LINEAR-TIME ALGORITHM FOR PARTIAL REPRESENTATION EXTENSION OF INTERVAL GRAPHS*

P. KLAVÍK[†], J. KRATOCHVÍL[‡], Y. OTACHI[§], T. SAITOH[¶], AND T. VYSKOČIL[‡]

Abstract. Interval graphs are intersection graphs of closed intervals of the real-line. The well-known computational problem, called *recognition*, asks for an input graph G whether it can be represented by closed intervals, i.e., whether G is an interval graph. There are several linear-time algorithms known for recognizing interval graphs.

In this paper, we study a generalization of recognition, called *partial representation extension.* Input of this problem consists of a graph G with a partial representation $\mathcal{R}'$ fixing positions of some intervals. The problem asks whether it is possible to place the remaining interval and create an interval representation $\mathcal{R}$ of the entire graph G extending $\mathcal{R}'$. We give a linear-time algorithm based on PQ-trees which solves this problem.

1. Introduction.

Graph representations and graph drawing are frequent topics of graph theory, studied for many theoretical and applied motivations. The class of *interval graphs* is one of the oldest classes of graphs, introduced by Hajós [10] already in 1957. An *interval representation* $\mathcal{R}$ of a graph G is a collection of closed intervals $\{R_v \mid v \in V(G)\}$ such that $uv \in E(G)$ if and only if $R_u \cap R_v \neq \emptyset$. In other words, $\mathcal{R}$ encodes the edges of G by intersections of the intervals. A graph G is called an interval graph if there exists an interval representation $\mathcal{R}$ of 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 decompositions. In many cases, very hard combinatorial problems are polynomially solvable for interval graphs; e.g., 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 [20, 23, 2].

Partial Representation Extension. For a fixed class $\mathcal{C}$, there is a natural well-studied problem called *recognition.* Given a graph G, the problem asks whether G belongs to $\mathcal{C}$. We denote this problem by $\textsc{Recog}(\mathcal{C})$. In the case of interval graphs, there are several known algorithms solving $\textsc{Recog}(\mathsf{INT})$ in linear time [4, 6].

We study a generalization of recognition called *partial representation extension*, introduced only recently in the conference version of this paper [16]. A *partial representation* $\mathcal{R}'$ of G is an interval representation of an induced subgraph G' of G. The intervals of G' are called *pre-drawn.* A representation $\mathcal{R}$ of G *extends* $\mathcal{R}'$ if it assigns the same intervals to the vertices of G', i.e., $R_v = R'_v$ for every $v \in V(G')$. Partial representation extension is the following decision problem:

---

*A conference version of this paper appeared in TAMC 2011 [16]. Supported by ESF Eurogiga project GraDR as GAČR GIG/11/E023. The first two authors are also supported by Charles University as GAUK 196213.

[†]Computer Science Institute, Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, 118 00 Prague, Czech Republic. E-mail: klavik@iuuk.mff.cuni.cz.

[‡]Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, 118 00 Prague, Czech Republic. E-mails: honza@kam.mff.cuni.cz and whisky@kam.mff.cuni.cz.

[§]School of Information Science, Japan Advanced Institute of Science and Technology. Asahidai 1-1, Nomi, Ishikawa 923-1292, Japan. Email: otachi@jaist.ac.jp

[¶]Graduate School of Engineering, Kobe University, Rokkodai 1-1, Nada, Kobe, 657-8501, Japan. E-mail: saitoh@eedept.kobe-u.ac.jp

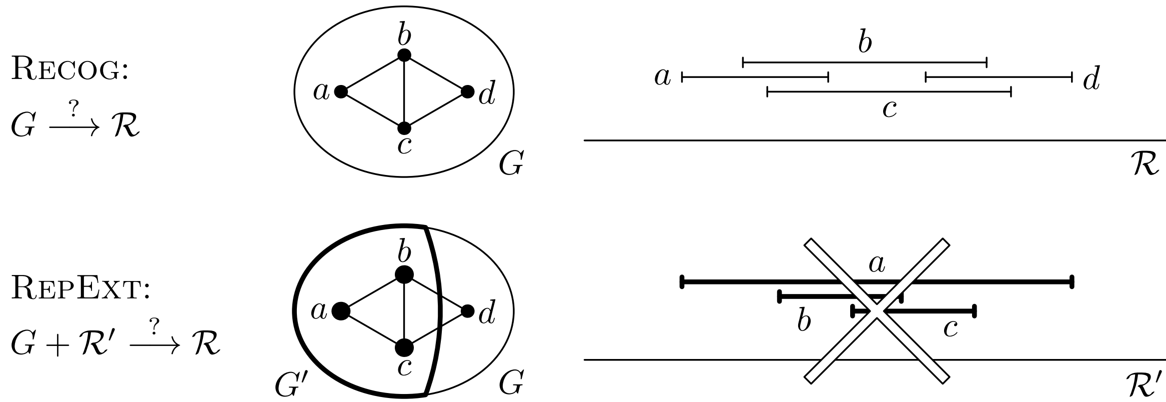

Fig. 1.1. *The graph G is an interval graph, but the partial representation $\mathcal{R}'$ is not extendible.*

Problem:	Partial Representation Extension – RepExt(INT)
Input:	A graph G and a partial representation $\mathcal{R}'$.
Question:	Is there an interval representation $\mathcal{R}$ of G extending $\mathcal{R}'$?

Figure 1.1 illustrates the difference between the recognition problem and the partial representation extension problem. In this paper, we establish the following main result:[1]

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

The partial representation extension problem can be considered in a more general setting of any intersection-defined class of graphs. A representation $\mathcal{R}$ of such a class $\mathcal{C}$ is a collection of sets R_v which belong to some well-defined class of objects (e.g., curves in the plane, k-dimensional boxes, chords of a circle). The graph is represented by $\mathcal{R}$ in exactly the same way, i.e., $uv \in E(G)$ if and only if $R_u \cap R_v \neq \emptyset$. For useful overview of these classes, see [9, 17, 22].

Previous results of RepExt. Concerning previous results for RepExt, the conference version of this paper [16] shows that interval representations can be extended in time $\mathcal{O}(n^2)$ and proper interval representations can be extended in time $\mathcal{O}(nm)$. For interval graphs, Bläsius and Rutter [3] improve this result to time $\mathcal{O}(n + m)$. In comparison, our linear-time algorithm is much simpler and easier to implement, based on the original idea of [16]. The algorithm of [3] uses a more general framework of simultaneous PQ-trees. Indeed this framework can be used to solve other problems, such as simultaneous planar embeddings or simultaneous interval graphs, but consequently their algorithm is quite involved.

