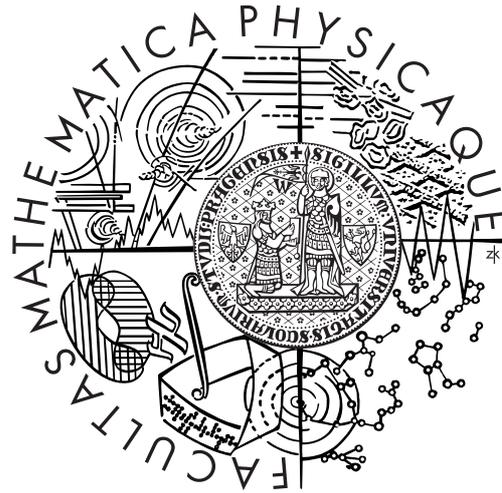


Department of Applied Mathematics
Charles University in Prague
Faculty of Mathematics and Physics



Pavel Klavík

Extending Partial Representations of Interval Graphs

SVOČ 2012, Bratislava

Acknowledgement

The results presented in this thesis are based on:

- Pavel Klavík, Bachelor's thesis, MFF UK, 2010.
- 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.
- Pavel Klavík, Jan Kratochvíl, Yota Otachi, Ignaz Rutter, and Tohsiki Saitoh.
Extending Partial Representations of Proper and Unit Interval Graphs.
In preparation.
- Pavel Klavík, Master's thesis, MFF UK, 2012.
In preparation.

Here, I 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. To describe which results were obtained by myself, I 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 (currently in writing). An origin of the partial representation extension problem is in my bachelor's thesis. We started working with interval graphs and later extended the problem to other graph classes. I would like to thank Pavol Hell for suggesting a PQ-tree approach.

Then based on my bachelor's thesis, we made a paper with a significant improvement of writing (based on many suggestions of coauthors) which got accepted for TAMC 2011 conference in Tokyo. After my talk there, Yota Otachi and Toshiki Saitoh contacted us with a suggestion how to extend proper intervals in linear time, based on a paper of Deng, Hell, and Huang. This later lead me to finding a polynomial time algorithm (based on LP) for extending unit interval graphs; thus solving 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 separately solved by Ignaz Rutter. During GD 2011, I discussed together with Ignaz and Jan our approach and possible faster combinatorial algorithms avoiding LP (which are not presented here and still unfinished), and thus we decided to write a future paper together.

Contents

1	Introduction	5
2	Extending Interval Graphs	9
2.1	PQ-trees and Interval Orders	9
2.2	Extending Interval Graphs	15
3	Extending Proper and Unit Interval Graphs	20
3.1	Extending Proper Interval Graphs	20
3.2	Extending Unit Interval Graphs	23
4	Conclusions	27
4.1	Simultaneous Representations	27
4.2	Allen Algebras	28
4.3	Open Problems	29
A	Pseudocodes of Algorithms	31
A.1	Reordering PQ-tree, general partial order	31
A.2	Reordering PQ-tree, interval order	32
A.3	Extending Interval Graphs	32
A.4	Extending Proper Interval Graphs	33
A.5	Extending Unit Interval Graphs	34
	Bibliography	35

Abstract

Representations of graphs are as old as graph theory. For a fixed representation, a natural question of recognition asks whether an input graph can be represented using this representation. Complexity of recognition for many types of representations is well studied and deeply understood.

In this thesis, we consider *interval graphs*. An interval graph has a representation which assigns intervals of the real line to the vertices such that two intervals intersect each other if and only if the corresponding vertices are adjacent. Also, we consider two famous subclasses of interval graphs: *proper interval graphs* (no interval is a proper subset of another interval) and *unit interval graphs* (all intervals have the unit length). All three classes can be recognized in time $\mathcal{O}(n + m)$.

We study the problem of partial representation extension where a part of the representation is fixed by the input and it asks whether it is possible to extend this partial representation to a representation of the whole input graph. In the case of interval graphs, some of the intervals are already pre-drawn by the input and we ask whether it is possible to add the rest of the intervals to this drawing. This problem is at least as hard as recognition.

We show that partial representation extension is solvable for all three classes in polynomial time. For interval graphs and proper interval graphs, we give linear-time algorithms in time $\mathcal{O}(n + m)$. For unit interval graphs, we give a polynomial time algorithm based on linear programming. We note that although proper and unit interval graphs are known to be equal, the partial representation extension problem distinguishes them. In the case of unit interval graphs, additional problems of computing with rational numbers are introduced.

Chapter 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 (and thus giving geometrical names to vertices, edges and faces of graphs). The theorem of Kuratowski [Kur30] provides the first combinatorial characterization of planar graphs and can be considered as the start of modern graph theory.

Intersection Representations. In this paper, we are interested in intersection representations which assign geometric objects to vertices of graphs and which encode edges by intersections of the 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 [Mar45], to obtain interesting classes of graphs we restrict sets R_v , for example to some specific geometric objects. Classical examples include interval graphs, circle graphs, permutation graphs, string graphs, convex graphs, and function graphs [Gol04, Spi03]. As seen from these two monographs, geometric intersection graphs are intensively studied for their applied motivations, algorithmic properties and many interesting theoretical results.

Naturally, the recognition problems for these classes are deeply studied and well understood. It asks whether an input graph has a representation \mathcal{R} belonging to a specific class. 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 known. For example, interval graphs can be recognized in linear time [BL76, COS09], while recognition of string graphs is **NP**-complete [Kra91, SSS02] (we note that belonging to **NP** is non-trivial since there are examples of string graphs requiring exponential number of crossing points in every representation [KM91]). 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 Representations. We study a question of partial representation extension for intersection-defined classes of graphs. A *partial representation* of G is a representation \mathcal{R}' of an induced subgraph G' . A representation \mathcal{R} of the whole graph G extends the

partial representation \mathcal{R}' if $R'_v = R_v$ for every $v \in V(G')$. The partial representation extension is the following decision problem:

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

In this thesis we investigate the complexity of partial representation extension for interval graphs and its two subclasses proper interval graphs and unit interval graphs; which we describe later. Now, we mention results related to other classes. A paper [KKKW12] studies partial representation extension of permutation and function graphs, and it shows that both classes are extendible in polynomial time and that the problem becomes **NP**-complete if the partial representation can specify the functions only partially. A paper [KKOS] studies chordal graphs and shows that the problem becomes **NP**-complete in almost every possible setting which is non-trivial.

A related problem of simultaneous graph representations was recently introduced by Jampani and Lubiw [JL10]: Given a set of graphs G_1, \dots, G_k with common vertices I (meaning for every $i \neq j$ is $V(G_i) \cap V(G_j) = I$), do there exist representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ such that \mathcal{R}^i represents G_i and the representations equal on I (meaning for every i, j and $v \in I$ is $R_v^i = R_v^j$)? Simultaneous representations are closely related to partial representations and we discuss the connection in Conclusions.

Several other problems have been considered in which a partial solution is given and the task is to extend it. For example, every k -regular bipartite graph is k -edge-colorable. But if some edges are pre-colored, the extension problem becomes **NP**-complete even for $k = 3$ [Fia03], and even when the input is restricted to planar graphs [Mar05]. For planar graphs, partial representation extension is solvable in linear time [ABF⁺10]. Every planar graph admits a straight-line drawing, but extending such representations is **NP**-complete [Pat06].

Interval Graphs. *Interval graphs* are one of the oldest and most studied classes of graphs, introduced by Hajos [Haj57]. A graph is an interval graph if it has a representation \mathcal{R} by closed intervals of the real line, i.e., each vertex v is represented by an interval R_v in such a way that $R_u \cap R_v \neq \emptyset$ if and only if $uv \in E(G)$. Interval graphs have many useful theoretical properties, for example they are perfect and related to path-width decomposition. Many hard combinatorial problems are polynomially solvable for interval graphs: maximum clique, k -coloring, maximum independent set, e.g. Interval graphs naturally appear in many applications concerning biology, psychology, time scheduling, archeology; see for example [Rob76, Sto68, Ben59]. We denote the class of interval graphs by **INT**.

