $\mathrm{minspan}(C)$ correctly.

Concerning a located component, we just modify the above approach slightly. The ordering $\lhd$ has to be compatible with the ordering of the pre-drawn intervals. For a group of indistinguishable intervals, all its intervals are pre-drawn and we know their ordering from left to right. So we just test whether $<$ and its reversal can be used. We have (at most) two possibilities for $\lhd$ which give the same $\mathrm{minspan}(C)$ but the minimum representations might be differently shifted, and we are able to construct both of them. If the pre-drawn intervals do not belong to one group, the ordering $\lhd$ is uniquely defined (if it is compatible with the ordering of pre-drawn intervals at all).

We compute the common ordering $\lessdot$ as before and place the endpoints in this ordering. The only exception is that the endpoints of the pre-drawn intervals are fixed. Additionally, we place the endpoints on the left of the left-most pre-drawn endpoint as far to the right as possible and similarly as far to the left as possible on the right of the right-most pre-drawn endpoint. The constructed representation is the smallest possible and gives $\mathrm{minspan}(C)$. □

General Properties of P-in-P. We use the maximal clique approach described in Section 2.2.1. Recall properties of maximal cliques from introduction of this chapter. For a component C, we denote by $\mathrm{cl}(C)$ the number of maximal cliques of C. Since the subtrees R_K corresponding to the maximal cliques are disjoint, they have to be ordered from left to right. There is the following well-known property of this ordering [12]:

Lemma 4.3 (Fulkerson and Gross). *A graph is an interval graph if and only if there exists an ordering of the maximal cliques $K_1 < \cdots K_{\mathrm{cl}(C)}$ such that for every vertex the cliques containing this vertex appear consecutively in this ordering.*

We quickly argue correctness of the lemma. Clearly, in an interval representation, all maximal cliques corresponding to one vertex v are appearing consecutively (otherwise the clique in between would be intersected by R_v in addition). On the other hand, having an ordering $<$ of the maximal cliques from the statement, we can construct a representation as follows. To each clique K_i, assign one vertex p_i of T (this vertex corresponds to the clique point from Section 2.2.1). Now for each vertex v, we assign $R_v = \{p_i : v \in K_i\}$. Since the maximal cliques appear consecutively, we assign a subpath to each interval. Also, the representation is correct.

For types FIXED and ADD, we again consider minimum span defined exactly as for proper interval graphs above. Clearly, $\mathrm{minspan}(C) \geq \mathrm{cl}(C)$. We show:

Lemma 4.4. *For an unlocated component C, $\mathrm{minspan}(C) = \mathrm{cl}(C)$ if C is a component of an interval graph. We can find a smallest representation in time $\mathcal{O}(n+m)$.*

Proof. We start by identifying maximal cliques in time $\mathcal{O}(n+m)$, using algorithm of Rose et al. [41]. To construct a smallest representation, we find an ordering from Lemma 4.3, using the PQ-tree algorithm [6] in time $\mathcal{O}(n+m)$. If such an ordering does not exist, the graph G is not an interval graph and no representation exists. If the ordering exists, we can construct a representation using exactly $\mathrm{cl}(C)$ vertices of the path as described above. □

Fixed Type Recognition. We just need to use the values minspan we already know how to compute.

Proposition 4.5. *Both* RECOG*(PROPER P-in-P, FIXED) *and* RECOG*(P-in-P, FIXED) *can be solved in time $\mathcal{O}(n+m)$.*

Proof. We process components $C_1, \dots, C_c$ one-by-one and place them from left to right on T'. If $\sum_{i=1}^{c} \mathrm{minspan}(C_i) \leq |T'|$, we can place the components using the smallest representation from Lemma 4.2 for PROPER P-in-P, resp. Lemma 4.4 for P-in-P. Otherwise, the path is too small and the representation cannot be constructed. □

Add Type Extension, PROPER P-in-P. We use the same technique as in the algorithm of Section 3.3.1 for unit interval graphs, with the difference that we can use minimum spans and Lemma 4.2 instead of linear programming.

Theorem 4.6. *The problem* REPEXT(PROPER INT, ADD) *can be solved in time $\mathcal{O}(n+m)$.*

Proof. Since the path can be expanded to left and to right as much as necessary, we can place unlocated components far to the left. So we only need to deal with located components, ordered $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$ from left to right. We place components from left to right. When we place C_i, it has to be placed on the right of C_{i-1}. We have (at most) two possible smallest representations corresponding to two different orderings of C_i. We test whether at least one of them can be placed on the right of C_{i-1} and pick the one minimizing the right-most vertex of T taken by C_i (leaving maximum possible space for $C_{i+1}, \dots, C_c$). If neither representation can be placed, the extension algorithm fails.

Now, if the algorithm finishes, it construct a correct representation. On the other hand, we place each component as far to the left as possible (while restricted by the previous components on the left). So if C_i cannot be placed, there exists no representation extending the partial representation. □

4.1.2 NP-complete Cases

The basic gadgets of the reduction are paths. They have the following minimum spans.

Lemma 4.7. *For* P-in-P*,* $\mathrm{minspan}(P_n) = n$*. For* PROPER P-in-P *and* $n \geq 2$*,* $\mathrm{minspan}(P_n) = n + 2$*.*

Proof. For P-in-P, the number of maximal cliques of P_n is n. For PROPER P-in-P, the ordering $\lessdot$ is

$$\ell_0 \lessdot \ell_1 \lessdot r_0 \lessdot \ell_2 \lessdot r_1 \lessdot \cdots \lessdot \ell_i \lessdot r_{i-1} \lessdot \cdots \lessdot \ell_n \lessdot r_{n-1} \lessdot r_n.$$

So there are n changes from ℓ_i to r_{i-1} and the minimum span is $n + 2$. □

We reduce the problem from 3-Partition. The input of 3-Partition consists of integers k, M and $A_1, \dots, A_{3k}$ such that $\frac{M}{4} < A_i < \frac{M}{2}$ for each A_i and $\sum A_i = kM$. It asks whether it is possible to partition A_i's into k triples such that the sets A_i of each triple sum to exactly M.[1] This problem is known to be strongly NP-complete (even with all integers of a polynomial size) [14].

Theorem 4.8. *The both* Fixed *type problems* RepExt(PROPER P-in-P, Fixed) *and* RepExt(P-in-P, Fixed) *are* NP*-complete.*

Proof. The described reduction works for both PROPER P-in-P and P-in-P with small modifications. For a given input of 3-Partition (with $M \geq 4$), we construct a graph G and its partial representation. As the fixed tree we choose $T' = P_{(M+1)k}$, with the vertices $p_0, \dots, p_{(M+1)k}$.

Now, the graph G contains two types of gadgets as separate components. First, it contains $k + 1$ *split gadgets* $S_0, \dots, S_k$ which split the path into k gaps of equal sizes M. Then it contains $3k$ *take gadgets* $T_1, \dots, T_{3k}$. A take gadget T_i takes in each representation space at least A_i of one of the k gaps.

For this reduction, the gadgets are particularly simple. The split gadget S_i is just a single pre-drawn vertex v_i with $R_{v_i} = \{p_{(M+1)i}\}$. The split gadgets clearly split the path into k equal gaps of size M. The take gadget T_i is a P_{A_i} for P-in-P, resp. P_{A_i-2}

[1] Notice that if a subset of A_i's sums to exactly M it has to be a triple due to size conditions.