The paper [14] improves [16] by giving a linear-time algorithm for proper interval graphs, and it also solves an open problem of [16] by giving an almost quadratic-time algorithm for unit interval graphs. These two results might seem surprising in the context of the Robert's Theorem [19] which states that these two classes are equal, i.e, PROPER INT = UNIT INT. But the partial representation extension problem distinguishes them. In the case of proper interval representation, a partial representation just prescribes some partial ordering of the endpoints of the intervals. But a partial unit interval representation gives in addition precise rational positions. The algorithm of [14] for unit interval graphs is based on linear programming and new structural results.

[1] We note that some minor natural assumptions for the input are necessary, and we further discuss them in the end of this section.

The paper [13] gives polynomial-time algorithms for permutation and function graphs. The paper [15] studies several possible versions of the problem for chordal graphs (in the setting of intersection graphs of subtrees of a tree) and shows that almost all of them are NP-complete. Concerning circle graphs, a polynomial-time algorithm is given by Chaplick et al. [5]. It is based on new structural results which describe via split decomposition all possible representations of circle graphs.

Every interval graph has a representation in which every endpoint is placed at an integer position. We note that extending such representations is an NP-complete problem [15]. (In this context we also require that an extending representation has its endpoints at integer positions.)

Concerning planar graphs, Angelini et al. [1] show that partial planar embeddings can be extended in linear time. The well-known Fáry's Theorem states that every planar graph has a straight-line embedding. But it is NP-complete to decide whether a partial straight-line embedding can be extended to a straight-line embedding of the entire graph [18].

Motivation for RepExt. To solve the recognition problem, an arbitrary representation can be constructed. Solving partial representation extension is harder and a better understanding of the structure of all possible representations seems to be necessary. This is a desirable property since one is forced to improve the structural understanding of the studied classes to solve this problem; and this structural understanding can be later applied in attacking of other problems.

The structure of all representations of interval graphs is already well understood [4], so for our algorithm we just use this structure. On the other hand, the papers [14, 5] build completely new structural results for unit interval and circle graphs which might be of independent interest.

Structure. This paper is structured as follows. In §2, we describe PQ-trees and give an algorithm for a reordering problem. In §3, this reordering algorithm is used as a subroutine for the main algorithm of Theorem 1.1. In §4, we discuss connections with the simultaneous representations problems and show that Theorem 1.1 gives an FPT algorithm for the simultaneous representations problem of interval graphs.

Remarks Concerning Input. To obtain the linear-time algorithm, we need some reasonable assumption on a 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.

We say that a partial representation is *sorted* if it gives the pre-drawn intervals sorted from left to right. We assume that the input partial representation is given sorted. If this assumption is not satisfied, the algorithms would need additional time $\mathcal{O}(k \log k)$ to sort the partial representation where k is the number of pre-drawn intervals. We note that Bläsius and Rutter [3] need the same assumption for their linear-time algorithm.

2. PQ-trees and the Reordering Problem.

To describe PQ-trees, we start with a motivational problem. An input of the *consecutive ordering problem* consists of 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 the elements* $E = \{a, b, c, d, e, f, g, h\}$ *and the restricting sets* $S_1 = \{a, b, c\}$, $S_2 = \{d, e\}$, *and* $S_3 = \{e, f, g\}$. *For instance, the orderings abcdefgh and fgedhacb are feasible. On the other hand, the orderings* $\underline{ac}defg\underline{b}h$ *(violates*

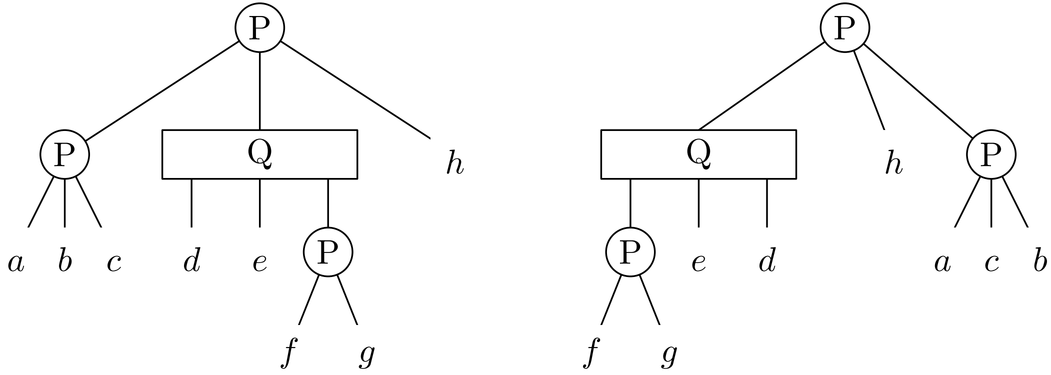

FIG. 2.1. *PQ-trees representing orderings abcdefgh and fgedhacb.*

S_1) and defhgabc (violates S_3) are not feasible.

PQ-trees. A PQ-tree is a tree structure designed for solving the consecutive ordering problem 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.* The tree is rooted and an order of children of every inner node is fixed. Also we assume that each inner node has at least two children. 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. We say that a tree T' is a *reordering* of T if it can be created from T by applying several reordering operations. Two trees are *equivalent* if one is reordering of the other. 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 paper, 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 [4] describe details of their construction.

2.1. The Reordering Problem for General Orderings. 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 $\lhd$, i.e., $a \lhd b$ implies $a <_{T'} b$.

PROBLEM:	The reordering problem – REORDER($T, \lhd$)
INPUT:	A PQ-tree T and a partial ordering $\lhd$.
QUESTION:	Is there a reordering T' of T compatible with $\lhd$?

Local Solutions. A PQ-tree defines some hierarchical structure on its elements. A *subtree* of a PQ-tree consists of one inner node and all its successors.

OBSERVATION 2.2. *Let S be a subtree of a PQ-tree T. Then the elements of E contained in S appear consecutively in $<_T$.*

We start with a lemma which states the following: If we can solve the problem locally (inside of some subtree), then this local solution is always correct; either there exists no solution of the problem at all, or our local solution can be extended to a

solution 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 $\lhd$. According to Observation 2.2, all elements contained in S appear consecutively in $<_{T'}$. Therefore, we can replace this local ordering of S by any other local ordering of S satisfying all constraints given by $\lhd$. We obtain another reordering of the whole tree T which is compatible with $\lhd$ and extends the pre-scribed local ordering of S. $\square$

The Algorithm. We describe the following algorithm for REORDER$(T, \lhd)$:

PROPOSITION 2.4. *The problem* REORDER$(T, \lhd)$ *can be solved in time $\mathcal{O}(e+m)$, where e is the number of elements and m is the number of comparable pairs in $\lhd$.*

Proof. The algorithm is based on the following greedy procedure. We represent the ordering $\lhd$ by a digraph having m edges. We reorder the nodes from the 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 the corresponding vertices in the digraph. We process a node of the PQ-tree when all its subtrees are already processed and the digraph is trivial.

