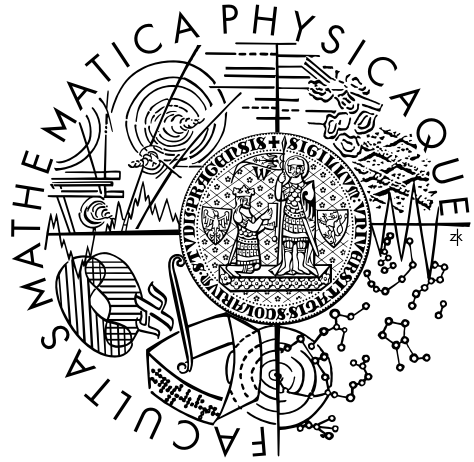
Charles University
Faculty of Mathematics and Physics



Doctoral Thesis

Computational Complexity in Graph Theory

Jan Kára

Department of Applied Mathematics
and
Institute for Theoretical Computer Science
Malostranské nám. 25
Prague, Czech Republic

2007

Declaration

This is to certify that I have written this thesis on my own and that the references include all the sources of information I have exploited. I authorize Charles University to lend this document to other institutions or individuals for the purpose of scholarly research.

Prague, March 7, 2007 .....

Jan Kára

Acknowledgement

I am very grateful to my advisor, Prof. RNDr. Jan Kratochvíl, CSc., for his guidance, advice, and other support during my years as a student at Charles University. I would also like to thank people of the departments, both here at Charles University and at Humboldt University in Berlin, for creating a friendly working atmosphere. In particular, I would like to thank Manuel Bodirsky for his enthusiasm and a very fruitful cooperation.

Contents

Chapter 1

Introduction

- What is the fastest way from Prague to Berlin?
- How many rooms do I need to satisfy all the reservations for a room in a hotel?
- Can we schedule jobs on a server, so that all the ordering requirements are met?
- Can we draw a circuit diagram so that no two components interconnections intersect?

These are examples of questions that we sometimes need to answer. Hence, it is natural to ask, how fast we can find the answer. Or formally said, what the time complexity of those problems is. In this thesis, we study the complexity of several problems of a "combinatorial nature". For some of them we design efficient algorithms, for others we show that they are unlikely to be solvable in a reasonable (polynomial) amount of time.

In the first part, we study Constraint Satisfaction Problems on infinite domain. It is a very general class of problems (for example our third question can be formulated in the framework of constraint satisfaction), which recently undergoes very fast development. We present general introduction to the state of the art in constraint satisfaction in Chapter 2. We continue by results for the most symmetric case of constraint satisfaction on infinite domains — equality constraint languages — in Chapter 3. Finally, in Chapter 4, we study a more practical class of languages — temporal constraint languages.

One thing our sample problems have in common is that they can be formulated in the language of graph theory. Sometimes, special properties of graphs arising in a particular problem can allow for an efficient solution. One special property a graph can have is, that it can be described in a geometric fashion. For example in our second sample problem, we can consider a graph in which vertices of the graph are reservations and two reservations are connected by an edge if and only

if they overlap. We look for an assignment of a number (room) to each vertex (reservation) so that two adjacent vertices get different numbers. This problem is generally hard to solve. However, in our case, it is possible to find a solution quickly, as our graphs are of a special kind. We can assign each reservation corresponding time-interval on a line. Again, we want to assign a number to each interval so that two intersecting intervals get different numbers. For intervals on a line, this problem is well-known to be solvable in polynomial time. Thus we see that a special structure of the graph and its geometric description can help us in designing an efficient solution.

In the second part of the thesis, we study graphs having these geometric descriptions. We show that all series-parallel graphs can be represented as contact graphs of segments in three directions in Chapter 6. We continue by designing a fixed parameter tractable algorithm for the Independent Set Problem in intersection graphs of segments in a fixed number of directions in Chapter 7. A problem of finding a balanced ordering of vertices, which is motivated by an effort to design an algorithm for nice drawings of a graph in the plane, is studied in Chapter 8.