The class of interval graphs has two famous subclasses. The first subclass *proper interval graphs* (denoted by **PROPER INT**) has representations in which no interval is a proper subset of another interval (for each $u, v \in V(G)$ we have $R_u = R_v$ or both $R_u \setminus R_v$ and $R_v \setminus R_u$ are non-empty). *Unit interval graphs* (denoted by **UNIT INT**) have representations with all the intervals of the unit length. Several linear-time recognition

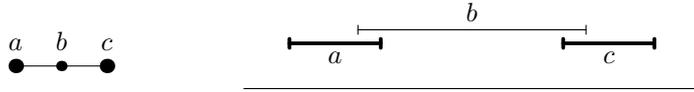


Figure 1.1: A partial representation of a path P_2 representing end-points far apart is extensible only by proper intervals.

algorithms are known for these classes [LO93, HH95, CKN⁺95, Cor04]. Also solving of many problems is even easier in the case of these classes.

A famous result of Roberts [Rob69] states that these two classes are equal:

$$\text{PROPER INT} = \text{UNIT INT}.$$

It is easy to see that $\text{UNIT INT} \subseteq \text{PROPER INT}$. To obtain the other inclusion, we scale and shift intervals. Instead of unit interval graphs, most of the papers concentrates on proper interval graphs which have an easier combinatorial structure. This is not the case of partial representation extension which distinguishes these classes and puts additional conditions in the case of unit interval graphs; see Figure 1.1.

Our Results. We study the partial representation extension problem for classes INT , PROPER INT and UNIT INT , and show that they are solvable in polynomial time.

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

We describe an algorithm for $\text{REPEXT}(\text{INT})$ based on a PQ-tree approach for their recognition [BL76]. We note that another linear-time algorithm was obtained independently by Bläsius and Rutter [BR11] but their algorithm is solving a more general problem and it is quite involved.

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

Theorem 1.3. *The problem $\text{REPEXT}(\text{UNIT INT})$ is solvable in polynomial time.*

The algorithm for $\text{REPEXT}(\text{PROPER INT})$ is based on a recognition algorithm of Corneil et al. [CKN⁺95]. We get a unique ordering of vertices of the graph and try to match this unique ordering on the partial representation. In the case of unit interval graphs, there are additional problems concerning positions which we solve by linear programming.

We note that partial representation extension problems for these classes can be stated in a more general settings of Allen algebras. But in this setting, the resulting more general problems are known to be **NP**-complete. For more details, see Conclusions.

Remarks Concerning Input. To obtain linear-time algorithms of theorems 1.1 and 1.2, we need some reasonable assumption on a partial representation which is given by the

input. Similarly, most of the graph algorithm can not 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.

For partial representations of interval graphs, we assume that its intervals are sorted from left to right. Alternatively, in the case of **INT** and **PROPER INT**, we may assume that intervals are in a *normalized form*, with endpoints in the whole number between 0 and $2k$ (where k is the number of pre-drawn intervals). This assumption makes sense since the solutions of **REPEXT** for these classes depends only on topology of the partial representation, not precise positions of endpoints.

The sorted intervals can be easily transformed into the normalized form while maintaining topology of the partial representation. Also, from a normalized representation, we can easily produce a sorted representation. If the partial representation would be given in unsorted, we would need additional time $\mathcal{O}(k \log k)$ to sort it first.

Structure. The thesis is structured as follows. In Chapter 2, we solve the partial representation extension for interval graphs. In Chapter 3, we solve **REPEXT(PROPER INT)** and **REPEXT(UNIT INT)**. We note that these chapters are completely independent and can be read separately. In Conclusions, we show related problems of simultaneous representations and Allen algebras, and give some open problems.

Notation. For a graph G , we denote its vertices by $V(G)$ and its edges by $E(G)$. Let

$$\mathcal{R} = \{R_v : v \in V(G)\}$$

be an interval representation of a graph G , meaning $R_u \cap R_v \neq \emptyset$ if and only if $uv \in E(G)$. By ℓ_v and r_v , we denote the left-endpoint and the right-endpoint of the interval R_v .

For a vertex v , an *open neighborhood* $N(v)$ is $\{u : uv \in E(G)\}$ and a *closed neighborhood* $N[v]$ is $N(v) \cup \{v\}$.

Chapter 2

Extending Interval Graphs

In this chapter, we solve the problem $\text{REPEXT}(\text{INT})$ 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 $\text{REPEXT}(\text{INT})$. 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.

Consecutive ordering. The input of the *consecutive ordering problem* is a set E of elements and restricting sets S_1, S_2, \dots, 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 $\underline{a}cde\underline{f}gbh$ (violates S_1) and $de\underline{f}hgabc$ (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

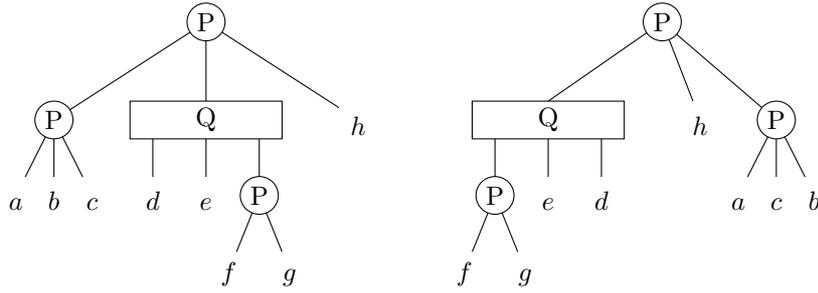


Figure 2.1: PQ-trees representing orderings $abcdefgh$ and $fgedhacb$.

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 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 [BL76] describe details of their construction.

Applying a Partial Order. Suppose that we have a PQ-tree T and a partial ordering \blacktriangleleft of its elements (leaves). We ask whether it is possible to reorder the PQ-tree T to a PQ-tree T' in such a way that an ordering $<_{T'}$ extends \blacktriangleleft , meaning $a \blacktriangleleft b$ implies $a <_{T'} b$.

Problem: REORDER(T, \blacktriangleleft)
Input: A PQ-tree T and a partial ordering \blacktriangleleft .
Output: Is it possible to reorder T to T' such that $<_{T'}$ extends \blacktriangleleft .

The algorithm we are going to describe works even for a general relation \blacktriangleleft . For example, \blacktriangleleft 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, \blacktriangleleft) 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 \blacktriangleleft .*

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.

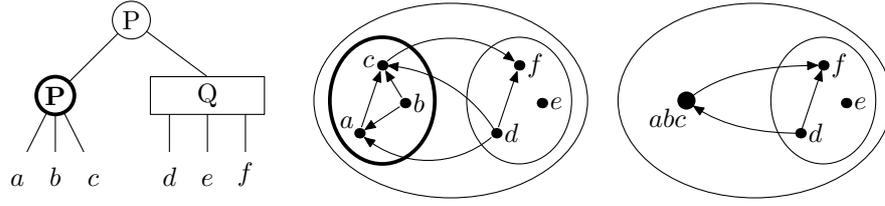


Figure 2.2: We show an example how reordering works. Consider the highlighted P-node of the PQ-tree on the left. The subdigraph induced by a, b and c has a topological sorting $b \rightarrow a \rightarrow c$. We contract these vertices into the vertex abc . Next, we keep the order of Q-node and contract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

Lemma 2.3. *Let S be a subtree of a PQ-tree T . If T can be reordered according to \blacktriangleleft then every local reordering of the subtree S according to \blacktriangleleft can be extended to a reordering of the whole tree T according to \blacktriangleleft .*

Proof. Let $<$ be an ordering obtained by reordering of the whole PQ-tree T according to \blacktriangleleft , so $< = <_{T'}$ for some reordering T' . Notice that all elements of S appear consecutively in $<$. Therefore, we can replace this local order of S by any other order satisfying all conditions given by \blacktriangleleft , and thus we obtain another correct reordering of the whole tree T . \square

This gives the following algorithm. We represent the ordering \blacktriangleleft 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 \blacktriangleleft . If no such reordering of S can be found, it is not possible to order the whole tree according to \blacktriangleleft . 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 \blacktriangleleft which is $\mathcal{O}(e + m)$. Each edge of the digraph \blacktriangleleft is

processed exactly once and then it is contracted. \square

Interval Orders. 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 \blacktriangleleft . For $u, v \in V(H)$, we define $u \blacktriangleleft v$ if and only if $r_u \leq \ell_v$. For example, in Figure 1.1 we obtain an interval order in which $a \blacktriangleleft c$ 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 $a \blacktriangleleft b$, we always know 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 [Tro97].