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 sort. 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 [24], and to check whether one of them is feasible. For an example, see Figure 2.2.

We need to argue correctness. The algorithm proceeds the tree from the bottom to the top. For every subtree S, it finds some reordering of S compatible with $\lhd$. If no such reordering of S exists, the whole tree T cannot be reordered according to $\lhd$.

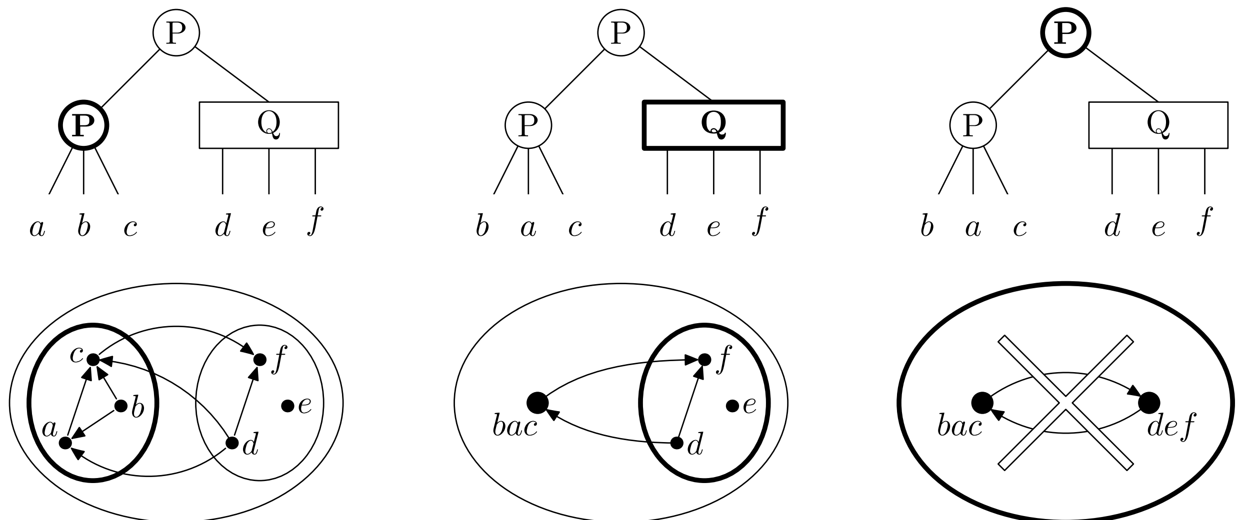

FIG. 2.2. *We show from left to right an example how the reordering algorithm works. First, we reorder the highlighted P-node on the left. The subdigraph induced by a, b and c has the topological sort $b \to a \to c$. We contract these vertices into the vertex bac. Next, we keep the order of the highlighted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

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 the 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 sort 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.

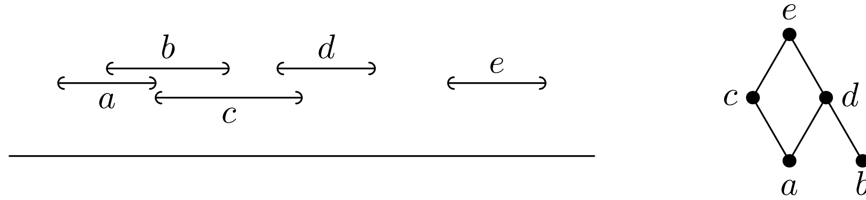

FIG. 2.3. *On the left, a collection of intervals. On the right, the Hasse diagram of the interval order $\lhd$ represented by these intervals.*

If a reordering of S exists, it is correct according to Lemma 2.3.

The algorithm can be implemented in linear time with respect to 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 before it is contracted. □

We note that the described algorithm works even for a general relation $\lhd$. For example, $\lhd$ does not have to be transitive (as in the example in Figure 2.2) or even acyclic (but in such a case, of course, no solution exists). A pseudocode is given in Algorithm 1.

2.2. The Reordering Problem for Interval Orders. Let E be a set and let $\{I_a = (\ell_a, r_a) \mid a \in E\}$ be a collection of open intervals.[2] Then these intervals *represent* the following partial ordering $\lhd$ on E. If two intervals I_a and I_b do not intersect, then one is on the left and the other is on the right. For $a, b \in E$, we put $a \lhd b$ if and only if $r_a \le \ell_b$. A partial ordering of E is called an *interval order* if there exists a collection of intervals representing this ordering in this way. See Figure 2.3 for an example.

Both interval graphs and interval orders are represented by collections of intervals, and indeed they are closely related [7]. The study of interval orders has the following

[2]For the purpose of §3, we allow empty intervals with $\ell_v = r_v$.

motivation. Suppose that the elements of E correspond to events and each interval describes when an event can happen in the timeline. 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 the corresponding events. Nevertheless, for purpose of this paper, we only need to know the definition of interval orders. For more information, see the survey [25].

Faster Reordering of PQ-trees. Let e be the number of elements of E and let $\lhd$ be an interval order of E represented by $\{I_a \mid a \in E\}$. We assume the representation is sorted which means that we know the order of the endpoints of the intervals from left to right. We show that for such $\lhd$ we can solve $\textsc{Reorder}(T, \lhd)$ faster:

PROPOSITION 2.5. *If $\lhd$ is an interval order given by a sorted representation, we can solve the problem* $\textsc{Reorder}(T, \lhd)$ *in time $\mathcal{O}(e)$ where e is the number of elements of T.*

For the following, let $\lessdot$ be a linear ordering of the endpoints ℓ_e and r_e of the intervals according to their appearance from left to right in the representation. To ensure that $a \lhd b$ if and only if $r_a \lessdot \ell_b$, we need to deal with endpoints sharing position. For them, we place in $\lessdot$ first the right endpoints (ordered arbitrarily) and then the left endpoints (again ordered abitrarily). For a sorted representation, this ordering $\lessdot$ can be computed in time $\mathcal{O}(e)$.

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 constraints. Using the interval representation of $\lhd$, we can just implement all steps faster then before.

The main trick is that we do not construct the digraph explicitly. Instead, we just work with sets of intervals corresponding to subtrees and compare them with respect to $\lhd$ fast. When we process a node, its children correspond to sets $\mathcal{I}_1, \ldots, \mathcal{I}_k \subseteq V(H)$ we already processed before. We test efficiently in time $\mathcal{O}(k)$ whether we can reorder these k subtrees according to $\lhd$. If it is not possible, the algorithm stops and outputs "no". If the reordering succeeds, we put all the sets together $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots \cup \mathcal{I}_k$, and proceed further.