1.1 Papers of the Author Related to the Thesis

This thesis is based on following papers of the author:

1. M. Bodirsky, C. Dangelmayr, and J. Kára: Representing Series-parallel Graphs as Intersection Graphs of Line Segments in Three Directions, In: *Innovative Applications of Information Technology for Developing World*, Kathmandu, Nepal, 2006.

2. M. Bodirsky and J. Kára: The complexity of equality constraint languages, *Proceedings of the International Computer Science Symposium in Russia (CSR'06)*, *LNCS* **3967** (2006), 114–126. Extended journal version is submitted.

3. M. Bodirsky and J. Kára, A Fast Algorithm and Lower Bound for Temporal Reasoning, submitted.

4. J. Kára and J. Kratochvíl: Fixed Parameter Tractability of Independent Set in Segment Intersection Graphs, In *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006,LNCS* **4169** (2006), 166–174.

5. J. Kára, J. Kratochvíl, and D. R. Wood, On the complexity of the balanced vertex ordering problem, *Proc. of 11th Annual International Conference, COCOON 2005*, *LNCS* **3595** (2005), 849–858. Extended journal version is submitted.

Part I

Constraint Satisfaction

Chapter 2

Introduction to Constraint Satisfaction

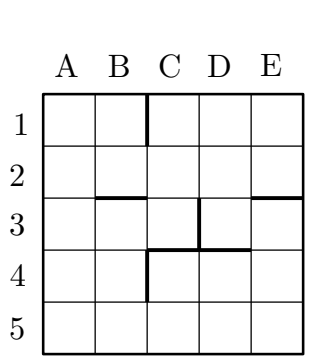
In this part, we present results about constraint satisfaction. This field experiences an intensive development carried out by a growing group of researchers. It is also interesting as a meeting point of several distinct fields of mathematics and computer science — algebra, combinatorics and logic. This synergy is extremely fruitful and sometimes brings unexpected connections and results. The area also attracts interest because many computational problems arising in artificial intelligence or generally computer science can be expressed as constraint satisfaction and optimization problems. In this chapter, we introduce basic definitions and theorems about constraint satisfaction.

2.1 Constraint Satisfaction Problems

The Constraint Satisfaction Problem (CSP) can be informally stated as follows: Given a finite set of variables, find an assignment of values to variables, subject to specified constraints. A wide variety of a real-world combinatorial problems such as planning [67], scheduling [95], frequency assignment problems [37], image processing [80], programming language analysis [81] or natural language understanding [2] can be formulated as a CSP. In database theory, constraint satisfaction has its applications in conjunctive-query evaluation [48, 68]. A CSP is interesting also from a theoretical point of view as it is quite expressive framework but in the natural parameterization all the problems seem to be either *tractable* (we use this term for problems solvable by an algorithm running in polynomial time) or NP-hard. We refer the reader to [45, 84] for general introduction to the complexity theory and definitions of complexity classes.

To demonstrate the versatility of a CSP we show several sample problems that can be modelled as a CSP:

Example 2.1. The n-queen problem: *Given n queens, find their placement*



List of words

an	da
ad	eddie
aid	li
and	nl
ape	pay
apple	plane
asp	re
bar	si
bay	

Figure 2.1: A crossword puzzle

on the $n \times n$ chessboard so that no two queens can attack each other.

Example 2.2. A crossword puzzle: *Given a crossword and a list of words, find an assignment of words to entries, so that each entry in a row/column is assigned a word that fits into given space and crossing words agree (see Figure 2.1).*

Example 2.3. A cryptography problem: *Given the pattern*

$$\begin{array}{cccc} S & E & N & D \\ M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

find an assignment of letters to digits so that the resulting sum is correct.

There are several ways how to formally state CSP. We begin with the algebraic formulation:

Definition 2.1. *An* instance of a constraint satisfaction problem *is a triple $(V, D, \mathcal{C})$ where*

- *V is a finite set of* variables*,*
- *D is a set (not necessarily finite) of* values *sometimes also called* domain*,*
- *$\mathcal{C}$ is a set of* constraints *$\{C_1, \ldots, C_q\}$, in which each constraint C_i is a pair (V_i, R_i), V_i being a list of variables of length m_i called the* constraint scope *(also denoted by $V(C_i)$) and R_i being an m_i-ary relation over the set D called the* constraint relation *(also denoted by $R(C_i)$).*