Let \blacktriangleleft be an interval order of e elements, with a normalized representation having all endpoints in whole numbers of $[0, 2e]$. 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 \blacktriangleleft , we show that we can solve `REORDER(T, \blacktriangleleft)` faster:

Proposition 2.4. *If \blacktriangleleft is an interval order with a normalized representation, we can solve the problem `REORDER(T, \blacktriangleleft)` 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 these steps faster than before.

We are not going to construct a digraph explicitly and therefore we are not doing any contractions. Instead, we are going to work with sets of intervals and compare them in \blacktriangleleft fast. When we process a node, its children correspond 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 say $\mathcal{I}_1 \blacktriangleleft \mathcal{I}_2$ if there exist $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$ such that $a \blacktriangleleft b$. Using the interval representation, we can test

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

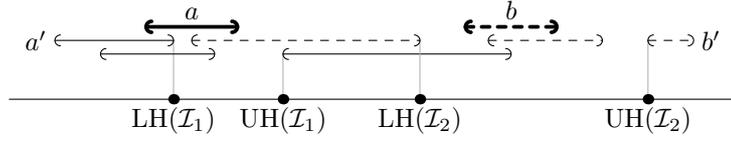


Figure 2.3: Normal intervals belong to \mathcal{I}_1 , dashed intervals belong to \mathcal{I}_2 . If $a \blacktriangleleft b$, then also $a' \blacktriangleleft b'$.

whether $\mathcal{I}_1 \blacktriangleleft \mathcal{I}_2$ 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 $a \blacktriangleleft b$, $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$. Then for every $a' \in \mathcal{I}_1, r_{a'} \leq r_a$ and every $b' \in \mathcal{I}_2, \ell_b \leq \ell_{b'}$ also $a' \blacktriangleleft b'$.*

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

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 $\mathcal{I}_1 \blacktriangleleft \mathcal{I}_2$ if and only if $a' \blacktriangleleft b'$. To simplify the description, we define handles of \mathcal{I} , a *lower handle* and *upper handle*:

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

Notice that $\text{LH}(\mathcal{I}) < \text{UH}(\mathcal{I})$ if and only if \mathcal{I} is not a clique. Using handles, we can compare sets of intervals fast: $\mathcal{I}_1 \blacktriangleleft \mathcal{I}_2$ if and only if $\text{LH}(\mathcal{I}_1) \leq \text{UH}(\mathcal{I}_2)$. 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 our equivalent 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 $\mathcal{I}_i \blacktriangleleft \mathcal{I}_j$ implies $\mathcal{I}_i < \mathcal{I}_j$ for every $i \neq j$. 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 both the lower and the upper handles of $\mathcal{I}_1, \dots, \mathcal{I}_k$ from left to right in time $\mathcal{O}(k)$. The handles can share coordinates, we first 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, $\mathcal{I}_i \blacktriangleleft \mathcal{I}_j$ if $\text{LH}(\mathcal{I}_i)$ is before $\text{UH}(\mathcal{I}_j)$ in the ordering. For an example, see Figure 2.4.

To find a topological sorting, we identify in each step a minimal element, remove its handles from the order and append the minimal element to the constructed topological sorting. 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 in some topological sorting if and only if there is no lower handle except maybe $\text{LH}(\mathcal{I}_j)$ on the left of $\text{UH}(\mathcal{I}_j)$.*

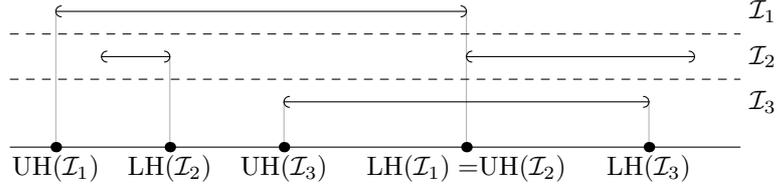


Figure 2.4: For sets \mathcal{I}_1 , \mathcal{I}_2 and \mathcal{I}_3 , we get a common order of handles $\text{UH}(\mathcal{I}_1) \leq \text{LH}(\mathcal{I}_2) \leq \text{UH}(\mathcal{I}_3) \leq \text{LH}(\mathcal{I}_1) \leq \text{UH}(\mathcal{I}_2) \leq \text{LH}(\mathcal{I}_3)$. Notice that \blacktriangleleft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

Proof. Notice that $\mathcal{I}_i \blacktriangleleft \mathcal{I}_j$ if and only if $\text{LH}(\mathcal{I}_i) \leq \text{UH}(\mathcal{I}_j)$. There is no such \mathcal{I}_i if and only if there is no $\text{LH}(\mathcal{I}_i)$ on the left of $\text{UH}(\mathcal{I}_j)$. \square

So, for example, if the ordering starts with two lower handles, there exists no topological sorting. If the first element of the ordering is $\text{LH}(\mathcal{I}_i)$ then \mathcal{I}_i has to be the unique minimal element. We just need to check whether there is some other $\text{LH}(\mathcal{I}_j)$ before $\text{UH}(\mathcal{I}_i)$, and if so, there is no minimal element and the topological sorting does not exist.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 $\text{LH}(\mathcal{I}_j)$; \mathcal{I}_j is a minimal element if there is no other lower handle on the left of $\text{UH}(\mathcal{I}_j)$.

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

$$\text{LH}(\mathcal{I}) = \min\{\text{LH}(\mathcal{I}_i)\} \quad \text{and} \quad \text{UH}(\mathcal{I}) = \max\{\text{UH}(\mathcal{I}_i)\}.$$

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 \blacktriangleleft 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 \blacktriangleleft

²This 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.

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 from minimal elements if it exists, using 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 $\mathcal{O}(e)$, the number of the elements. \square

2.2 Extending Interval Graphs

In this section, we describe an algorithm solving $\text{REPEXT}(\text{INT})$ in time $\mathcal{O}(n + m)$. Interval representations have closed intervals. Intervals may share the endpoints and may have zero lengths but this is not very key in the functioning of the algorithm.

We first describe recognition of interval graphs. Then we show how to modify the PQ-tree approach to solve $\text{REPEXT}(\text{INT})$.

Recognition. Recognition of interval graphs in linear time was a long-standing open problem, first solved by Booth and Lueker [BL76] 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 [COS09].

We modify the PQ-tree approach to solve $\text{REPEXT}(\text{INT})$ 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 [FG65]:

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.*

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 (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.

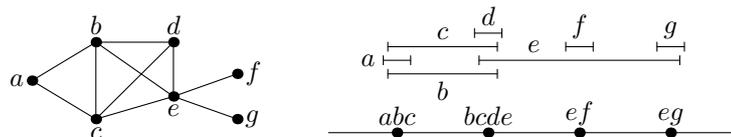


Figure 2.5: An interval graph and one of its representations with denoted clique-points.