Comparing Subtrees. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be sets of intervals. We say $\mathcal{I}_1 \lhd \mathcal{I}_2$ if there exist $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$ such that $a \lhd b$. We want to show that using the interval representation and some precomputation, 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 2.6. *Suppose that $a \lhd b$, $a \in \mathcal{I}_1$ and $b \in \mathcal{I}_2$. Then for every $a' \in \mathcal{I}_1, r_{a'} \lessdot r_a$ and every $b' \in \mathcal{I}_2, \ell_b \lessdot \ell_{b'}$ also holds that $a' \lhd b'$.*

Proof. From the definition, $a \lhd b$ if and only if $r_a \leq \ell_b$. We have $r_{a'} \lessdot r_a \lessdot \ell_b \lessdot \ell_{b'}$, and thus $a' \lhd b'$. See Figure 2.4. □

Using the previous lemma, we just need to compare a having the left-most r_a to b having the right-most ℓ_b since $\mathcal{I}_1 \lhd \mathcal{I}_2$ if and only if $a \lhd b$.

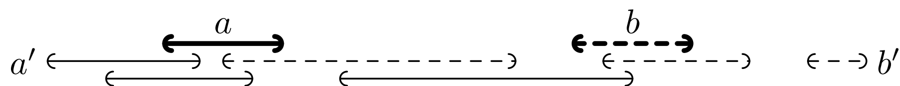

FIG. 2.4. *The normal intervals belong to $\mathcal{I}_1$ and the dashed intervals belong to $\mathcal{I}_2$. If $a \lhd b$, then also $a' \lhd b'$.*

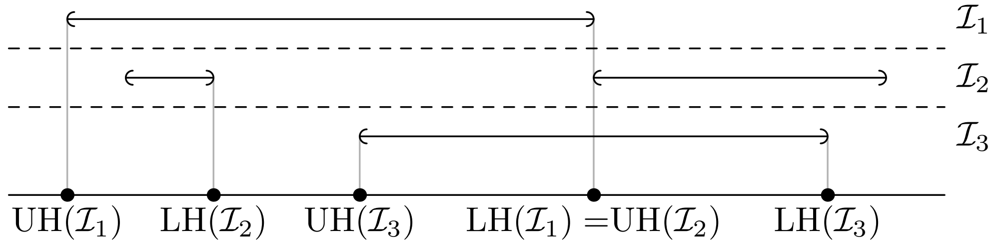

FIG. 2.5. *The handles for sets* $\mathcal{I}_1$, $\mathcal{I}_2$ *and* $\mathcal{I}_3$. *We have* $\mathrm{UH}(\mathcal{I}_1) \lessdot \mathrm{LH}(\mathcal{I}_2) \lessdot \mathrm{UH}(\mathcal{I}_3) \lessdot \mathrm{LH}(\mathcal{I}_1) \lessdot \mathrm{UH}(\mathcal{I}_2) \lessdot \mathrm{LH}(\mathcal{I}_3)$. *According to (2.2), we get* $\mathcal{I}_1 \lhd \mathcal{I}_2$, $\mathcal{I}_2 \lhd \mathcal{I}_3$, *and* $\mathcal{I}_1 \not\lhd \mathcal{I}_3$; *so the relation* $\lhd$ *on sets of intervals is not necessarily transitive.*

To simplify the description, these special endpoints of intervals used for comparisons are called *handles*. More precisely, for each set of intervals $\mathcal{I}$, we define a *lower handle* and *upper handle*:

$$\mathrm{LH}(\mathcal{I}) = \min\{r_x \mid x \in \mathcal{I}\} \qquad \text{and} \qquad \mathrm{UH}(\mathcal{I}) = \max\{\ell_x \mid x \in \mathcal{I}\}. \tag{2.1}$$

We note that $\mathrm{LH}(\mathcal{I}) \lessdot \mathrm{UH}(\mathcal{I})$ if $\mathcal{I}$ is not a clique. Using handles, we can compare sets of intervals fast. According to Lemma 2.6, we have:

$$\mathcal{I}_1 \lhd \mathcal{I}_2 \qquad \text{if and only if} \qquad \mathrm{LH}(\mathcal{I}_1) \lessdot \mathrm{UH}(\mathcal{I}_2). \tag{2.2}$$

For an example, see Figure 2.5.

So throughout the algorithm, we just remember these handles for each processed subtree, and we do not need to remember which specific intervals are contained in the subtree. So the handles serve in the same manner as the contraction operation on the digraph.

Reordering Nodes. We describe how to reorder children of a processed node fast using the handles. Let $\mathcal{I}_1, \dots, \mathcal{I}_k$ be sets of intervals corresponding to the subtrees defined by children of this node. Suppose that we know their handles and have them ordered according to $\lessdot$ as in Figure 2.5. Let $\widetilde{\lessdot}$ be the ordering $\lessdot$ restricted to the handles of $\mathcal{I}_1, \dots, \mathcal{I}_k$.

A linear ordering $<$ of sets $\mathcal{I}_1, \dots, \mathcal{I}_k$ is called *topological sort* if $\mathcal{I}_i \lhd \mathcal{I}_j$ 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 sort. If it is a Q-node, we need to test whether the current ordering or its reversal is a topological sort.

An element $\mathcal{I}_j$ is *minimal* if there is no $\mathcal{I}_i$ such that $\mathcal{I}_i \lhd \mathcal{I}_j$. The following lemma allows to locate minimal elements:

LEMMA 2.7. *Let* $\mathcal{I}_j$ *be an element. It is a minimal element if and only if there is no lower handle* $\mathrm{LH}(\mathcal{I}_i)$ *for* $i \neq j$ *such that* $\mathrm{LH}(\mathcal{I}_i) \widetilde{\lessdot} \mathrm{UH}(\mathcal{I}_j)$.

Proof. According to (2.2), $\mathcal{I}_i \lhd \mathcal{I}_j$ if and only if $\mathrm{LH}(\mathcal{I}_i) \lessdot \mathrm{UH}(\mathcal{I}_j)$. If these is no such $\mathcal{I}_i$, then $\mathcal{I}_j$ is minimal. □

We can use this lemma to identify all minimal elements:

- If the ordering $\widetilde{\lessdot}$ starts with two lower handles $\mathrm{LH}(\mathcal{I}_i)$ and $\mathrm{LH}(\mathcal{I}_j)$, there exists no minimal element. The reason is that all upper handles are larger, and so both $\mathcal{I}_i$ and $\mathcal{I}_j$ are smaller than everything else; specifically, we get $\mathcal{I}_i \lhd \mathcal{I}_j \lhd \mathcal{I}_i$.
- If the first element of the ordering $\widetilde{\lessdot}$ is $\mathrm{LH}(\mathcal{I}_i)$ then $\mathcal{I}_i$ is the unique candidate for a minimal element. We just need to check whether there is some other

LH($\mathcal{I}_j$) smaller than UH($\mathcal{I}_i$), and if so, no minimal element exists.[3]

- If $\widetilde{\lessdot}$ starts with a consecutive group of upper handles, we have several candidates for a minimal element. First, all $\mathcal{I}_i$'s of these upper handles are minimal elements. Second, if the lower handle following the group of upper handles is LH($\mathcal{I}_j$), then $\mathcal{I}_j$ is a candidate for a minimal element. As above, $\mathcal{I}_j$ is minimal if there is no other lower handle smaller than UH($\mathcal{I}_j$).