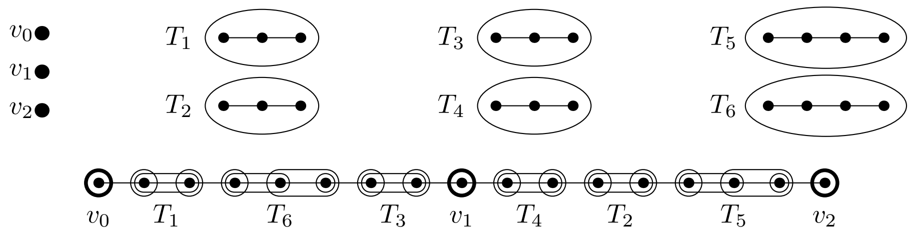

Figure 4.3: An example of the reduction for the input set of 3-Partition: $k = 2$, $M = 7$, $A_1 = A_2 = A_3 = A_4 = 2$ and $A_5 = A_6 = 3$. On top, the constructed subpath-in-path graph is depicted. On bottom, the partial representation (depicted in bold) is extended.

for PROPER P-in-P. According to Lemma 4.7, $\mathrm{minspan}(T_i) = A_i$. The representation is extendible if and only if it is possible to place the take gadgets into the k gaps. For an illustration, see Figure 4.3. The reduction is clearly polynomial.

To conclude the proof, we show that the partial representation is extendible if and only if the corresponding 3-Partition input has a solution. If the partial representation is extendible, the take gadgets T_i are divided into the k gaps on the path which forms a partition. Based on conditions for sizes of A_i's, each gap contains exactly three take gadgets of the total minimum span M; thus the partition solves the 3-Partition problem. On the other hand, a solution of 3-Partition describes how to place the take gadgets into the k gaps and construct the extending representation. ☐

Corollary 4.9. *The problem* RepExt(P-in-P, Add) *is* NP*-complete.*

Proof. We use the above reduction with one additional pre-drawn interval v attached to everything in G. We put $R_v = \{p_0, \dots, p_{(M+1)k}\}$, so it contains the whole tree T'. Now since representation of each T_i has to intersect R_v, it has to be placed inside of the k gaps as before. ☐

4.1.3 Parametrized Complexity

In this subsection, we study parametrized complexity. The parameters are the number of components c, the number of pre-drawn intervals k and the size of the tree t.

By Number of Components. In the above reduction, one might ask whether it is possible to make the reduction graph G connected. For P-in-P, it is indeed possible to add universal vertices adjacent to everything in G and thus making G connected, as in the proof of Corollary 4.9. The following results answers this question for PROPER P-in-P negatively (unless P = NP):

Proposition 4.10. *The problem* RepExt(PROPER P-in-P, Fixed) *is fixed-parameter tractable in the number of components c, solvable in time $\mathcal{O}((n+m)c!)$.*

Proof. There is $c!$ possible orderings of the components from left to right, and we test each of them. We show that for a given ordering of components, we can solve the problem in time $\mathcal{O}(n+m)$; thus gaining the total time $\mathcal{O}((n+m)c!)$. We solve the problem almost the same as in the proof of Theorem 4.6. The only difference is that we deal with all the components instead of only the located components.

We process the components from left to right. When we process C_i, we place it on the right of C_{i-1} as far to the left as possible. For unlocated C_i, we can take any ordering. For located C_i, we test both orderings and take the one placing C_i further to the left. We construct the representation in time $\mathcal{O}(n+m)$ and the algorithm is clearly correct; see the proof of Theorem 4.6 for more details. □

It is necessary for NP-hardness of the problem RepExt(PROPER P-in-P, Fixed) to have some pre-drawn subpaths. On the other hand, also some unlocated components are necessary. If all the components would be located, there is a unique ordering ◀ and we can test it in time $\mathcal{O}(n+m)$ as described above. In general, for c components and c' located components, we need to test only $c!/c'!$ different orderings.

By Number of Pre-drawn Intervals. In the reduction in Theorem 4.8, we need to have k pre-drawn intervals. One could ask, whether the problems becomes simpler with a small number of pre-drawn intervals. We answer this negatively. For PROPER P-in-P, the problem is in XP and W[1]-hard with respect to k. For P-in-P, we show that it is W[1]-hard.

We start with two closely related problems BinPacking and GenBinPacking. In the both problems, we have k bins and n items of integer sizes. The question is whether we can pack (partition) these items into the k bins when sizes of the bins are limited. For BinPacking, all the bins have the same size. For GenBinPacking, the bins have different sizes. Formally:

Problem:	BinPacking
Input:	Integers k, ℓ, V and $A_1,\dots,A_\ell$.
Question:	Does there exist a k-partition $\mathcal{P}_1,\dots,\mathcal{P}_k$ of $A_1,\dots,A_\ell$ such that $\sum_{A_i\in\mathcal{P}_j} A_i \le V$ for every $\mathcal{P}_j$.

Problem:	GenBinPacking
Input:	Integers k, ℓ, $V_1,\dots,V_k$ and $A_1,\dots,A_\ell$.
Question:	Does there exist a k-partition $\mathcal{P}_1,\dots,\mathcal{P}_k$ of $A_1,\dots,A_\ell$ such that $\sum_{A_i\in\mathcal{P}_j} A_i \le V_j$ for every $\mathcal{P}_j$.

Lemma 4.11. *The problems* BinPacking *and* GenBinPacking *are polynomially equivalent.*

Proof. Obviously BinPacking is a special case of GenBinPacking. On the other hand, let k, ℓ, $V_1,\dots,V_k$ and $A_1,\dots,A_\ell$ be an instance of GenBinPacking. We construct an instance k', ℓ', V' and $A'_1,\dots,A'_{\ell'}$ of BinPacking as follows. We put $k'=k$, $\ell'=\ell+k$ and $V'=2\cdot\max V_i+1$. The weights of the first ℓ items are the same, i.e, $A'_i=A_i$ for $i=1,\dots,\ell$. The additional items $A'_{\ell+1},\dots,A'_{\ell+k}$ are called *large* and we put $A'_{\ell+i}=V'-V_i$ for $i=1,\dots,k$.

Now, each bin has to contain exactly one large item since two large items take more space than V'. After placing large items into the bins, we obtain bins of sizes

$V_1, \ldots, V_k$ in which we have to place the remaining items. This corresponds exactly to the original GenBinPacking instance. □

If the input sizes are encoded in binary, the problem is NP-complete even for $k = 2$. The more interesting version which we use here is that the sizes are encoded in unary so all sizes are polynomial. In such a case, the BinPacking problem is known to be solvable in time $t^{\mathcal{O}(k)}$ using dynamic programming where t is the total size of all items. And it is W[1]-hard with respect to the parameter k [24]. Similar holds for RepExt(PROPER P-in-P, Fixed):

Proof of Theorem 1.7. For a given instance of the BinPacking problem, we can solve it by RepExt(PROPER P-in-P, Fixed) in a similar manner as in the reduction in Theorem 4.8. As T', take a path $P_{(V+1)k}$. As G, take P_{A_i-2} for each A_i and pre-drawn vertices $v_0, \ldots, v_k$ such that $R_{v_i} = \{p_{(V+1)i}\}$. The rest of the argument is exactly as in the proof of Theorem 4.8.