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 $\text{cp}(a)$.

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.

In the rest of the section, by a clique we mean a maximal clique. The cliques of an interval graph have in total $\mathcal{O}(n+m)$ vertices and can be found in linear time [RTL76]. A feasible ordering of the 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 cliques containing this vertex. Using PQ-trees, we can find a feasible ordering of maximal cliques and recognize an interval graph in time $\mathcal{O}(n+m)$.

Extending INT. 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 \blacktriangleleft of the cliques. Using the algorithm described in Section 2.1, we try to reorder the PQ-tree according to \blacktriangleleft in time $\mathcal{O}(n+m)$. We will 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 \blacktriangleleft . 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 $\text{cp}(a)$ can be placed only to a part containing exactly the intervals of $I(a)$ and no other pre-drawn intervals.

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

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

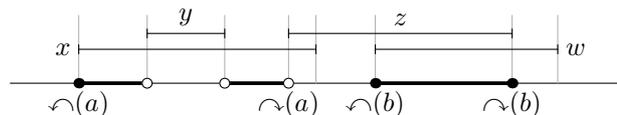


Figure 2.6: Clique-points $\text{cp}(a)$ and $\text{cp}(b)$, having $I(a) = \{x\}$ and $I(b) = \{z, w\}$, can be placed to the bold parts of the real lines.

For an example, see Figure 2.6. Notice that it does not mean that the clique-point $\text{cp}(a)$ can be placed to all the points between $\lrcorner(a)$ and $\rceil(a)$. 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 $a \blacktriangleleft b$ if $\lrcorner(a) \leq \lrcorner(b)$. It is quite natural since $a \blacktriangleleft b$ implies that every correct representation has to place $\text{cp}(a)$ to the left of $\text{cp}(b)$. For example, the cliques a and b in Figure 2.6 satisfy $a \blacktriangleleft b$.

Lemma 2.8. *The relation \blacktriangleleft is an interval order.*

Proof. The intervals of the interval order \blacktriangleleft correspond to the cliques of G . To a clique a , we assign an open interval $I_a = (\lrcorner(a), \rceil(a))$. The definition of \blacktriangleleft exactly states that $a \blacktriangleleft b$ 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 $\text{REPEXT}(\text{INT})$, 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 \lrcorner and \rceil for all the cliques and obtain a normalized interval representation of an interval order \blacktriangleleft .
3. Reorder the PQ-tree according to the interval order \blacktriangleleft , using Section 2.1.
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 \blacktriangleleft . In Step 3, we apply the algorithm described in Section 2.1. 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 $\text{cp}(a)$. Let $\text{cp}(b)$ be the last placed clique-point. Consider the infimum over all the points where the clique-point $\text{cp}(a)$ can be placed and that are to the right of the clique-point $\text{cp}(b)$. If there is a single such point on the right of $\text{cp}(b)$ (equal to the infimum), we place $\text{cp}(a)$ there. Otherwise $\lrcorner(a) < \rceil(a)$ and we place the clique-point $\text{cp}(a)$ to the right of this infimum by an appropriate epsilon, for example the length of the shortest part (see definition of \blacktriangleleft) divided by n . We can easily implement this greedy subroutine in time $\mathcal{O}(n)$.

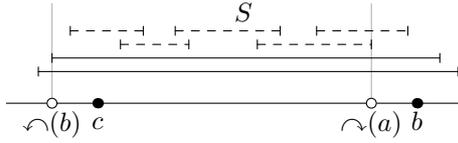


Figure 2.7: An illustration of the proof: The positions of the clique-points b and c , the intervals of S are dashed.

The following lemma states that this greedy procedure can not fail.

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

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

The clique-point $\text{cp}(b)$ 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 $\curvearrowright(b)$ (otherwise we could place $\text{cp}(b)$ to $\curvearrowright(b)$ or right next to it). Let $\text{cp}(c)$ be the right-most clique-point placed between $\curvearrowright(b)$ and $\text{cp}(b)$. Every point between $\text{cp}(c)$ 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.

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

Proof of Theorem 1.1. The conditions given by the interval order \blacktriangleleft on the order of the clique-points are clearly necessary. If it is not possible to reorder the PQ-tree according to \blacktriangleleft , the structure of the interval graph is inconsistent with the partial representation and $\text{REPEXT}(\text{INT})$ is not solvable. On the other hand, if the PQ-tree can be reordered according to \blacktriangleleft , 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 \blacktriangleleft in time $\mathcal{O}(n+m)$. Finally, the representation can be constructed in time $\mathcal{O}(n+m)$. \square

Chapter 3

Extending Proper and Unit Interval Graphs

In this chapter, we describe how to extend partial representations of proper and unit interval graphs. In the case of proper interval graphs, the algorithm is based on (somewhat) uniqueness of representation of a proper interval graph. Also, there is some additional work we have to do for each component of connectivity.

For unit interval graphs, the extension algorithm is based on the algorithm for proper interval graphs. But the partial representations introduce another restrictions concerning precise rational number positions which we solve using linear programming.

3.1 Extending Proper Interval Graphs

First, we introduce some properties of proper interval graphs and describe recognition algorithm of Corneil et al. [CKN⁺95].

Overview. For a representation of a general interval graphs, the order of the left endpoints can be completely different from the order of the right endpoints. In the case of proper interval graphs, the orders of the left endpoints and the right endpoints are always the same. This makes their recognition much simpler. Roberts [Rob68] gave the following characterization:

Lemma 3.1 (Roberts). *A graph is a proper interval graph if there exists a linear ordering $<$ of the vertices having every closed neighborhood consecutive.*

How many different orderings may a graph admit? Possibly many but we are going to show that all of them have a very simple structure. Vertices u and v are called *indistinguishable* if $N[u] = N[v]$. Notice that indistinguishable vertices may be swapped in the ordering and they may be represented by completely same intervals (this is actually true in general for intersection representations). In the graph, we have some *groups* of indistinguishable vertices. They always appear consecutively in the ordering and their order is not particularly important.

Deng et al. [DHH96] proved the following:

Lemma 3.2 (Deng et al.). *For a connected proper interval graph, the ordering $<$ 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.

Recognition. We sketch a simple 2-sweep linear-time recognition algorithm of Corneil et al. [CKN⁺95] which we later modify. Suppose that the graph is connected, otherwise we work with each component separately. A vertex v is called *left-anchor* if it appears in some ordering $<$ as the left-most vertex. The algorithm runs BFS (breadth-first search) twice. The first BFS starts in an arbitrary vertex u and locates some left-anchor v . The second BFS runs from v and constructs an ordering $<$ of Lemma 3.1 if the input graph is a proper interval graph.

Let L_k denote the set of vertices of the k th level of the second BFS. The ordering is given by the second BFS in the following way:

1. The ordering is primarily given by the levels; $u \in L_k, v \in L_{k+1}$ gives $u < v$.
2. Inside each level, additional order is given by number of neighbors in neighboring levels. Vertices of L_k are ordered in increasing order of

$$v \in L_k : |N(v) \cap L_{k+1}| - |N(v) \cap L_{k-1}|.$$

Vertices incomparable by both conditions form groups of indistinguishable vertices and can be ordered arbitrarily. Corneil et al. [CKN⁺95] prove that the obtained ordering $<$ satisfies Lemma 3.1 if and only if the input graph is a proper interval graph. For an example, see Figure 3.1.