For every topological sort, the ℓ-th element is minimal when restricted to the elements $\{\ell, \ell+1, \dots, k\}$. So we can construct all topological sorts as follows. We repeatedly detect all minimal element $\mathcal{I}_i$ and always pick one of them. (For different choices we get different topological sorts). We remove the handles of the picked minimal element $\mathcal{I}_i$ from $\widetilde{\lessdot}$ and append $\mathcal{I}_i$ to the constructed topological sort. We stop when all elements are placed in the topological sort. If in some step no minimal element exists, we know that no topological sort exists.

For a P-node, we just need to find any topological sort by repeated removing of minimal elements. For a Q-node, we test whether the current ordering or its reversal is a topological sort. We iterate through each of the prescribed sorts, check whether each element is a minimal element, and then removing its handles from $\widetilde{\lessdot}$. In both cases, if we find a correct topological sort, we use it reorder the children of the node. Otherwise, the reordering is not possible and the algorithm outputs "no". We are able to do the reordering of the node in time $\mathcal{O}(k)$.

The Algorithm. Now, we are ready to show that our algorithm allow us to find a reordering of the PQ-tree T according to an interval order $\lhd$ with a sorted representation in time $\mathcal{O}(e)$:

Proof. [Proposition 2.5] We first deal with details of the implementation. First, we precompute the handles for every set of intervals corresponding to the subtree of every inner node of T. For each leaf, the two handles correspond for the two endpoints. We proceed the tree from bottom to the root. Suppose that we have an inner node corresponding to the set $\mathcal{I}$ of intervals and it has k children corresponding to $\mathcal{I}_1, \dots, \mathcal{I}_k$ for which we already know their handles. Then we can calculate handles of $\mathcal{I}$ using

$$\mathrm{LH}(\mathcal{I}) = \min\bigl\{\mathrm{LH}(\mathcal{I}_i)\bigr\} \qquad \text{and} \qquad \mathrm{UH}(\mathcal{I}) = \max\bigl\{\mathrm{UH}(\mathcal{I}_i)\bigr\}. \tag{2.3}$$

This can clearly be computed in time $\mathcal{O}(e)$, and we also note for each endpoint a list of nodes for which it is a handle. Using these list, we can iterate the sorted representation and compute the orderings $\widetilde{\lessdot}$ for every inner node of T, again in $\mathcal{O}(e)$.

Now we test each inner node of T with the given ordering $\widetilde{\lessdot}$ whether its subtrees can be reordered according to $\lhd$. The algorithm is correct since it works in the same way as in Proposition 2.4, based on lemmas 2.6 and 2.7.

Concerning the time complexity, we already discussed that we are able to compare sets of intervals using handles in a constant time, by Lemma 2.6. The precomputation of all orderings $\widetilde{\lessdot}$ takes time $\mathcal{O}(e)$. 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 $\mathcal{O}(e)$. ☐

For a pseudocode, see Algorithm 2. We note that when the orderings $\widetilde{\lessdot}$ are constructed for all inner nodes, then we do not need to proceed the tree from bottom to the top. We can proceed them independently in parallel and a reordering T' of T exists if and only if we succeed in reordering of every inner node.

[3]This can be done in constant time if we remember in each moment positions of the two left-most lower handles in the ordering, and update this information after removing one of them from $\widetilde{\lessdot}$.

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

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

1: Calculate the handles for each invididual leaf of T and initiate an empty list for each endpoint.
2: Process the nodes of T from the bottom to the root:
3: **for** a processed node N **do**
4: Compute the handles of N using (2.3).
5: Add the node N to the lists of the two endpoints which are the handles of N.
6: **end for**
7: Iterate the sorted representation and construct the ordering $\widetilde{\lessdot}$ for each inner node N.
8: Again process the nodes from bottom to the root:
9: **for** a processed node N with the ordering $\widetilde{\lessdot}$ **do**
10: **if** the node N is a P-node **then**
11: Find any topological sort by removing minimal elements from $\widetilde{\lessdot}$.
12: If it exists, reorder N according to it, otherwise output "no".
13: **else if** the node N is a Q-node **then**
14: Test whether the current ordering or its reversal is a topological sort.
15: Process the prescribed topological sort from left to right, check for every element whether it is minimal and remove its handles from $\widetilde{\lessdot}$.
16: If at least one ordering is correct, reorder the node, otherwise output "no".
17: **end if**
18: **end for**
19: **return** A reordering T' of T.

3. Extending Interval Graphs. In this section, we describe an algorithm solving REPEXT(INT) in time $\mathcal{O}(n+m)$ which uses Proposition 2.5 as a subroutine. Unlike in §2, the interval representations consist of closed intervals. We allow the intervals to share the endpoints and to have zero length. We first describe recognition of interval graphs. Then we show how to modify the PQ-tree approach to solve REPEXT(INT).

3.1. Recognition using PQ-trees. Recognition of interval graphs in linear time was a long-standing open problem, first solved by Booth and Lueker [4] 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 [6].

We modify the PQ-tree approach to solve REPEXT(INT) in time $\mathcal{O}(n+m)$. Recall the PQ-trees from §2.

Maximal Cliques. The PQ-tree approach is based on the following characterization of interval graphs, due to Fulkerson and Gross [8]:

LEMMA 3.1 (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,

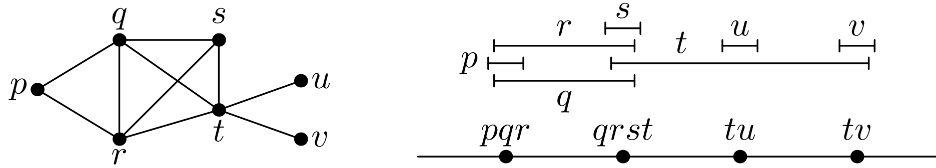

FIG. 3.1. *An interval graph and one of its representations with denoted clique-points.*

consider the intervals representing the vertices of this maximal 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 3.1. The ordering of the clique-points from left to right gives the ordering required by Lemma 3.1. Every vertex appears in consecutive maximal cliques since it is represented by an interval. For a maximal 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 maximal cliques containing this vertex. In this way, we obtain a valid interval representation of the graph.

Recognition Algorithm. Every chordal graph has at most $\mathcal{O}(n)$ maximal cliques of total size $\mathcal{O}(n+m)$ and they can be found in linear time [21]. Since every interval graph is chordal, we run this subroutine. If this subroutine fails, the input graph is not an interval graph, and the recognition algorithm outputs "no".