Now, we want to solve RepExt(PROPER P-in-P, Fixed) using 2^k instances of GenBinPacking (which is polynomially equivalent to BinPacking), where k is the number of pre-drawn intervals.

First we deal with located components $C_1 \blacktriangleleft \cdots \blacktriangleleft C_c$. For each component, we have two possible orderings $<$ and using Lemma 4.2 we get (at most) two possible smallest representations which might be differently shifted. In total, we have at most $2^c \leq 2^k$ possible representations keeping $C_1, \ldots, C_c$ as small as possible leaving maximum possible gaps for unlocated components. We test each of these representations.

Now, let $C'_1, \ldots, C'_{c'}$ be the unlocated components. For each C'_i, we compute $\mathrm{minspan}(C'_i)$ using Lemma 4.2. The goal is to place unlocated components in the $c+1$ gaps between representations of the located components $C_1, \ldots, C_c$. We can solve this problem using GenBinPacking as follows. We have $k+1$ bins of sizes equal to the gaps between the representations of $C_1, \ldots, C_c$. We have c' items of sizes $A_i = \mathrm{minspan}(C'_i)$. The solution of GenBinPacking tells in which gaps the unlocated components can be placed. If there exists no solution, this specific smallest representation of located components can not be used.

We can test all 2^k possible representations. Thus we get the required weak truth-table reduction. □

For a pseudocode of one reduction, see Algorithm 9 in Appendix A.9.

Corollary 4.12. *The problem* RepExt(PROPER P-in-P, Fixed) *is* W[1]*-hard and belongs to* XP*, solvable in time $n^{\mathcal{O}(k)}$ where k is the number of pre-drawn intervals.*

Proof. Both obtained easily by Theorem 1.7. □

Proposition 4.13. *The problems* RepExt(P-in-P, Fixed) *and* RepExt(P-in-P, Add) *are* W[1]*-hard with respect to the parameter k where k is the number of pre-drawn intervals.*

Proof. Modify the reduction in Theorem 4.8 and Corollary 4.9 as in the proof of Theorem 1.7. □

By Size of the Path. We show that the NP-complete cases are fixed-parameter tractable with respect to the size of the path t. It is easy to find a solution by a brute-force algorithm:

Proposition 4.14. *For t the size of T', the problems* RepExt(PROPER P-in-P, Fixed) *and* RepExt(P-in-P, Fixed) *are fixed-parameter tractable with respect to the parameter t. They can be solved in time $\mathcal{O}(n + m + f(t))$ where $f(t) = t^{2t^2}$.*

Proof. In a pruned graph, the non-pre-drawn vertices has to be represented pairwise different. There are at most t^2 possible different subpaths of a path with t vertices so the pruned graph can contain at most t^2 non-pre-drawn vertices; otherwise the extension is not possible. We can test every possible assignment of t^2 non-pre-drawn vertices to t^2 subpaths, and for each assignment we test whether it is a correct representation. □

4.2 Path and Chordal Graphs

We present several results concerning P-in-T and T-in-T classes. We use many results from Section 4.1 as basic tools here.

4.2.1 Polynomial Cases

The recognition problem for types Add and Both is equivalent to standard recognition without any additional tree T'. Indeed, we can modify T' by adding an arbitrary tree to it. If the input graph is either P-in-T or T-in-T, there exists a tree T'' in which the graph can be represented. We produce T by attaching T'' to T' in any way. Clearly, the graph can be represented in T as well, completely ignoring the part T' for example.

For path graphs, the original recognition algorithm is due to Gavril [15] in time $\mathcal{O}(n^4)$. The current fastest algorithm is by Schäffer [43] in time $\mathcal{O}(nm)$. For chordal graphs, there is a beautiful simple algorithm in time $\mathcal{O}(n + m)$ by Rose et al. [41].

4.2.2 NP-complete Cases

All the remaining cases from the table of Figure 1.7 are NP-complete. We describe a modification of the reduction of Theorem 4.8 for path and chordal graphs. We start with the simplest reduction for the Fixed type and then modify it for the other problems.

Fixed Type. For Fixed type, we can avoid pre-drawn subtrees, using an additional structure of the tree.

Proposition 4.15. *Both* Recog*(P-in-T, Fixed) *and* Recog*(T-in-T, Fixed) *are* NP-*complete.*

Proof. We again reduce from 3-Partition with an input k and M, and for technical purposes let $M \geq 4$. We construct a graph G and a tree T as follows. The tree T is a

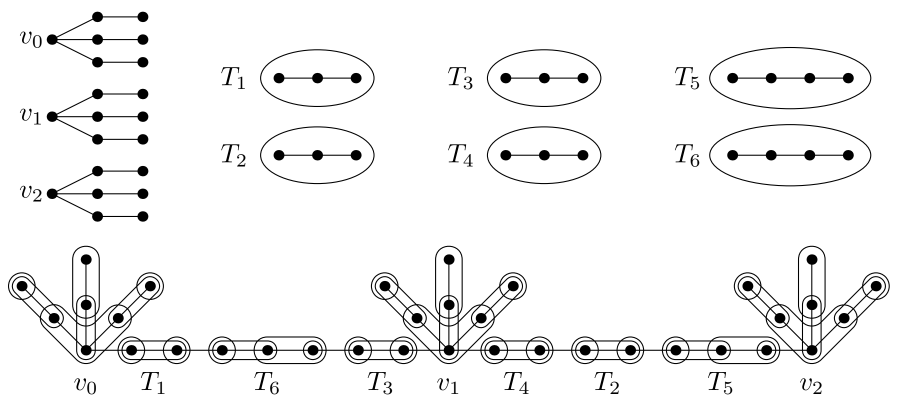

Figure 4.4: For the same input set of 3-Partition, on top the constructed graph is depicted. On bottom, a representation is constructed, giving a solution $\{A_1, A_3, A_6\}$ and $\{A_2, A_4, A_5\}$.

path $P_{(M+1)k}$ (denote its vertices $p_0, \dots, p_{(M+1)k}$) with additional three paths of length two attached to every vertex $p_{(M+1)i}$, for each $i = 0, \dots, k$; see Figure 4.4.

Each split gadget S_i is a star, depicted on the left of Figure 4.4. When the split gadgets are placed as in the figure, they split the tree into k gaps exactly as the pre-drawn vertices in the proof of Theorem 4.8. Each take gadget T_i is the same path P_{A_i} as before.

What remains is to argue correctness of the reduction. We claim that each R_{v_i} contains at least one branch vertex (actually exactly one, since there are exactly $n+1$ branch vertices in T). If some R_{v_i} would only contain non-branch vertices, then it would not be possible to represent three disjoint neighbors u_1, u_2 and u_3 of this v_i since each $R_{u_j} \setminus R_{v_i}$ has to be non-empty.