To obtain the unique ordering of Lemma 3.2, we need to do a simple modification. It holds that two vertices are indistinguishable if and only if they are not compared by $<$ with one exception: The left anchor v is always the minimal element of the ordering $<$. We only need to check whether the left-anchor belongs to the following group of indistinguishable vertices, and if so, we join it with this group. For example from Figure 3.1, the left anchor v_1 is indistinguishable with v_2 and v_3 , and thus we modify the ordering to

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

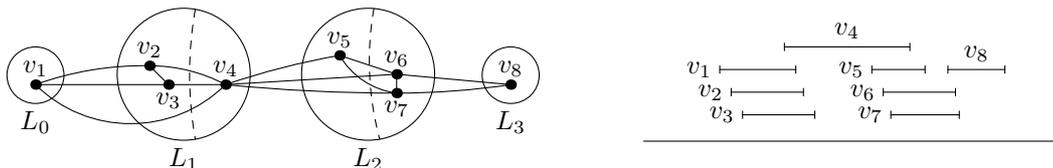


Figure 3.1: Ordering constructed by the second BFS from a left-anchor v_1 is the following: $v_1 < (v_2, v_3) < v_4 < v_5 < (v_6, v_7) < v_8$ (indistinguishable groups in brackets). The corresponding interval representation is on the right.

Extending. If an input graph is not connected, we calculate for every pre-drawn interval to which component it belongs. If two pre-drawn intervals of one component are split by a pre-drawn interval of another component, the partial representation is not extendible. Otherwise, we can work with components separately since the gaps between components are large enough to construct any proper interval graph there; notice that this does not hold for unit interval graphs.

Now, for each component, we calculate a partial order $<$ of its intervals as described before (with no order on groups of indistinguishable vertices). The pre-drawn intervals are in some fixed order. The component is extendible if and only if this order agrees with the partial order $<$ (or its reversal). If so, the partial representation gives some additional order for groups of indistinguishable vertices. To construct a representation, we construct any topological sorting and obtain a linear ordering \triangleleft .

Constructing Representation. We describe how to construct an exact representation. First, we reserve for each component some *area* in which we construct its representation. It contains every pre-drawn interval and some space around.

Components which contain at least one pre-drawn interval are called *located*. Located components are in some order from left to right. The neighboring located components have some gaps between their pre-drawn intervals. We split this gap in half and assign it to these components. We assign some additional space to the leftmost and the rightmost component. Finally, we place non-located components to the right of last located component (and assign area of some length to them). For an example, see Figure 3.2.

We draw each individual component by the following procedure. From \triangleleft , we can easily calculate the common order of left and right endpoints. We start with the order of the left endpoints, which is exactly given by \triangleleft . To this order, we can mix the right endpoints since we know how many neighbors each interval has on the right: If an interval v_i has r right neighbors v_{i+1}, \dots, v_{i+r} , then r_i is placed right after ℓ_{i+r} .

The area reserved for the component is split to several parts by endpoints of pre-drawn intervals. To each part, we place right amount of points equidistantly. For an example, see Figure 3.3.

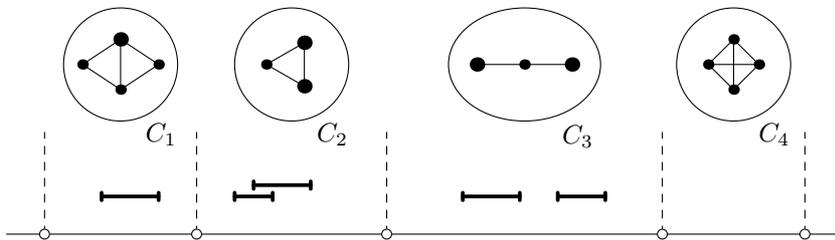


Figure 3.2: An example of a graph with four components C_1, \dots, C_4 . The pre-drawn intervals give order of the located components: $C_1 < C_2 < C_3$. The non-located component C_4 is placed to the right. For each component, we reserve some area in which we construct a representation.

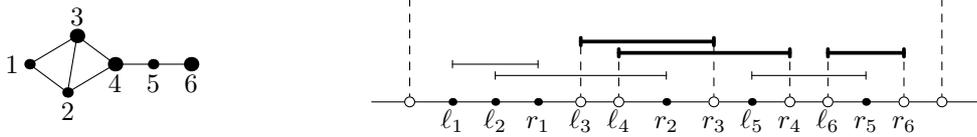


Figure 3.3: For a component with order $1 \triangleleft 2 \triangleleft 3 \triangleleft 4 \triangleleft 5 \triangleleft 6$, we construct a representation in its reserved area. First, we compute the common order of the left and the right endpoints: $l_1 < l_2 < r_1 < l_3 < l_4 < r_2 < r_3 < l_5 < r_4 < l_6 < r_5 < r_6$. The endpoints of the pre-drawn intervals split the area to several parts. We place the remaining endpoints in this order equidistantly to these parts.

A pseudocode of the algorithm is in Section A.4. Now we are ready to prove that $\text{REPEXT}(\text{PROPER INT})$ can be solved in time $\mathcal{O}(n + m)$:

Proof of Theorem 1.2. Clearly, when the algorithm fails in some step, the partial representation is not extendible. Otherwise, we obtain an ordering of endpoints according to \triangleleft such that pre-drawn intervals are at correct places. We obtain a correct representation of the graph extending the partial representation since the ordering of the endpoint satisfies Lemma 3.1.

Concerning time complexity, \triangleleft can be constructed in time $\mathcal{O}(n + m)$ if the input partial representation is given sorted/normalized. Using \triangleleft , we can construct a representation of the whole G in time $\mathcal{O}(n + m)$. \square

3.2 Extending Unit Interval Graphs

Since every unit interval graph is a proper interval graph [Rob69], the algorithm is based on algorithm described in Section 3.1. Partial representation extension of unit interval graphs is harder since partial representations pose additional geometrical restrictions concerning lengths and positions. We describe how to solve these restrictions using linear programming.

We note that for most of the graph algorithms, it is very easy to deal with disconnected graphs by applying the same algorithm for each component separately. In the case of unit interval graph extension, components restrict each other space and it is non-trivial to deal with them. For an example, see Figure 3.4.

Approach. In the rest of the algorithm, we care only about located components (with at least one interval in the partial representation). Unlocated components can be placed far to the right using a standard unit interval graph recognition algorithm. The located components are ordered by the partial representation from left to right.

According to Lemma 3.2, each located component can be represented in (at most) two different ways. Unlike proper interval graphs, we cannot choose one of them arbitrarily, since neighboring components restrict each other's space. For example, a component C_1 in Figure 3.4 can be represented only in one way to allow extension

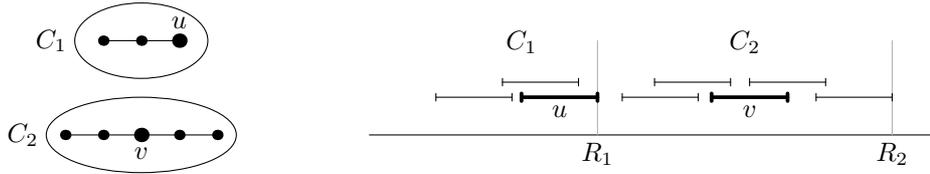


Figure 3.4: The component C_1 can be represented only in the single way. Its reversal would block space for the component C_2 .

of C_2 .

We can process located components $C_1 < C_2 < \dots < C_c$ from left to right and try to represent each component as far to the left as possible. For each component, we calculate two orderings of its vertices $v_1 \triangleleft v_2 \triangleleft \dots \triangleleft v_k$ as described in Section 3.1 (one for $<$ and one for its reversal). We use greedily the ordering which allows us to push the component further to the left, leaving more space for the remaining components.

Linear Programs. We denote the right-most endpoint of a component C_i by R_i , and let $R_0 = -\infty$. For each component C_t we solve (at most) two linear programs, one for each possible ordering. For each ordering, we minimize value of R_t and we use the ordering with a smaller value of R_t .