According to Lemma 3.1, we want to find an ordering of the maximal clique in which maximal clique containing each vertex appear consecutively. Recall the consecutive ordering problem from §2. Here, the elements E are the maximal cliques of the graph. For each vertex v, we introduce the restricting set S_v containing all the maximal cliques containing this vertex v. Using PQ-trees, we can find a feasible ordering of the maximal cliques and recognize an interval graph in time $\mathcal{O}(n+m)$.

3.2. Modification for RepExt. We first sketch the algorithm. We construct a PQ-tree T for the input graph while ignoring the partial representation. The partial representation gives another restriction—an interval order $\lhd$ of the maximal cliques. Using Proposition 2.5, we try to find a reordering T' of the PQ-tree T according to $\lhd$ in time $\mathcal{O}(n+m)$. We are going to prove the following proposition: *The partial representation is extendible if and only if the reordering algorithm succeeds.*

Since our proof is constructive, we can use to build a representation $\mathcal{R}$ extending the partial representation $\mathcal{R}'$. We place clique-points on the real line according to the ordering $<_{T'}$. We need to be more careful in this step. Since several intervals are pre-drawn, we cannot change their representations, so the clique-points has to be placed correctly. Using the clique-points, we construct the remaining intervals in a similar manner as in Figure 3.1.

Now, we describe everything in detail.

Restricting Clique-points. Suppose that there exists a representation $\mathcal{R}$ extending $\mathcal{R}'$. Then $\mathcal{R}$ gives some ordering $<$ of the clique-points from left to right. We want to show that pre-drawn intervals partially specify position of some clique-points and give some necessary condition for $<$.

For a maximal clique a, let $I(a)$ denote the set of all the pre-drawn intervals that are contained in a. Then $I(a)$ restricts possible position of $\mathrm{cp}(a)$ to only those points x of the real line which are covered in $\mathcal{R}'$ by exactly the pre-drawn intervals of $I(a)$

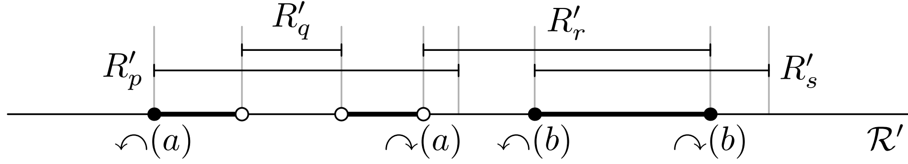

FIG. 3.2. *The partial representation $\mathcal{R}'$ consisting of four pre-drawn intervals. Clique-points* $\mathrm{cp}(a)$ *and* $\mathrm{cp}(b)$*, having $I(a) = \{p\}$ and $I(b) = \{r, s\}$, can be placed to the bold parts of the real lines.*

and no others. 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:

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

Notice that $(\curvearrowleft(a), \curvearrowright(a))$ is a subinterval of $\bigcap_{u \in I(a)} R'_u$. For an example, see Figure 3.2.

For a clique-point $\mathrm{cp}(a)$, the structure of all x where $\mathrm{cp}(a)$ can be placed is simple. The pre-drawn intervals split the line into several *parts*, traversed by the same intervals; denoted in Figure 3.2 by gray lines. A clique-point $\mathrm{cp}(a)$ can be placed only to those part which contain exactly the intervals of $I(a)$ and no other pre-drawn intervals. So the set of all points x where $\mathrm{cp}(a)$ can be placed is a union of intervals.

Notice also that the definition of $\curvearrowleft(a)$ and $\curvearrowright(a)$ does not imply that $\mathrm{cp}(a)$ can be placed to all the points between $\curvearrowleft(a)$ and $\curvearrowright(a)$. If a clique-point cannot be placed at all, the given partial representation is clearly not extendible.

The Interval Order $\lhd$. For two maximal cliques a and b, we define $a \lhd b$ if $\curvearrowright(a) \le \curvearrowleft(b)$. The definition of $\lhd$ is quite natural since $a \lhd b$ implies that every extending representation $\mathcal{R}$ has to place $\mathrm{cp}(a)$ to the left of $\mathrm{cp}(b)$. For example, the maximal cliques a and b in Figure 3.2 satisfy $a \lhd b$.

LEMMA 3.2. *The relation $\lhd$ is an interval order.*

Proof. The intervals representing $\lhd$ correspond to the maximal cliques of G. To a maximal 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. ◻

LEMMA 3.3. *For a sorted partial representation $\mathcal{R}'$, we can compute the sorted representations of $\lhd$ in time $\mathcal{O}(n + m)$.*

Proof. As stated above, our interval graph has $\mathcal{O}(n)$ maximal cliques containing in total $\mathcal{O}(n + m)$ vertices. We compute for every pre-drawn vertex the list of the maximal cliques containing it, and for every maximal clique a the number $|I(a)|$ of pre-drawn vertices it contains. We initiate an empty list W and a counter i of pre-drawn intervals covering the currently sweeped point. We sweep the real line from left to right and compute the sorted representation of $\lhd$.

When sweeping there are two types of events. If we encounter a set of endpoints of pre-drawn intervals sharing a point, we first process the left endpoints, then we update $\curvearrowleft$ and $\curvearrowright$ for this point[4], and then we process the right endpoints. If we sweep over a part, we just update $\curvearrowleft$ and $\curvearrowright$. We do in details the following:

[4]We need to update also here since it might happen that the interval $(\curvearrowleft(a), \curvearrowright(a))$ is empty for some maximal clique a. This can happen only if some pre-drawn interval of $I(a)$ is a singleton.

- *If we encounter a left endpoint* ℓ_u, then we increase the counter i. For every clique a containing u, we increase its counter. If some clique a has all pre-drawn intervals placed over ℓ_u, we add a into the list W of watched cliques.
- *If we encounter a right endpoint* r_u, we decrease the counter of pre-drawn intervals. We ignore all maximal cliques a containing u till the end of the procedure, and naturally we also remove them from W if there are any.
- *Update of* $\curvearrowleft$ *and* $\curvearrowright$ is done for all cliques $a \in W$ such that $|I(a)| = i$. Notice that we currently sweep over exactly i pre-drawn intervals, and therefore we have to sweep over exactly the pre-drawn intervals of $I(a)$. We update $\curvearrowleft(a)$ to the left-most point of the current point/part if it is not yet initialized. And we update $\curvearrowright(a)$ to the right-most such point.

In the end, we output the computed $\curvearrowleft$ and $\curvearrowright$ naturally sorted from left to right. If for some maximal clique a, the value $\curvearrowleft(a)$ was not initiated, the clique-point $\mathrm{cp}(a)$ cannot be placed and the procedure outputs “no”.