Now since each branch vertex is taken by one R_{v_i}, the tree T is split into k gaps as before. Since $|A_i| > 2$, each T_i can be represented only inside of these gaps. Notice that the total sum of free vertices in the gaps has to be kM, and therefore the split gadgets has to be represented entirely in the attached stars as in Figure 4.4. The rest of the reduction works exactly as in Theorem 4.8. □

Sub Type. By modifying the above reduction, we get:

Theorem 4.16. *The problems* Recog*(P-in-T, Sub) *and* Recog*(T-in-T, Sub) *are* NP-*complete.*

Proof. We need to modify the two gadgets from the reduction in Theorem 4.15 in such a way that subdivision of the tree does not help in placing them. Subdivision only increases the number of non-branch vertices. Thus a take gadget T_i requires A_i branch vertices. Similarly, the split gadget S_i is more complicated. See Figure 4.5 on top.

The tree T is constructed as follows. We start with a path $P_{(M+1)k}$ with vertices $p_0, \dots, p_{(M+1)k}$. To each vertex $p_{(M+1)i}$ we attach a subtree isomorphic to the trees in Figure 4.5 on bottom. To the other vertices of the path, we attach one leaf per a vertex.

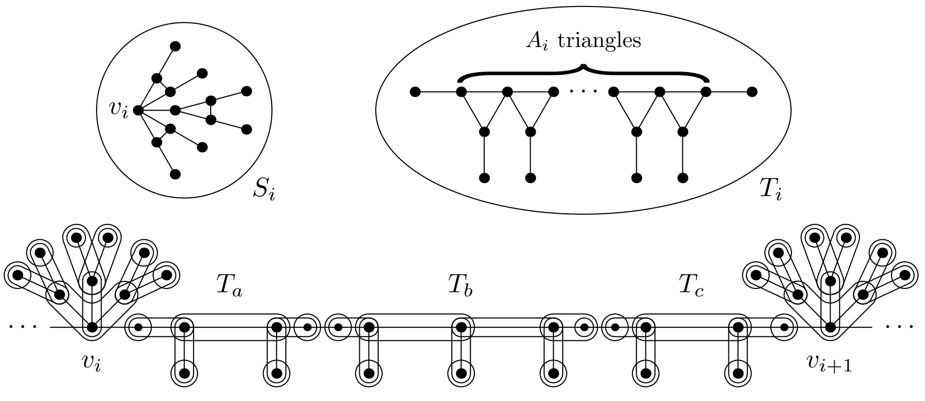

Figure 4.5: On top, the split gadget S_i on left and the take gadget T_i on right. On bottom, a part of the tree T, the small vertices are added by subdivision. The gap between two split gadgets contains three take gadgets T_a, T_b and T_c giving one triple $\{A_a, A_b, A_c\}$ with $A_a + A_b + A_c = M$.

Clearly, for a given solution of 3-Partition, we can construct a correct representation in a subdivided tree. On the other hand, we are going to show how to construct a solution of 3-Partition from a given tree representation.

Recall maximal cliques from introduction of this chapter. Notice that each triangle $u_1u_2u_3$ in the gadgets is a maximal cliques K (and for each triangle, we get a different maximal clique). As such, R_K has to contain a branch vertex since $N[u_i] \neq N[u_j]$ for each $i \neq j$. The gadget S_i contains three triangles, each taking one branch vertex of T. In addition, their R_K's are connected by R_{v_i} which has to contain another branch vertex. So in total, S_i contains at least four branch vertices. The gadget T_i contains A_i triangles, and so it requires at least A_i branch vertices. Since the number of branch vertices of T is limited, each S_i takes exactly four branch vertices and each T_i takes exactly A_i branch vertices.

Now, if some T_i would contain a branch vertex of the subtrees attached to $p_{(M+1)j}$, at least on its branch vertices would not be used (either not taken by T_i or T_i would require at least $A_i + 1$ branch vertices). So each S_i has to take branch vertices of the subtrees attached to $p_{(M+1)j}$ for some j, and the take gadgets have to be placed inside the gaps exactly as before. □

Theorem 4.17. *Even with only a single subtree pre-drawn, i.e, $|G'| = 1$, the problems* RepExt(T-in-T, Add) *and* RepExt(T-in-T, Both) *are* NP*-complete.*

Proof. We easily modify the above reductions; for Add type, the reduction of Proposition 4.15, for Both type, the reduction of Theorem 4.16. The modification adds one pre-drawn vertex v into G adjacent to everything such that $R_v = T'$. The representation of v spans the whole tree and thus forces the entire representation into T'.

We now deal just with Add type, for Both is the argument exactly the same. Let T' be the partial tree and let T be the tree in which the representation is constructed, so T' is a subtree of T. We claim that we can restrict a representation of each vertex of G into T' and thus obtain a correct representation inside the subtree T'.

Let $x \in V$. Since $xv \in E$, the intersection of R_x and T' is a non-empty subtree. So $R'_x = R_x \cap T'$ is a representation of G in T'. To argue the correctness, let x and y be two different vertices from v (otherwise trivial). If $xy \notin E$, then $R_x \cap R_y = \emptyset$, and so $R'_x \cap R'_y = \emptyset$ as well. Otherwise xyv is a triangle in G, and thus by the Helly property the subtrees R_x, R_y and $R_v = T'$ have a non-empty common intersection, giving $R'_x \cap R'_y$ non-empty. This gives a reduction from $\textsc{RepExt}(\textsf{T-in-T}, \textsc{Fixed})$. □

For path graphs, one can use a similar technique of a pre-drawn universal vertex attached to everything. But there is the following difficulty: To do so, the input partial tree T' has to be a path. For type $\textsc{Both}$, the complexity of $\textsc{RepExt}(\textsf{P-in-T}, \textsc{Both})$ remains open. For type $\textsc{Add}$, we get the following weaker result:

Proposition 4.18. *The problem* $\textsc{RepExt}(\textsf{P-in-T}, \textsc{Add})$ *is* $\textsf{NP}$*-complete.*

Proof. Similarly as in Theorem 4.17, add a pre-drawn universal vertex v on the path T' constructed in the reduction in Theorem 4.8 such that $R_v = T'$. The rest is exactly as above. □

4.2.3 Parametrized Complexity

We deal with parametrized complexity of the problems and we give only minor and partial results in this direction. Unlike in Section 4.1, parametrization by number of pre-drawn subtrees k is mostly not helpful. We show that every problem with exception of $\textsc{RepExt}(\textsf{P-in-T}, \textsc{Add})$ is already $\textsf{NP}$-complete for $k = 0$ or $k = 1$. For $\textsc{RepExt}(\textsf{P-in-T}, \textsc{Add})$, we have only a weaker result that it it $\textsf{W[1]}$-hard with respect to the paramater k since Proposition 4.13 straightforwardly generalizes.

Similarly, the low number of components c does not make the problem any easier. We can easily modify the above reductions to show that all problems remain $\textsf{NP}$-complete even if the graph G is connected.

Concerning size of the tree, Proposition 4.14 straightforwardly generalizes:

Proposition 4.19. *Let t be the size of T'. The problems* $\textsc{RepExt}(\textsf{P-in-T}, \textsc{Fixed})$ *and* $\textsc{RepExt}(\textsf{T-in-T}, \textsc{Fixed})$ *are fixed-parameter tractable with respect to t. They can be solved in time $\mathcal{O}(n + m + f'(t))$ where $f(t) = 2^{t2^t}$.*

Proof. Proceed exactly as before, prune the graph and test all possible assignments. The only difference is that T has at most 2^t subtrees. □

We note that a more precise bound for the number of subtrees could be use but we did not try to better estimate the function f'.