The question in the problem is, whether there exists a solution *to $(V, D, \mathcal{C})$, i. e., a function $s : V \to D$ such that for each $C_i, 1 \leq i \leq q$, the image of the constraint scope V_i ($s(V_i) := (s(v) \mid v \in V_i)$ is in the relation R_i.*

Any instance of CSP can be also easily rephrased to a logic form. We just consider equivalent predicates ρ_i on the set D instead of constraint relations R_i and ask whether a formula $\rho_1(V_1) \wedge \ldots \wedge \rho_q(V_q)$ is satisfiable. This form is often used in database theory as it closely corresponds to the conjunctive query evaluation [68], as is demonstrated by the following example:

Example 2.4. *A* relational database *is a finite collection of* tables*. A table is comprised of a* scheme *and an* instance*, where*

- *A scheme is a finite set of* attributes *and each attribute has a defined set of possible values (called* domain*).*
- *An instance is a finite set of* rows*, where each row is a mapping that assigns to each attribute from the scheme a value from its domain.*

A standard question in the relational databases is the Conjunctive query evaluation *problem [48, 68]. The problem is to decide, whether a given* conjunctive query *to the relational database, i. e., a query of the form $\rho_1 \wedge \ldots \wedge \rho_q$ with $\rho_i, 1 \leq i \leq q$ being an atomic formula, has a solution.*

Another equivalent reformulation of the constraint satisfaction problem is using homomorphisms [40, 74]. We need to introduce a few definitions before we can state the reformulation itself:

Definition 2.2. *A* relational signature τ *is a set of* relation symbols R_i*, each associated with a finite* arity m_i*. A* (relational) structure Γ *over the relational signature τ (also called τ-*structure *or* constraint language*) is a set D_Γ (the* domain*) together with a relation $R_i \subseteq D_\Gamma^{m_i}$ for each relation symbol R_i of arity m_i from τ. For simplicity, we use the same symbol for a relation symbol and the corresponding relation. If necessary, we write R^Γ to indicate that we are talking about the relation R belonging to the structure Γ.*

Definition 2.3. *Let Γ and Γ' be τ-structures. A* homomorphism *from Γ to Γ' is a function f from D_Γ to $D_{\Gamma'}$ such that for each m_i-ary relation symbol R_i in τ and each m_i-tuple $(a_1, \ldots, a_{m_i})$, if $(a_1, \ldots, a_{m_i}) \in R^\Gamma$, then $(f(a_1), \ldots, f(a_{m_i})) \in R^{\Gamma'}$. In this case, we say that the mapping f* preserves *the relation symbols R_i.*

Now, we are ready to state an alternative definition of the constraint satisfaction problem instance.

Definition 2.4. *An* instance of a constraint satisfaction problem *is a pair of relational structures (S, Γ) with the same relational signature, such that S is finite. The question in the problem is, whether there exists a homomorphism from S to Γ.*

The idea, why this definition is equivalent to Definition 2.1, is that tuples in relational structure S define constraint scopes and relational structure Γ defines corresponding constraint relations. Now, we show several more examples of various combinatorial problems expressed as a CSP. More problems can be found for example in [58, 74]. For the sake of brevity, we often use homomorphism formulation in our examples.

Example 2.5. Graph Coloring*: An instance of* Graph Coloring *problem [45] consists of a graph G and an integer k. The question is whether the vertices of G can be labelled with numbers $\{1, \ldots, k\}$ (called* colors*) in such a way that adjacent vertices are labelled with different numbers.*

This problem can be expressed as an CSP *instance (G, K_k), where K_k denotes a complete graph on k vertices.*

Example 2.6. Clique*: An instance of* Clique *problem [45] consists of a graph $G = (V, E)$ and an integer k. The question is whether there exists a subset of vertices $U \subseteq V$ of size k such that every two distinct vertices from U are connected by an edge.*

This problem can be expressed as a CSP *instance (K_k, G) provided G has no loops (i. e., vertex connected by an edge to itself).*

Example 2.7. System of Equations*: In this problem, we are given a system of q linear equations on some set of variables $x_1, \ldots, x_k$ assuming values from a field $\mathbb{F}$. The question is to find an assignment of values to variables so that each linear equation is satisfied.*