The procedure is clearly correct, and it remains to argue that this can be done in linear time. We have the cliques in W partitioned according to $|I(a)|$. Also notice that when we sweep over i pre-drawn intervals, then there is no $a \in W$ such that $|I(a)| > i$ and if $a, b \in W$ such that $|I(a)| = |I(b)| = i$, then necessarily $I(a) = I(b)$. But then $\curvearrowleft(a) = \curvearrowleft(b)$ and $\curvearrowright(a) = \curvearrowright(b)$, so we can ignore b for the rest of the sweep procedure and set in the end the value according to the clique a. This implementation clearly runs in $\mathcal{O}(n+m)$. □

Using Lemma 3.3, we can construct the sorted representation of $\lhd$ in time $\mathcal{O}(n+m)$. We can test $\textsc{Reorder}(T, \lhd)$ in time $\mathcal{O}(n+m)$ using Proposition 2.5. The following proposition is key for the correctness of the algorithm.

PROPOSITION 3.4. *A partial representation $\mathcal{R}'$ is extendible if and only if G is an interval graph and the problem* $\textsc{Reorder}(T, \lhd)$ *can be solved.*

Proof. The condition forced by $\lhd$ are clearly necessary. So if $\mathcal{R}'$ is extendible, then the both condition have to be satisfied. It remain to show the other implication.

Since G is an interval graph, there exists a PQ-tree T representing all feasible orderings of the maximal cliques. There exists a reordering T' of T according to $\lhd$, and we denote $<_{T'}$ as $<$. We construct a representation $\mathcal{R}$ extending $\mathcal{R}'$ as follows. We place the clique-points according to $<$ from left to right, always greedily as far to the left as possible.

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 prove by contradiction that this greedy procedure cannot fail; see Figure 3.3. 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 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

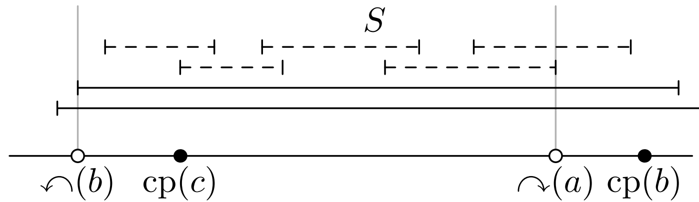

Fig. 3.3. *An illustration of the proof: The positions of the clique-points* cp(b) *and* cp(c)*, the intervals of S are dashed.*

right of $\curvearrowleft(b)$ (otherwise we could place cp(b) to $\curvearrowleft(b)$ or right next to it). Let cp(c) be the right-most clique-point placed between $\curvearrowleft(b)$ and cp(b). Every point between 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 $I(b)$ covering any point of the real line in $[c, \curvearrowright(a)]$; depicted dashed in Figure 3.3.

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 $<$ because every pair of adjacent vertices from S is contained in some maximal clique of C.

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, then $a < b$ which contradicts $b < a$. □

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. Independently of the partial representation, find the maximal cliques and construct a PQ-tree T representing all feasible orderings of the maximal cliques.
2. Construct the sorted representation of the interval order $\lhd$ by Lemma 3.3.
3. Using Proposition 2.5, test whether there is a reordering T' of the PQ-tree T according to $\lhd$.
4. Place the clique-points from left to right according to $<_{T'}$ on the real line,

---

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: Sweep $\mathcal{R}'$ from left to right and construct the sorted representation of $\lhd$.
3: Use Algorithm 2 to reorder the PQ-tree according $\lhd$.
4: If any of these steps fails, no representation exists and output "no".

5: Place the clique-point according to the ordering $<_{T'}$ from left to right:
6: **for** a clique-point cp(a) placed after cp(b) **do**
7: Compute the infimum of all points of the real line on the right of cp(b) where cp(a) can be placed.
8: If there is single such point, place cp(a) there.
9: Otherwise place cp(a) by ε on the right of the infimum, where ε is the size of the smallest part divided by n.
10: **end for**
11: Construct $\mathcal{R}$ for the remaining intervals on top of the placed clique-points.

12: **return** A representation $\mathcal{R}$ extending $\mathcal{R}'$.

---

greedily as far to the left as possible.

5. Using these clique-points, construct a representation $\mathcal{R}$ extending $\mathcal{R}'$.

Step 1 is the original recognition algorithm. In Step 2, we compute splitting of the real line into parts and construct a sorted representation of $\lhd$. In Step 3, we apply the algorithm of Proposition 2.5. Step 4 is the greedy procedure from the proof of Proposition 3.4. In Step 5, we construct intervals representing the vertices of $G \setminus G'$ as in Figure 3.1; we construct each such interval on top of the corresponding clique-points. See Algorithm 3 for a pseudocode.

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

Proof. [Theorem 1.1] The correcness of the algorithm is implied by Proposition 3.4. Concerning the complexity, the total size of all maximal cliques is at most $\mathcal{O}(n+m)$ and the PQ-tree can be constructed in this time. Using Lemma 3.3, we can construct the sorted representation of $\lhd$ in time $\mathcal{O}(n+m)$. According to Proposition 2.5, the PQ-tree can be reordered according to $\lhd$ in time $\mathcal{O}(n+m)$. Finally, a representation $\mathcal{R}$ extending $\mathcal{R}'$ can be constructed, if necessary, in time $\mathcal{O}(n+m)$. □

4. Simultaneous Representations of Interval Graphs. The input of the simultaneous representations problem gives several graphs $G_1, \dots, G_k$ having a common intersection I. The task is to construct their representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ which represent the vertices of I the same; see Figure 4.1. Formally it is the following decision problem:

Problem:	Simultaneous Representations – SimRep($\mathcal{C}$)
Input:	Graphs $G_1, \dots, G_k$ such that $G_i \cap G_j = I$ for all $i \neq j$.
Question:	Do there exist representations $\mathcal{R}^1, \dots, \mathcal{R}^k$ such that $\mathcal{R}^i = \{R^i_u \mid u \in V(G_i)\}$ represents G_i and for every $u \in I$ we have $R^i_u = R^j_u$ for every i and j.

Jampani et al. [12] show that for permutation and comparability graphs the problem can be solved in polynomial time for any number of graphs, and for chordal graphs the problem is polynomially solvable for $k = 2$ and NP-complete when k is a part of the input. For two interval graphs, the paper [11] gives an $\mathcal{O}(n^2 \log n)$ algorithm, which Bläsius and Rutter [3] improve to $\mathcal{O}(n+m)$. For circle graphs, the problem is NP-complete when k is a part of the input [5] and open for $k = 2$.