Let ε be a small length (which we describe later) such that each pair of non-intersecting intervals can be represented in distance at least ε . For ordering $v_1 \triangleleft v_2 \triangleleft \dots \triangleleft v_k$ of vertices of C_t , we solve the following linear program (with variables ℓ_1, \dots, ℓ_k):

$$\begin{aligned} \text{Minimize:} \quad & R_t := \ell_k + 1, \\ \text{subject to:} \quad & R_{t-1} + \varepsilon \leq \ell_1, \end{aligned} \tag{3.1}$$

$$\ell_i \leq \ell_{i+1}, \quad \forall i = 1, \dots, k-1, \tag{3.2}$$

$$\ell_i = \text{a constant}, \quad \text{if } v_i \text{ is pre-drawn}, \tag{3.3}$$

$$\ell_i \geq \ell_j - 1, \quad \forall v_i v_j \in E, v_i \triangleleft v_j, \tag{3.4}$$

$$\ell_i + \varepsilon \leq \ell_j - 1, \quad \forall v_i v_j \notin E, v_i \triangleleft v_j. \tag{3.5}$$

Constraint (3.1) is a barrier created by component C_{t-1} . Constraints (3.2) force the ordering of intervals. Constraints (3.3) force a representation to extend the partial representation. Constraints (3.4) and (3.5) prescribe edges and non-edges of the graph.

We note that it is possible to reduce the number of constraints from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Using the ordering constraints (3.2), we can replace constraints (3.4) and (3.5) by just two constraints for each vertex. Let v_i be non-adjacent to v_j but adjacent to v_{j+1} (and $v_j \triangleleft v_i$), so v_j is the rightmost non-neighbor of v_i on the left. Then we just need to state the constraint (3.4) for v_{j+1} and v_i and the constraint (3.5) for v_j and v_i .

Value of Epsilon. For a given partial representation, we consider how small a grid is required such that all intervals can have endpoints on this grid. In other words, the value of epsilon prescribes the required resolution of the drawing. In [CKN⁺95], it

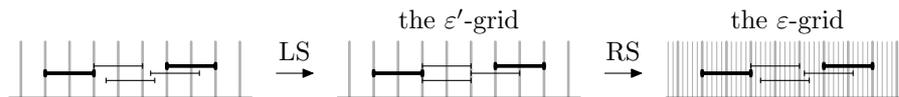


Figure 3.5: In the first step, we shift intervals to the left to the ε' -grid. We have left shifts of v_1, \dots, v_5 equal $(0, 0, \frac{1}{2}\varepsilon', \frac{1}{3}\varepsilon', 0)$. In the second step, we shift to the right in the refined ε -grid. Right shifts have the same relative order as left shifts: $(0, 0, 2\varepsilon, \varepsilon, 0)$.

is described that for the empty partial representation (i.e., just recognition of unit interval graphs) a grid of size $\frac{1}{n}$ is sufficient.

Let ε' be a size of a grid which contains all endpoints of the pre-drawn intervals. Formally, let the partial representation have the left endpoint ℓ_i at position $\frac{p_i}{q_i}$. Then

$$\varepsilon' := \frac{1}{\text{lcm}(q_1, q_2, \dots, q_r)}, \quad \text{and} \quad \varepsilon := \frac{\varepsilon'}{n}. \quad (3.6)$$

We show that this ε -grid is sufficient to extend the partial representation.

Lemma 3.3. *For a representation extending a partial representation of a unit interval graph, there exists another representation having endpoints on the ε -grid where ε is defined by (3.6).*

Proof. We construct an ε -grid representation in two steps. First, we shift intervals to the left; for an interval v_i , the size of the left shift is denoted by $\text{LS}(v_i)$. Then, we shift intervals slightly back to the right; the size of the right shift is denoted by $\text{RS}(v_i)$. The shifting process is shown in Figure 3.5.

In the first step, we consider the ε' -grid and shift all the intervals to the left to the closest grid-point. So $\text{LS}(v_i) < \varepsilon'$ for all intervals v_i . Notice that the pre-drawn intervals are not shifted since the ε' -grid contains their endpoints; the ε' -grid was constructed in this way. Original intersections are kept by this shifting. On the other hand, we may introduce additional intersections by shifting two non-intersecting intervals to each other (but they can only touch), for example v_2 and v_4 in Figure 3.5.

The second step shifts the intervals to the right in the refined ε -grid to remove these additional intersections. We want to find a mapping

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

such that for all pairs (v_i, v_j) having $r_i = \ell_j$ is $\text{RS}(v_i) \geq \text{RS}(v_j)$ if and only if $v_i v_j \in E$.

We first argue that this right shifting will produce a correct ε -grid representation. First, it does not create additional intersections; non-intersecting pairs of intervals are in distance at least $\varepsilon' = n\varepsilon$ and we shift at most $(n-1)\varepsilon$. Also, if two intervals overlap by at least ε' , their intersection is not removed. The only intersections which are modified are touching pairs of intervals (v_i, v_j) having $r_i = \ell_j$. The mapping RS shifts these pairs correctly according to the edges of the graph.

To conclude the proof, we need to show that such a mapping RS exists. If we would relax the image of RS to $[0, \varepsilon')$, LS would be one correct mapping (since

reversal of LS produces the original correct representation). But correctness of RS depends only on the relative order of the shifts. So we can sort the left shifts and construct the right shifts in the same order. If $LS(v_i)$ is the k th smallest shift, we put $RS(v_i) = (k - 1)\varepsilon$; see Figure 3.5. \square

In the case of recognition, the partial representation is empty and we get $\varepsilon' = 1$ and $\varepsilon = \frac{1}{n}$. Lemma 3.3 gives in a particularly clean way the result mentioned in [CKN⁺95] that the grid of size $\frac{1}{n}$ is sufficient to construct unrestricted representations of unit interval graphs. In comparison, the paper [CKN⁺95] shows how to construct this representation directly from the ordering \triangleleft . Lemma 3.3 needs to have some representation to construct an ε -grid representation.

Also, Lemma 3.3 states that it is always possible to construct the ε -grid representation having “the same” topology as the original representations: Overlapping pairs of intervals keep overlapping and touching pairs of intervals keep touching.

Putting Together. Now, we are ready to show that the linear programming approach solves the problem $\text{REPEXT}(\text{UNIT INT})$ in polynomial time.

Proof of Theorem 1.3. According to Lemma 3.3, if the partial representation is extendible, there exists a representation on the ε -grid which means non-intersecting intervals are in distance at least ε .

For each component, there are at most two orderings we need to test since intervals in the same group can be represented identically.¹ Since we are trying to push the components as far to the left as possible, every component C_t has in every possible ε -grid representation the right-most endpoint at position greater or equal R_t . Therefore, if there exists a representation, there also exists an ε -grid representations which is constructed by the linear programs.

The approach is clearly polynomial in time, depending on the number of vertices n , the number of the edges m and the rational numbers in the partial representation \mathcal{R}' . \square

¹Not if they are pre-drawn but in such a case the algorithm does not move them and they just fix part of the ordering \triangleleft in this group. The rest of the group can be represented in an arbitrary order.

Chapter 4

Conclusions

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

4.1 Simultaneous Representations

A recent open problem by Jampani and Lubiw [JL10] is related to the partial representation extension for many classes. This relation is more a general principle than a precise mathematical statement.

We state once again the definition of this problem. Given a set of graphs G_1, \dots, G_k with common vertices I (meaning for every $i \neq j$ is $V(G_i) \cap V(G_j) = I$), do there exist representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ such that \mathcal{R}^i represents G_i and the representations equal on I (meaning for every i, j and $v \in I$ is $R_v^i = R_v^j$)? For an example, see Figure 4.1. We denote this problem $\text{SIMREP}(\mathcal{C})$ for a class \mathcal{C} .