This problem can be expressed as a CSP *instance $(X, D, \mathcal{E})$ (here it is more convenient to use the algebraic formulation of a* CSP*), where $X = \{x_1, \ldots, x_k\}$, D is a set of elements of the field $\mathbb{F}$ and $\mathcal{E}$ is a set of constraints $\{E_1, \ldots, E_q\}$. Constraint scope of E_i is the set of variables present in the i-th equation and its constraint relation corresponds to possible solutions of this equation.*

Example 2.8. Satisfiability*: An instance of* Satisfiability *problem [45] is a formula ϕ, which is a conjunction of a set of* clauses*. Each clause is a disjunction of* literals *and each literal is either a variable or a negation of a variable. The question is whether there is an assignment of truth values to variables such that ϕ is true.*

This problem can be expressed as a CSP *instance $(X, \{0, 1\}, \mathcal{C})$ (we again use the algebraic formulation of a* CSP*), where X is a set of variables of the formula and for each clause C with variables $x_1, \ldots, x_k$, there is a constraint C' in $\mathcal{C}$ with the constraint scope $x_1, \ldots, x_k$ and a constraint relation $R = \{(y_1, \ldots, y_k) \in \{0, 1\}^k : C(y_1, \ldots, y_k)$ is true$\}$.*

Example 2.9. Graph Isomorphism*: In this problem, we are given two graphs $G = (V, E), H = (V', E')$ and we have to decide whether there exists a bijection f between V and V' such that $\{f(u), f(v)\} \in E'$ if and only if $\{u, v\} \in E$.*

We can express the problem as $((V, E, \overline{E}), (V', E', \overline{E'}))$*, where* $\overline{E} = \binom{V}{2} \setminus E$ *and* $\overline{E'} = \binom{V'}{2} \setminus E'$*.*

Example 2.10. HAMILTONIAN CIRCUIT*: An instance of* HAMILTONIAN CIRCUIT *problem consists of a graph* $G = (V, E)$*. The question is whether there is a subgraph of* G *isomorphic to a cycle having* $|V|$ *vertices. Recall that* a cycle *with* k *vertices,* C_k*, is a graph on a vertex set* $\{v_1, \ldots, v_k\}$ *with the edge set* $\{\{v_i, v_{i+1}\}, 1 \leq i < k\} \cup \{\{v_k, v_1\}\}$*.*

A formulation as a CSP *can be as follows:* $((V, C_{|V|}, \neq_V), (V, E, \neq_V))$*.* $\neq_V$ *is defined as* $\{(u, v) \in V^2 : u \neq v\}$*.*

2.2 Parameterizations of CSP

Given the versatility of CSP, it is no surprise that generally CSP is NP-hard. In particular, several problems shown above are well-known NP-hard problems [45]. Therefore it is natural to study parameterizations of the problem. There are two natural parameterizations of the problem, which can be informally stated as:

1. Restrict the structure of constraint scopes in the instance.

2. Restrict allowed types of constraint relations.

The first parameterization can be formally expressed as a structural requirement on the hypergraph defined by constraint scopes. The second parameterization is formally described by a set of allowed constraint relations. Note that in the homomorphism formulation, this corresponds to the restriction of the first or the second relational structure.

The case, when the structure of constraint scopes is fixed, has been studied in connection with databases [48, 68]. The complete classification (under a mild complexity theory assumption) of the complexity of problems in this parameterization has been recently given in [50]. The *core* of a finite relational structure Γ is a substructure Γ' of Γ such that every endomorphism of Γ' is an automorphism of Γ'. We refer the reader for the definition of tree-width to [89].