Relation to RepExt. For many classes, the simultaneous representations problem is closely related to the partial representation extension problem. A negative example is the class of chordal graphs, denoted by CHOR. The problem SimRep(CHOR) is

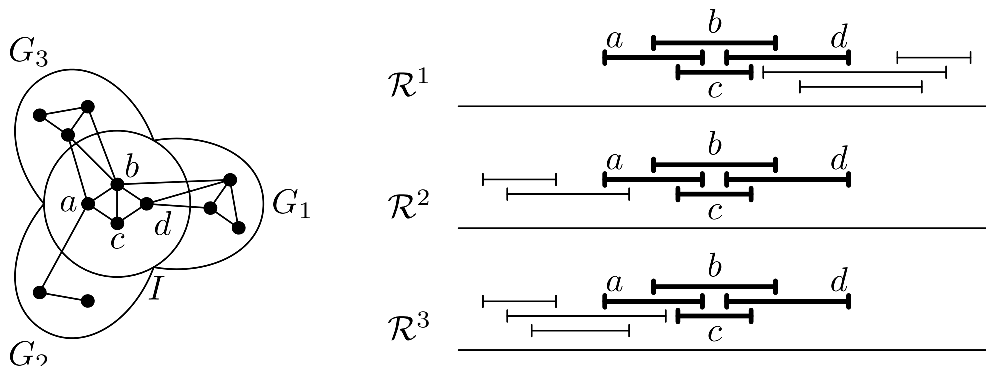

Fig. 4.1. *An example of three interval graphs with simultaneous representations assigning the same intervals to $I = \{a, b, c, d\}$.*

polynomially solvable for $k = 2$ [12], but REPEXT(CHOR) is NP-complete [15]. On the other hand for example for interval graphs, we get the following relations.

One can easily reduce REPEXT(INT) to SIMREP(INT) for $k = 2$ as follows. As the graph G_1, we put the input graph G, and as I we put the pre-drawn vertices $V(G')$. Now consider $\mathcal{R}'$, add a path going from left to right consisting of short intervals. This represents some interval graph which we put as G_2. The key is that G_2 fixes any representation of I to be topologically equivalent to $\mathcal{R}'$. So the question whether G_1 can be simultaneously represented with G_2 is equivalent to REPEXT(INT). The linear-time algorithm for REPEXT(INT) of Bläsius and Rutter [3] is based on this reduction.

The other relation is that if I is small enough, we can use the partial representation extension algorithm to test all possible representations of I. As a straightforward corollary of Theorem 1.1, the problem SIMREP(INT) is FPT in the size of I:

COROLLARY 4.1. *The problem* SIMREP(INT) *can be solved in* $\mathcal{O}((n+m)(2\ell)!)$ *where* $\ell = |I|$, $n = |V(G_1)| + \cdots + |V(G_k)|$, *and* $m = |E(G_1)| + \cdots + |E(G_k)|$.

Proof. There are $(2\ell)!$ different representations of I, given by all possible orderings of the 2ℓ endpoints. (Indeed, many of these orderings do not give a correct representation of I.) For each representation of $\mathcal{R}'$ of I, we test whether it is extendible to representations $\mathcal{R}^1, \dots, \mathcal{R}^k$. This can be done by running k times the algorithm of Theorem 1.1 which takes the total time $\mathcal{O}(n+m)$. Since we need to test $(2\ell)!$ possible representations, the total time is $\mathcal{O}((n+m)(2\ell)!)$.

For the correctness, if the algorithm succeeds in constructing $\mathcal{R}^1, \dots, \mathcal{R}^k$, the simultaneous representations problem is solvable. On the other hand, if the simultaneous representations problem is solvable, there exists some common representation of I and we test some representation which is topologically equivalent to it. Since solvability of partial representation extension depends only on the ordering of the endpoints, then the representation $\mathcal{R}'$ is extendible to $\mathcal{R}^1, \dots, \mathcal{R}^k$. □

Acknowledgements. We are very thankful to Pavol Hell for suggesting the PQ-trees approach, and to Martin Balko and Jiří Fiala for many comments concerning writing.

REFERENCES

[1] 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.
[2] S. BENZER, *On the topology of the genetic fine structure*, Proc. Nat. Acad. Sci. U.S.A., 45 (1959), pp. 1607–1620.
[3] T. BLÄSIUS AND I. RUTTER, *Simultaneous PQ-ordering with applications to constrained embedding problems*, in SODA '13, 2013.
[4] 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 (1976), pp. 335–379.
[5] S. CHAPLICK, R. FULEK, AND P. KLAVÍK, *Extending partial representations of circle graphs*, In preparation., (2013).
[6] D. G. CORNEIL, S. OLARIU, AND L. STEWART, *The LBFS structure and recognition of interval graphs*, SIAM Journal on Discrete Mathematics, 23 (2009), pp. 1905–1953.
[7] P.C. FISHBURN, *Interval orders and interval graphs: a study of partially ordered sets*, Wiley, 1985.
[8] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs.*, Pac. J. Math., 15 (1965), pp. 835–855.
[9] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, North-Holland Publishing Co., 2004.

[10] G. Hajós, *Über eine Art von Graphen*, Internationale Mathematische Nachrichten, 11 (1957), p. 65.

[11] K. R. Jampani and A. Lubiw, *Simultaneous interval graphs*, in Algorithms and Computation, vol. 6506 of Lecture Notes in Computer Science, 2010, pp. 206–217.

[12] ———, *The simultaneous representation problem for chordal, comparability and permutation graphs*, Journal of Graph Algortihms and Applications, 16 (2012), pp. 283–315.

[13] P. Klavík, J. Kratochvíl, T. Krawczyk, and B. Walczak, *Extending partial representations of function graphs and permutation graphs*, 7501 (2012), pp. 671–682.

[14] P. Klavík, J. Kratochvíl, Y. Otachi, I. Rutter, T. Saitoh, M. Saumell, and T. Vyskočil, *Extending partial representations of proper and unit interval graphs*, In preparation., (2012).

[15] P. Klavík, J. Kratochvíl, Y. Otachi, and T. Saitoh, *Extending partial representations of subclasses of chordal graphs*, 7676 (2012), pp. 444–454.

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

[17] T. A. McKee and F. R. McMorris, *Topics in Intersection Graph Theory*, SIAM Monographs on Discrete Mathematics and Applications, 1999.

[18] M. Patrignani, *On extending a partial straight-line drawing*, in Lecture Notes in Computer Science, vol. 3843, 2006, pp. 380–385.

[19] F. S. Roberts, *Indifference graphs*, Proof techniques in graph theory, (1969), pp. 139–146.

[20] ———, *Discrete Mathematical Models, with Applications to Social, Biological, and Environmental Problems*, Prentice-Hall, Englewood Cliffs, 1976.

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

[22] J. P. Spinrad, *Efficient Graph Representations*, Field Institute Monographs, 2003.

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

[24] The Beatles, *All you need is love*, (1967).

[25] W. T. Trotter, *New perspectives on interval orders and interval graphs*, in in Surveys in Combinatorics, Cambridge Univ. Press, 1997, pp. 237–286.