Partial Representation Extension “solves” Simultaneous Representations. First of all, let the size of I be small. If there exist only small number of different representations of the subgraph induced by the vertices of I , we can test all of them and solve SIMREP using REPEXT . Thus we can get a fixed-parameter tractable algorithm with parameter $\ell = |I|$. We show this for interval graphs:

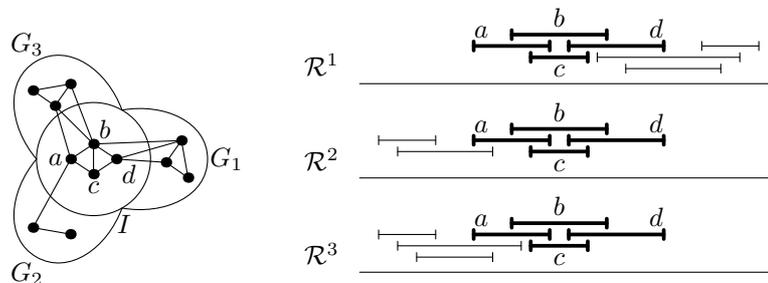


Figure 4.1: A simultaneous representation of G_1 , G_2 and G_3 with $I = \{a, b, c, d\}$.

Proposition 4.1. *We can solve SIMREP(INT) in time $\mathcal{O}((n+m) \cdot (2\ell)!)$, where n is the total number of vertices of all graphs and m is the total number of edges of all graphs.*

Proof. If the representation of I would be given, we could use the algorithm for REPEXT(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_i whether it is possible to extend the partial representation to a representation of entire G_i . This can be done in time $\mathcal{O}(n+m)$.

An interval graph with ℓ vertices has $\mathcal{O}((2\ell)!)$ topologically different representations. Therefore, we can test all possible representation of I and get the running time $\mathcal{O}((n+m) \cdot (2\ell)!)$. □ □

Similar relation holds for SIMREP(PROPER INT). We note that this is currently the best known algorithm for SIMREP(INT) when $k > 2$, no polynomial time algorithm is known. Jampani and Lubiw [JL10] give an algorithm in time $\mathcal{O}(n^2 \log n)$ for SIMREP(INT) when $k = 2$. A recent paper of Bläsius and Rutter [BR11] 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, this relation was suggested to us by Lubiw. Suppose that we have a graph G and a partial representation of G' . We construct an instance of SIMREP as follows: $I = G'$, $G_1 = G$ and G_2 contains additional vertices fixing the only possible representation of I to be topologically equivalent to the partial representation \mathcal{R}' . In such a case, we can solve REPEXT by SIMREP for two graphs G_1 and G_2 .

For example in the case of interval graphs, we construct G_2 as follows. Consider the partial representation and add a long path of short intervals on top of it. These intervals will be additional vertices of G_2 and will force a grid on the real line. This grid forces I to be represented topologically the same as in the partial representation. Thus we can solve REPEXT(INT) using SIMREP(INT) which is used in [BR11] to obtain an algorithm solving REPEXT(INT) in time $\mathcal{O}(n+m)$.

We note that this relation does not always work, it does not have to be possible to construct such a graph G_2 . For example in the case of partial representation extension of chordal graphs, the problem is NP-complete [KKOS]. On the other hand, the problem of simultaneous representations of chordal graphs is solvable in a polynomial time [JL09].

4.2 Allen Algebras

The following types of problems are studied in theory of artificial intelligence and time reasoning; see Golumbic [Gol98] 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

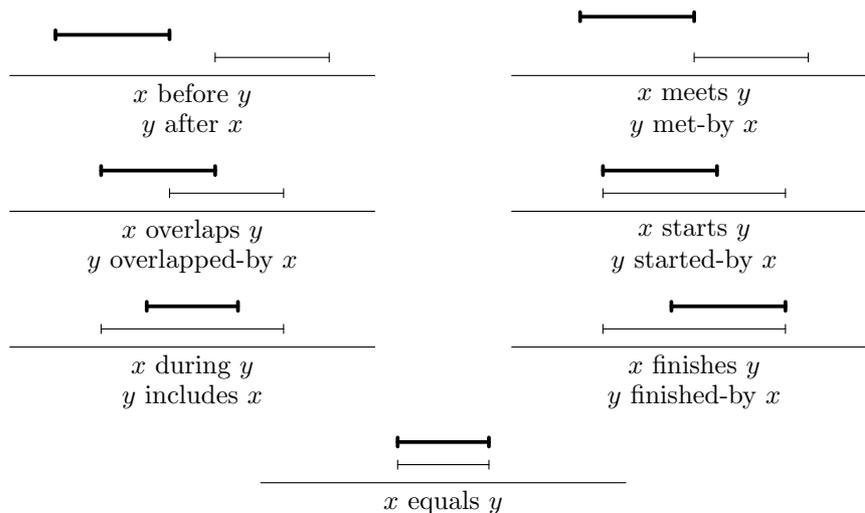


Figure 4.2: Thirteen primitive relations between the thick interval x and the thin interval y .

endpoints, Allen [All83] characterized thirteen *primitive relations* between the events, see Figure 4.2. A *relation* is a union 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 relations are satisfied. This is called the *interval satisfiability problem* (ISAT). Vilian and Kautz [VK86] proved that ISAT is NP-complete. Golumbic and Shamir [GS93] gave a more simple proof, using the interval graph sandwich problem.

Notice that we can describe $\text{REPEXT}(\text{INT})$ and $\text{REPEXT}(\text{PROPER INT})$ in the settings of Allen Algebras. In the case of $\text{REPEXT}(\text{INT})$, 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 intervals fixed by a partial representation, we give singleton relations. Similarly, we can specify $\text{REPEXT}(\text{PROPER INT})$.

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

4.3 Open Problems

To conclude the thesis, we state two open problems.

Question 4.2. *Is it possible to solve $\text{REPEXT}(\text{UNIT INT})$ in time $\mathcal{O}(n + m)$ or to solve it combinatorially without linear programming?*

This problem is still a work in progress but we believe it is possible to construct a much faster algorithm.

The formulation of the next problem is not very precise. All algorithm for solving partial representation extension problems we know are working 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. Similarly work the algorithms for extension of comparability, permutation and function graphs [KKKW12]. Also, the extension algorithm for planar graphs [ABF⁺10] uses SPQR-trees to work with all possible representations of the given planar graph.

Question 4.3. *Is it possible to solve some REPEXT problem more “directly”, without “testing” all possible representations?*

Appendix A

Pseudocodes of Algorithms

A.1 Reordering PQ-tree, general partial order

The pseudocode is in Algorithm 1.

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

Require: A PQ-tree T and a partial ordering \blacktriangleleft .

Ensure: A reordering T' of T such that $<_{T'}$ extends \blacktriangleleft if it exists.

- 1: Construct a digraph of \blacktriangleleft .
 - 2: Process nodes from bottom to the root:
 - 3: **for** a processed node N **do**
 - 4: Consider a subdigraph induced by 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.

Algorithm 2 Reordering a PQ-tree, interval order – REORDER(T, \blacktriangleleft)

Require: A PQ-tree T and an interval order \blacktriangleleft (with a normalized representation).

Ensure: A reordering T' of T such that $<_{T'}$ extends \blacktriangleleft if it exists.

- 1: Calculate both handles for each interval separately.
 - 2: Process nodes from bottom to the root:
 - 3: **for** a processed node N **do**
 - 4: Let subtrees of N represent sets $\mathcal{I}_1, \dots, \mathcal{I}_k$.
 - 5: Sort their lower and upper handles from left to right.
 - 6: **if** the node N is a P-node **then**
 - 7: Find any topological sorting by repeated removing of minimal elements.
 - 8: If it exists, reorder N according to it, otherwise output “no”.
 - 9: **else if** the node N is a Q-node **then**
 - 10: Test whether the current ordering or its reversal is compatible.
 - 11: 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.
 - 12: If one ordering is compatible, reorder the node, otherwise output “no”.
 - 13: **end if**
 - 14: Compute handles for $\mathcal{I} = \mathcal{I}_1 \cup \dots \cup \mathcal{I}_k$.
 - 15: **end for**
 - 16: **return** A reordering T' of T .
-

A.3 Extending Interval Graphs

The pseudocode is in Algorithm 3.