5 Conclusions

In this thesis, we show recent developments in the study of the partial representation extension problems. We deal with classes of interval graphs, proper interval graphs, unit interval graphs and chordal graphs. We describe concepts and techniques common to these algorithms, and we believe that they can be applied to other classes as well.

We conclude this thesis by describing two related problems to the partial representation extension problem. Also, we give some open problems.

5.1 Simultaneous Representations

In the introduction, we give a formal definition of the simultaneous representations problem, considered recently by Jampani and Lubiw [23, 22]. To recall the problem, the input gives graphs $G_1, \dots, G_k$ and the problem asks whether there exist representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ assigning the same sets to the common vertices I. For a class $\mathcal{C}$, we denote this problem by $\textsc{SimRep}(\mathcal{C})$.

Here, we show that for many classes $\mathcal{C}$, the simultaneous representations problem is closely related to the partial representation extension problem, and many techniques useful for one problem can be applied to the other problem as well. This relation is more like a general principle than a precise mathematical statement.

Partial Representation Extension 'solves' Simultaneous Representations. Let the size of I be small. If there exists only a small number of different representations of the subgraph induced by the vertices of I, we can test all of them and solve $\textsc{SimRep}$ using $\textsc{RepExt}$. So if the $\textsc{RepExt}$ problem is solvable in polynomial time, we can get a fixed-parameter tractable algorithm with parameter $i = |I|$. We show this for interval graphs:

Proposition 5.1. *We can solve* $\textsc{SimRep}(\mathsf{INT})$ *in time* $\mathcal{O}((n+m)\cdot(2i)!)$*, where* n *is the total number of vertices of all graphs and* m *is the total number of edges of all graphs.*

Proof. If a representation $\mathcal{R}_I$ of I would be given, we could use the algorithm for $\textsc{RepExt}(\mathsf{INT})$ to test whether it is possible to extend it to a simultaneous representation of all the graphs. We just need to test for every graph G_j whether it is possible to extend the partial representation $\mathcal{R}_I$ to a representation of entire G_j. This can be done in time $\mathcal{O}(n+m)$, using Theorem 1.1.

Since only the left-to-right order of endpoints is important, an interval graph with i vertices has $\mathcal{O}((2i)!)$ topologically different representations. Therefore, we can test all possible representation of I and get the running time $\mathcal{O}((n+m)\cdot(2i)!)$. $\square$

Similar relation holds for some other classes, for example PROPER INT. We note that this is currently the best known algorithm for SimRep(INT) when $k > 2$, no polynomial-time algorithm is known. For $k = 2$, Jampani and Lubiw [23] give an algorithm in time $\mathcal{O}(n^2 \log n)$, and a recent paper of Bläsius and Rutter [5] improves the running time to $\mathcal{O}(n+m)$.

Simultaneous Representations "solves" Partial Representation Extension. Another relation is that we are sometimes able to solve RepExt by SimRep, which was suggested to us by Lubiw. Suppose that we have a graph G and a partial representation $\mathcal{R}'$ of G'. We construct an instance of SimRep for $k = 2$ as follows: $I = G'$, $G_1 = G$ and G_2 contains additional auxiliary vertices. The purpose of these auxiliary vertices is to fix a representation of I to be topologically equivalent to the partial representation $\mathcal{R}'$. Therefore, the solution $\mathcal{R}^1$ and $\mathcal{R}^2$ of the simultaneous representations problem gives a topologically correct representation $\mathcal{R}^1$ extending $\mathcal{R}'$.

For example in the case of interval graphs, we construct G_2 as follows. Consider the partial representation and add a long path consisting of short intervals on top of it. These intervals correspond to the auxiliary verticse added to G_2 and force a grid-like structure on the real line. This grid forces I to be represented topologically the same as in the partial representation $\mathcal{R}'$. Thus we can solve RepExt(INT) using SimRep(INT) which is used in [5] to obtain an algorithm solving RepExt(INT) in time $\mathcal{O}(n+m)$.

We note that this relation does not always work. First, it might not be possible to construct such a graph G_2, forcing a topologically unique representation of I. For example, the problem of simultaneous representations of chordal graphs is solvable in a polynomial time [22]. But the corresponding partial representation extension problem, considered in Chapter 4, is NP-complete; see Theorem 4.17. Second, it might happen that the topological equivalency of the representations in not sufficient, for example for partially represented unit interval graphs.

Applying Our Techniques. Concerning classes PROPER INT and UNIT INT, the simultaneous representations problem similarly distinguishes these classes. One can easily construct two graphs which have simultaneous proper interval representations but not simultaneous unit interval representations. We believe that many techniques developed in Chapter 3 can be applied for the SimRep(PROPER INT) and SimRep(UNIT INT) problems. We already have some partial results in this direction which are not yet written, and we just sketch our ideas here.

First, one needs to construct simultaneous left-to-right orderings $<_1, \dots, <_k$ having the same order on I. If these orderings do not exist, the simultaneous representations cannot be constructed. Using these orderings, we can easily construct simultaneous proper interval representations. For unit interval graphs, we can additionally apply linear programming or some shifting approach.

There are two main problems for which we do not know a solution yet. We are

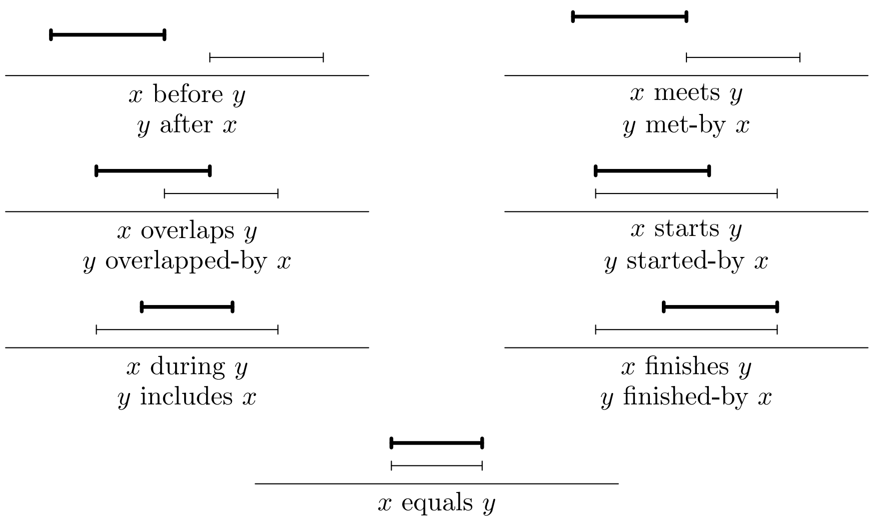

Figure 5.1: Thirteen primitive relations between the thick interval x and the thin interval y.

able to construct orderings $<_1, \dots, <_k$ only if the graphs $G_1, \dots, G_k$ are connected. If they are not connected, the construction of the orderings $<_1, \dots, <_k$ is not clear. Second, for shifting algorithm, we need to choose some initial representation, and it is not clear how many shifting iterations are necessary.

5.2 Allen Algebras

The following type of problems is studied in theory of artificial intelligence and time reasoning; see Golumbic [16] for a survey. Suppose that we have several events which happened at some time. To every event, we can assign an interval of the timeline. Now, for some pairs of events we know relations. Allowing shared endpoints, Allen [1] characterized thirteen *primitive relations* between the events, see Figure 5.1. A *relation* is a set of several primitive relations. For some pairs of events, we specify relations in which they can occur. For example, we can specify for events x and y that either x is before y, or x is during y.

We want to find intervals representing the events such that all the specified relations are satisfied. This is called the *interval satisfiability problem.* Vilian and Kautz [47] proved that this problem is NP-complete. Golumbic and Shamir [18] gave a more simple proof, using the interval graph sandwich problem.

Notice that we can describe RepExt(INT) and RepExt(PROPER INT) in the settings of Allen Algebras. In the case of RepExt(INT), we do it in the following way. If vertices are non-adjacent, we assign them the relation $\{\text{before}, \text{after}\}$. If they are adjacent, we assign the complement of the previous relation. For pairs of pre-drawn intervals, we give singleton relations. Similarly, we can specify RepExt(PROPER INT).

But these more general problem are NP-complete, so they do not help in solving RepExt(INT) and RepExt(PROPER INT). On the other hand, the results presented in this thesis show that some very specific problems concerning Allen Algebras are polynomially solvable.

5.3 Open Problems

We state several open problems.

Different Approach? The formulation of the first problem is not very precise. As we already described in the introduction, all the algorithm for solving partial representation extension problems we know are based on the following principle. First, we use some (known) way how to find all possible representations of an input graph. Then we derive some necessary conditions from the partial representations. Moreover, we show that these conditions are sufficient. Then, we test these conditions on all possible representations of the graph. If some representation satisfies them, we can use this representation to extend the input partial representation.

In the case of INT, we use PQ-trees. For PROPER INT and UNIT INT, we use uniqueness of the representation. The algorithms for extension of comparability, permutation and function graphs is based on properties of modular decomposition [25]. Also, the extension algorithm for planar graphs [2] uses SPQR-trees to work with all possible representations of the given planar graph.

Question 5.2. *Is it possible to solve some* RepExt *problem more "directly", without "testing" all possible representations?*

Faster Algorithm for Unit Interval Graphs. Concerning Chapter 3, we give only one open problem:

Problem 5.3. *Is it possible to solve* RepExt(UNIT INT) *in time $\mathcal{O}(r)$, where r is the size of the input?*

Two Problems for Chapter 4. One of the main goals of Chapter 4 is to stimulate future research in this area. Therefore, we give two open problems.

The first problems concerns the only open case in the table in Figure 1.7.

Problem 5.4. *What is complexity of* RepExt(P-in-T, Both)*?*

Concerning parametrized complexity, we believe it is useful to first attack problems related to interval graphs. This allows to develop tools for more complicated chordal graphs. A generalization of Theorem 1.7 and Corollary 4.12 for P-in-P seems to be particularly interesting. The PQ-tree approach seems to be a good starting point.

Problem 5.5. *Does* RepExt(P-in-P, Fixed) *belong to* XP *with respect to k where k is the number of pre-drawn intervals?*

A Algorithms

In this appendix, we give pseudocodes of the main algorithms of this thesis. These pseudocodes are not very detail and not written formally in any programming language. They are meant to be overviews of all important steps done in each algorithm.

A.1 Reordering PQ-tree, General Partial Order

The pseudocode is in Algorithm 1. It is described in Section 2.1.1.

Algorithm 1 Reordering a PQ-tree – REORDER($T, \lhd$)

```
Require: A PQ-tree $T$ and a partial ordering $\lhd$.
Ensure: A reordering $T'$ of $T$ such that $<_{T'}$ extends $\lhd$ if it exists.
1: Construct a digraph of $\lhd$.
2: Process the nodes of $T$ from bottom to the root:
3: for a processed node $N$ do
4:     Consider the subdigraph induced by the children of $N$.
5:     if the node $N$ is a P-node then
6:         Find a topological sorting of the subdigraph.
7:         If it exists, reorder $N$ according to it, otherwise output "no".
8:     else if the node $N$ is a Q-node then
9:         Test whether the current ordering or its reversal is compatible with the subdigraph.
10:        If yes, reorder the node, otherwise output "no".
11:    end if
12:    Contract the subdigraph into a single vertex.
13: end for
14: return A reordering $T'$ of $T$.
```

A.2 Reordering PQ-tree, Interval Order

The pseudocode is in Algorithm 2. The description is in Section 2.1.2.

Algorithm 2 Reordering a PQ-tree, interval order – Reorder($T, \lhd$)

Require: A PQ-tree T and an interval order $\lhd$ (with a normalized representation).
Ensure: A reordering T' of T such that $<_{T'}$ extends $\lhd$ if it exists.

```
Calculate handles for each invididual interval.

Process the nodes from bottom to the root:
for a processed node $N$ do
    Let the subtrees of $N$ represent sets $\mathcal{I}_1, \dots, \mathcal{I}_k$.
    Sort their lower and upper handles from left to right.
    if the node $N$ is a P-node then
        Find any topological sorting by repeated removing of minimal elements.
        If it exists, reorder $N$ according to it, otherwise output "no".
    else if the node $N$ is a Q-node then
        Test whether the current ordering or its reversal is compatible.
        Process the ordering from left to right, check for every element whether it is minimal and remove its handles from the common order of handles.
        If one ordering is compatible, reorder the node, otherwise output "no".
    end if
    Compute handles for $\mathcal{I} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$.
end for

return A reordering $T'$ of $T$.
```

A.3 Extending Interval Graphs

The pseudocode is in Algorithm 3. It is described in Section 2.2.

Algorithm 3 Extending Interval Graphs – RepExt(INT)

Require: An interval graph G and a partial representation $\mathcal{R}'$.
Ensure: A representation $\mathcal{R}$ extending $\mathcal{R}'$ if it exists.

```
Compute maximal cliques and construct a PQ-tree.
Compute $\curvearrowleft$ and $\curvearrowright$ and construct an interval order $\lhd$.
Use Algorithm 2 to reorder the PQ-tree according $\lhd$.
If any of these steps fails, no representation exists and output "no".

Place the clique-point in order from left to right, as far to the left as possible.
Construct a representation $\mathcal{R}$, using the clique-points.

return A representation $\mathcal{R}$ extending $\mathcal{R}'$.
```

A.4 Extending Proper Interval Graphs

The pseudocode is in Algorithm 4. The description is in Section 3.1.