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.

- 1: Compute maximal cliques and construct a PQ-tree.
 - 2: Compute \curvearrowright and \curvearrowleft and construct an interval order \blacktriangleleft .
 - 3: Use the Algorithm from Section A.2 to reorder the PQ-tree according \blacktriangleleft .
 - 4: If any of these steps fails, no representation exists and output “no”.
 - 5: Place the clique-point in order from left to right, as far to the left as possible.
 - 6: Construct a representation \mathcal{R} , using the clique-points.
 - 7: **return** A representation \mathcal{R} extending \mathcal{R}' .
-

A.4 Extending Proper Interval Graphs

The pseudocode is in Algorithm 4.

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: Find located components and construct their linear order.
 - 2: **if** a component has its pre-drawn intervals split by some other component **then**
 - 3: Return “no”, \mathcal{R}' is not extendible.
 - 4: **end if**
 - 5: Calculate a reserved area for each component.
 - 6: **for** each component **do**
 - 7: Find a left-anchor v using BFS.
 - 8: Run BFS from v .
 - 9: Calculate a partial order $<$.
 - 10: Check whether the fixed order of pre-drawn intervals agrees with $<$ or its reversal.
 - 11: **if** both orders do not agree **then**
 - 12: The component is not extendible, output “no”.
 - 13: **end if**
 - 14: Produce a linear ordering of the vertices \triangleleft .
 - 15: Calculate the common order of the left and the right endpoints.
 - 16: Place the endpoints into the reserved area.
 - 17: **end for**
 - 18: **return** A representation \mathcal{R} extending \mathcal{R}' .
-

A.5 Extending Unit Interval Graphs

The pseudocode is in Algorithm 5.

Algorithm 5 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: Solve unlocated components separately.
 - 2: Process the located components C_1, \dots, C_c from left.
 - 3: **for** a located component C_t **do**
 - 4: Find the ordering $<$ (as described in Algorithm 4).
 - 5: Test $<$ and its reversal whether they are compatible with \mathcal{R}' , using the linear program.
 - 6: **if** some ordering is compatible **then**
 - 7: Use the ordering minimizing the value of R_t of the linear program.
 - 8: **else**
 - 9: No representation exists, output “no”.
 - 10: **end if**
 - 11: **end for**
 - 12: **return** A representation \mathcal{R} extending \mathcal{R}' .
-

Bibliography

- [ABF⁺10] P. Angelini, G. D. Battista, F. Frati, V. Jelínek, J. Kratochvíl, M. Patrignani, and I. Rutter. Testing Planarity of Partially Embedded Graphs. In *SODA '10: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010.
- [All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
- [Ben59] S. Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. U.S.A.*, 45:1607–1620, 1959.
- [BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *Journal of Computational Systems Science*, 13:335–379, 1976.
- [BR11] T. Bläsius and I. Rutter. Simultaneous PQ-Ordering with Applications to Constrained Embedding Problems. *CoRR*, abs/1112.0245, 2011.
- [CKN⁺95] D. G. Corneil, H. Kim, S. Natarajan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55(2):99–104, 1995.
- [Cor04] D. G. Corneil. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. *Discrete Appl. Math.*, 138(3):371–379, 2004.
- [COS09] D. G. Corneil, S. Olariu, and L. Stewart. The LBFS Structure and Recognition of Interval Graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1905–1953, 2009.
- [DHH96] X. Deng, P. Hell, and J. Huang. Linear-Time Representation Algorithms for Proper Circular-Arc Graphs and Proper Interval Graphs. *SIAM J. Comput.*, 25(2):390–403, 1996.
- [FG65] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15:835–855, 1965.
- [Fia03] J. Fiala. NP completeness of the edge precoloring extension problem on bipartite graphs. *J. Graph Theory*, 43(2):156–160, 2003.

- [Gol98] M. C. Golumbic. Reasoning about time. In *Mathematical Aspects of Artificial Intelligence*, F. Hoffman, ed., volume 55, pages 19–53, 1998.
- [Gol04] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. North-Holland Publishing Co., 2004.
- [GS93] M. C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *J. ACM*, 40(5):1108–1133, 1993.
- [Haj57] G. Hajós. Über eine Art von Graphen. *Internationale Mathematische Nachrichten*, 11:65, 1957.
- [HH95] P. Hell and J. Huang. 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.
- [JL09] K. Jampani and A. Lubiw. The Simultaneous Representation Problem for Chordal, Comparability and Permutation Graphs. In *Algorithms and Data Structures*, volume 5664 of *Lecture Notes in Computer Science*, pages 387–398. 2009.
- [JL10] K. Jampani and A. Lubiw. Simultaneous Interval Graphs. In *Algorithms and Computation*, volume 6506 of *Lecture Notes in Computer Science*, pages 206–217. 2010.
- [KKKW12] P. Klavík, J. Kratochvíl, T. Krawczyk, and B. Walczak. Extending partial representations of function graphs and permutation graphs. *Submitted to ESA 2012, track A*, 2012.
- [KKOS] P. Klavík, J. Kratochvíl, Y. Otachi, and T. Saitoh. Hardness of Partial Representation Extension for Chordal Graphs. *In preparation*.
- [KM91] J. Kratochvíl and J. Matoušek. String graphs requiring exponential representations. *J. Comb. Theory, Ser. B*, 53(1):1–4, 1991.
- [Kra91] J. Kratochvíl. String graphs. II. recognizing string graphs is NP-hard. *Journal of Combinatorial Theory, Series B*, 52(1):67–78, 1991.
- [Kur30] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.
- [LO93] P. J. Looges and S. Olariu. Optimal greedy algorithms for indifference graphs. *Comput. Math. Appl.*, 25:15–25, 1993.
- [Mar45] E. S. Marczewski. Sur deux propriétés des classes d’ensembles. *Fund. Math.*, 33:303–307, 1945.
- [Mar05] D. Marx. NP-completeness of list coloring and precoloring extension on the edges of planar graphs. *J. Graph Theory*, 49(4):313–324, 2005.

- [Pat06] M. Patrignani. On Extending a Partial Straight-Line Drawing. In *Lecture Notes in Computer Science*, volume 3843, pages 380–385, 2006.
- [Rob68] F. S. Roberts. *Representations of indifference relations*, Ph.D. Thesis. Stanford University, 1968.
- [Rob69] F. S. Roberts. Indifference graphs. In *F. Harary (Ed.), Proof Techniques in Graph Theory*, pages 139–146. Academic Press, 1969.
- [Rob76] F. S. Roberts. *Discrete Mathematical Models, with Applications to Social, Biological, and Environmental Problems*. Prentice-Hall, Englewood Cliffs, 1976.
- [RTL76] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- [Spi03] J. P. Spinrad. *Efficient Graph Representations*. Field Institute Monographs, 2003.
- [SSŠ02] M. Schaefer, E. Sedgwick, and D. Štefankovič. Recognizing string graphs in NP. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 1–6, 2002.
- [Sto68] K. E. Stoffers. Scheduling of traffic lights—a new approach. *Transportation Research*, 2:199–234, 1968.
- [Tro97] W. T. Trotter. New perspectives on interval orders and interval graphs. In *in Surveys in Combinatorics*, pages 237–286. Cambridge Univ. Press, 1997.
- [VK86] M. Vilian and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proc. Fifth Nat'l. Conf. on Artificial Intelligence*, pages 337–382, 1986.