Algorithm 4 Extending Proper Interval Graphs – RepExt(PROPER INT)

Require: A proper interval graph G and a partial representation $\mathcal{R}'$.
Ensure: A representation $\mathcal{R}$ extending $\mathcal{R}'$ if it exists.

```
1:  if pre-drawn intervals of some component do not appear consecutively then
2:      Return “no”, $\mathcal{R}'$ is not extendible.
3:  end if
4:  Assign to the components pairwise disjoint open segments containing their pre-
    drawn intervals.
5:  for each component do
6:      Construct the partial orderings $<$ and $<_{G'}$.
7:      if there exists an ordering $\lhd$ extending $<_{G'}$, and $<$ or its reversal then
8:          Construct a representation of the component in its open segment as follows.
9:          Compute a common ordering $\lessdot$, starting with $\ell_1 \lessdot \cdots \lessdot \ell_n$:
10:         for each $r_i$ do
11:             Insert $r_i$ right before $\ell_j$ such that $v_j$ is the first non-neighbor of $v_i$ on the
                right (or append $r_i$ to the end, if $v_j$ does not exist).
12:         end for
13:         Using $\lessdot$, construct a representation of the component by placing non-pre-
            drawn endpoints equidistantly into the gaps.
14:     else
15:         Return “no”, $\mathcal{R}'$ is not extendible.
16:     end if
17: end for
18: return A representation $\mathcal{R}$ extending $\mathcal{R}'$.
```

A.5 Bounded Representation – LP Approach

The pseudocode is in Algorithm 5. The description is in Section 3.3.1.

Algorithm 5 Bounded Representation of Unit Interval Graphs – BoundRep

Require: A unit interval graph G, bound contraints and ordering $\blacktriangleleft$.
Ensure: A representation $\mathcal{R}$ in the ordering $\blacktriangleleft$ satisfying the bounds if it exists.

```
 1: Compute the value $\varepsilon$ according to the bounds $\frac{p_1}{q_1}, \dots, \frac{p_b}{q_b}$:
        $$\varepsilon' := \frac{1}{\mathrm{lcm}(q_1, q_2, \dots, q_b)}, \qquad \text{and} \qquad \varepsilon := \frac{\varepsilon'}{n}.$$
 2: Process the components $C_1, \dots, C_c$ according to $\blacktriangleleft$.
 3: for a component $C_t$ do
 4:     Find the ordering $<$.
 5:     Construct linear orderings $\lhd$ for $<$ and its reversal.
 6:     For each ordering, solve one linear program using the previous value $E_{t-1}$.
 7:     if at least one program has a solution then
 8:         Use the solution minimizing the value of $E_t$ of the linear program.
 9:     else
10:         No representation exists, output "no".
11:     end if
12: end for
13: return A representation $\mathcal{R}$ in the ordering $\blacktriangleleft$ satisfying the bounds.
```

A.6 Extending Unit Interval Graphs

The pseudocode is in Algorithm 6. The description is in Section 3.4.

Algorithm 6 Extending Unit Interval Graphs – RepExt(UNIT INT)

Require: A unit interval graph G and a partial representation $\mathcal{R}'$.
Ensure: A representation $\mathcal{R}$ extending $\mathcal{R}'$ if it exists.

```
1: Deal with unlocated components separately, using the standard algorithm.
2: Construct the ordering $\blacktriangleleft$ for located components.
3: Set bounds exactly for each pre-drawn interval $v_i$ at position $\ell_i$, set
       $$\mathrm{lbound}(v_i) = \mathrm{ubound}(v_i) = \ell_i.$$
4: Solve the located components using the bounded representation problem.
5: If no bounded representation exists, output "no".
6: return A representation $\mathcal{R}$ extending $\mathcal{R}'$.
```

A.7 Bounded Representation – Shifting Algorithm

The pseudocode is in Algorithm 7. It is described in Section 3.3.5.

Algorithm 7 Bounded Representation of Unit Interval Graphs – BoundRep

```
Require: A unit interval graph $G$, bound contraints and ordering $\blacktriangleleft$.
Ensure: A representation $\mathcal{R}$ in the ordering $\blacktriangleleft$ satisfying the bounds if it exists.
 1: Constuct a pruned graph, as described in Section 3.3.4.
 2: Compute the value $\varepsilon$ according to the bounds $\frac{p_1}{q_1}, \dots, \frac{p_b}{q_b}$:
        $$\varepsilon' := \frac{1}{\mathrm{lcm}(q_1, q_2, \dots, q_b)}, \qquad \text{and} \qquad \varepsilon := \frac{\varepsilon'}{n^2}.$$
 3: Process the components $C_1, \dots, C_c$ according to $\blacktriangleleft$.
 4: for a component $C_t$ do
 5:     Find the ordering $\lhd$ (which is the same as $<$ before).
 6:     For $\lhd$ and its reversal, construct the left-most representation (with additional
        lower bound $E_{t-1}$).
 7:     Precompute rounded lower bounds.
 8:     Construct an initial representation using [8].
 9:     Proceed the first phase using the LeftShiftprocedure.
10:     Proceed the second phase using the LeftShiftprocedure.
11:     for each LeftShift($v_i$) do
12:         if $\bar{\ell}_i$ is the strict maximum then
13:             Shift $v_i$ to position $\bar{\ell}_i$.
14:         else
15:             The interval $v_i$ becomes fixed.
16:             Compute precise and rounded positions of $v_i$.
17:             Remove $\beta_i$ from the position cycle.
18:         end if
19:     end for
20:     if at least one left-most representation satisfies the upper bounds then
21:         Use the left-most representation minimizing the value of $E_t$.
22:     else
23:         No representation exists, output “no”.
24:     end if
25: end for
26: return A representation $\mathcal{R}$ in the ordering $\blacktriangleleft$ satisfying the bounds.
```

A.8 Subpath-in-Path Graphs – Sub Type Extension

The pseudocode is in Algorithm 8. The description is in Section 4.1.1.

Algorithm 8 Sub Type Extension of (Proper) Subpath-in-Path Graphs

```
Require: A (proper) subpath-in-path graph $G$ and a partial representation $\mathcal{R}'$.
Ensure: A representation $\mathcal{R}$ extending $\mathcal{R}'$ if it exists.
 1: if unlocated components can be placed then
 2:     Place them and ignore them in the rest of the algorithm.
 3: else
 4:     No representation exists, output "no".
 5: end if
 6: if some subpath contains $p_1$ then
 7:     Add $p_0$, $v_\leftarrow$ (and $\bar{p}_0$ for PROPER P-in-P).
 8: end if
 9: if some subpath contains $p_t$ then
10:     Add $p_{t+1}$, $v_\rightarrow$ (and $\bar{p}_{t+1}$ for PROPER P-in-P).
11: end if
12: Apply the algorithm for type BOTH.
13: Deal with vertices created by subdivision of $p_0p_1$ and $p_tp_{t+1}$ if necessary.
14: Modify the representation to construct $\mathcal{R}$.
15: return A representation $\mathcal{R}$ extending $\mathcal{R}'$.
```

A.9 Proper Subpath-in-Path Graphs by Bin Packing

The pseudocode is in Algorithm 9. It is described in Section 4.1.3.

Algorithm 9 Sub Type Extension of Subpath-in-Path Graphs

```
Require: A proper subpath-in-path graph $G$ and a partial representation $\mathcal{R}'$ having
    $k$ pre-drawn intervals.
Ensure: A representation $\mathcal{R}$ extending $\mathcal{R}'$ if it exists.
 1: Test all $2^k$ possible orderings of located components.
 2: for each ordering do
 3:     Place the located components according to the ordering and minimum spans.
 4:     Distribute unlocated components into gaps using BINPACKING.
 5: end for
 6: if no ordering works then
 7:     No representation exists, output "no".
 8: end if
 9: return A representation $\mathcal{R}$ extending $\mathcal{R}'$.
```

B Bibliography

[1] Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM 26(11), 832–843 (1983)

[2] Angelini, P., Battista, G.D., Frati, F., Jelínek, V., Kratochvíl, J., Patrignani, M., Rutter, I.: Testing planarity of partially embedded graphs. In: SODA '10: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (2010)

[3] Benzer, S.: On the topology of the genetic fine structure. Proc. Nat. Acad. Sci. U.S.A. 45, 1607–1620 (1959)

[4] Biró, M., Hujter, M., Tuza, Z.: Precoloring extension. I. interval graphs. Discrete Mathematics 100(1–3), 267–279 (1992)

[5] Bläsius, T., Rutter, I.: Simultaneous PQ-ordering with applications to constrained embedding problems. CoRR abs/1112.0245 (2011)

[6] Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and planarity using pq-tree algorithms. Journal of Computational Systems Science 13, 335–379 (1976)

[7] Corneil, D.G.: A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. Discrete Appl. Math. 138(3), 371–379 (2004)

[8] Corneil, D.G., Kim, H., Natarajan, S., Olariu, S., Sprague, A.P.: Simple linear time recognition of unit interval graphs. Information Processing Letters 55(2), 99–104 (1995)

[9] Corneil, D.G., Olariu, S., Stewart, L.: The LBFS structure and recognition of interval graphs. SIAM Journal on Discrete Mathematics 23(4), 1905–1953 (2009)

[10] Deng, X., Hell, P., Huang, J.: Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. SIAM J. Comput. 25(2), 390–403 (1996)

[11] Fiala, J.: NP completeness of the edge precoloring extension problem on bipartite graphs. J. Graph Theory 43(2), 156–160 (2003)

[12] Fulkerson, D.R., Gross, O.A.: Incidence matrices and interval graphs. Pac. J. Math. 15, 835–855 (1965)

[13] Fürer, M.: Faster integer multiplication. In: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing. pp. 57–66. STOC '07 (2007)

[14] Garey, M.R., Johnson, D.S.: Complexity results for multiprocessor scheduling under resource constraints. SIAM Journal on Computing 4(4), 397–411 (1975)

[15] Gavril, F.: A recognition algorithm for the intersection of graphs of paths in trees. Discrete Mathematics 23, 211–227 (1978)

[16] Golumbic, M.C.: Reasoning about time. In: Mathematical Aspects of Artificial Intelligence, F. Hoffman, ed. vol. 55, pp. 19–53 (1998)

[17] Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. North-Holland Publishing Co. (2004)

[18] Golumbic, M.C., Shamir, R.: Complexity and algorithms for reasoning about time: a graph-theoretic approach. J. ACM 40(5), 1108–1133 (1993)

[19] Hajós, G.: Über eine Art von Graphen. Internationale Mathematische Nachrichten 11, 65 (1957)

[20] Hell, P., Huang, J.: Lexicographic orientation and representation algorithms for comparability graphs, proper circular arc graphs, and proper interval graphs. Journal of Graph Theory 20(3), 361–374 (1995)

[21] Hujter, M., Tuza, Z.: Precoloring extension. II. graph classes related to bipartite graphs. Acta Mathematica Universitatis Comenianae 62(1), 1–11 (1993)

[22] Jampani, K., Lubiw, A.: The simultaneous representation problem for chordal, comparability and permutation graphs. In: Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 5664, pp. 387–398 (2009)

[23] Jampani, K., Lubiw, A.: Simultaneous interval graphs. In: Algorithms and Computation, Lecture Notes in Computer Science, vol. 6506, pp. 206–217 (2010)

[24] Jansen, K., Kratsch, S., Marx, D., Schlotter, I.: Bin packing with fixed number of bins revisited. In: Algorithm Theory - SWAT 2010, Lecture Notes in Computer Science, vol. 6139, pp. 260–272 (2010)

[25] Klavík, P., Kratochvíl, J., Krawczyk, T., Walczak, B.: Extending partial representations of function graphs and permutation graphs. Accepted to ESA 2012, track A (2012)

[26] Klavík, P., Kratochvíl, J., Otachi, Y., Ignaz, R., Saitoh, T., Saumell, M., Vyskočil, T.: Extending partial representations of proper and unit interval graphs. In preparation. (2012)

[27] Klavík, P., Kratochvíl, J., Otachi, Y., Saitoh, T.: Extending partial representations of subclasses of chordal graphs. In preparation. (2012)

[28] Klavík, P., Kratochvíl, J., Otachi, Y., Saitoh, T., Vyskočil, T.: Extending partial representations of interval graphs. Journal version of TAMC paper, in preparation. (2012)

[29] Klavík, P., Kratochvíl, J., Vyskočil, T.: Extending partial representations of interval graphs. In: Theory and Applications of Models of Computation - 8th Annual Conference, TAMC 2011. Lecture Notes in Computer Science, vol. 6648, pp. 276–285 (2011)

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

[31] Kratochvíl, J., Matoušek, J.: String graphs requiring exponential representations. J. Comb. Theory, Ser. B 53(1), 1–4 (1991)

[32] Kuratowski, K.: Sur le problème des courbes gauches en topologie. Fund. Math. 15, 217–283 (1930)

[33] Looges, P.J., Olariu, S.: Optimal greedy algorithms for indifference graphs. Comput. Math. Appl. 25, 15–25 (1993)

[34] Marczewski, E.S.: Sur deux propriétés des classes d'ensembles. Fund. Math. 33, 303–307 (1945)

[35] Marx, D.: NP-completeness of list coloring and precoloring extension on the edges of planar graphs. J. Graph Theory 49(4), 313–324 (2005)

[36] McKee, T.A., McMorris, F.R.: Topics in Intersection Graph Theory. SIAM Monographs on Discrete Mathematics and Applications (1999)

[37] Patrignani, M.: On extending a partial straight-line drawing. In: Lecture Notes in Computer Science. vol. 3843, pp. 380–385 (2006)

[38] Roberts, F.S.: Representations of indifference relations, Ph.D. Thesis. Stanford University (1968)

[39] Roberts, F.S.: Indifference graphs. In: F. Harary (Ed.), Proof Techniques in Graph Theory. pp. 139–146. Academic Press (1969)

[40] Roberts, F.S.: Discrete Mathematical Models, with Applications to Social, Biological, and Environmental Problems. Prentice-Hall, Englewood Cliffs (1976)

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

[42] Schaefer, M., Sedgwick, E., Štefankovič, D.: Recognizing string graphs in NP. In: STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. pp. 1–6 (2002)

[43] Schäffer, A.A.: A faster algorithm to recognize undirected path graphs. Discrete Appl. Math 43, 261–295 (1993)

[44] Spinrad, J.P.: Efficient Graph Representations. Field Institute Monographs (2003)

[45] Stoffers, K.E.: Scheduling of traffic lights–a new approach. Transportation Research 2, 199–234 (1968)

[46] Trotter, W.T.: New perspectives on interval orders and interval graphs. In: in Surveys in Combinatorics. pp. 237–286. Cambridge Univ. Press (1997)

[47] Vilian, M., Kautz, H.: Constraint propagation algorithms for temporal reasoning. In: Proc. Fifth Nat'l. Conf. on Artificial Intelligence. pp. 337–382 (1986)