Theorem 2.1. *[50] Suppose that FPT*$\neq$*W[1]. Then for every recursively enumerable class* $\mathbb{C}$ *of structures of bounded arity holds: If there is* $w \geq 1$ *such that the core of every structure in* $\mathbb{C}$ *has tree-width at most* w*, then the problem of the existence of a homomorphism from* $\mathbb{C}$ *to some structure* D *is tractable. Otherwise the homomorphism problem is NP-complete.*

In this thesis, we consider the second parameterization. I. e., we restrict the set of allowed constraint relations. Formally, we use the following definition:

Definition 2.5. *Let* Γ *be a constraint language. A* constraint satisfaction problem over Γ, $\mathrm{CSP}(\Gamma)$, *is the subclass of a* CSP *such that every constraint relation of the instance of* $\mathrm{CSP}(\Gamma)$ *belongs to* Γ. *For convenience, we sometimes write just* $\mathrm{CSP}(\mathcal{R})$ *for a set of relations* $\mathcal{R}$ *and leave the domain implicit. We sometimes write* $\mathrm{CSP}(R)$ *for a relation* R *instead of* $\mathrm{CSP}(\{R\})$ *to simplify the notation.*

In the homomorphism formulation of CSP, the problem $\mathrm{CSP}(\Gamma)$ naturally corresponds to the problem with fixed target τ-structure Γ. Observe that for example problem System of Equations (Example 2.7) can be expressed as $\mathrm{CSP}(\Gamma)$ for a fixed Γ. On the other hand, it is not immediately obvious, how to express for instance the problem Coloring from Example 2.5 as $\mathrm{CSP}(\Gamma)$ for a fixed Γ. Certainly, the corresponding parameterized problem k-coloring (i. e., given a graph decide whether it can be properly colored by k colors) is easy to express. For the general problem Coloring, similarly as for problems such as Satisfiability (Example 2.8), the key is to use the fact that Γ can be infinite. So Γ can contain needed constraint relations for every number of colors, size of a clause, respectively. Existence of such infinite constraint languages introduces a slight ambiguity in the definition of tractability, which is commonly avoided by following definitions:

Definition 2.6. *A constraint language* Γ *is called* tractable *if for each finite* $\Gamma' \subseteq \Gamma$ *the problem* $\mathrm{CSP}(\Gamma')$ *is tractable. The language* Γ *is called* globally tractable *if* $\mathrm{CSP}(\Gamma)$ *is tractable.*

Intuition behind these definitions is that if we speak about simple tractability, our algorithm can depend exponentially on the sizes of constraint relations as they are fixed and finite. On the other hand, if we want a global tractability for an infinite language, our algorithm has to run in time polynomial in the sizes of constraint relations (up to constantly many exceptions).

It is easy to see that for finite languages, definitions are equivalent. For infinite languages, global tractability implies tractability but not vice versa (see [21]). Actually, it is an open problem whether tractability generally implies global tractability also for infinite languages. For most languages studied so far, this seems to be the case.

Another possible source of confusion is the case when the domain D of Γ is infinite. In such cases, instance $(V, D, \mathcal{C})$ of $\mathrm{CSP}(\Gamma)$ contains infinite D and therefore the size of the input to an algorithm could be considered infinite. To avoid this, we never allow infinite D to be a real part of the input and we rather deal with the domain implicitly.

Because of the versatility of the Satisfiability problem, several variations of the problem were introduced and studied. The best known are: Horn-sat, 3-sat, or Nae-sat [45].

Example 2.11. Horn-sat*: In this problem, we have to solve* Satisfiability *problem for a formula in which each clause has at most one positive literal.*

Example 2.12. 3-SAT*: This is a* SATISFIABILITY *problem for a formula in which each clause has exactly three literals.*

Example 2.13. NAE-SAT *(Not-all-equal-satisfiability): In this problem, we are given a formula ϕ with each clause containing exactly three literals. We are to find an assignment of truth values to variables such that in each clause, there is at least one literal evaluated as true and at least one literal evaluated as false.*

While the first of the above problems is well-known to be tractable [33] the other two problems are NP-complete [45]. Above modifications of SATISFIABILITY problem motivated a definition of a *generalized satisfaction problem*, which in our framework exactly corresponds to the constraint satisfaction problem over the domain $\{0, 1\}$. Schaefer has completely classified complexity of generalized satisfaction problems in his paper [91] in 1978. We present here his result using our definitions.

Theorem 2.2. *[91] Let Γ be a constraint language over the domain $\{0, 1\}$.* $\mathrm{CSP}(\Gamma)$ *is tractable if at least one of the following holds:*

- *Each relation in Γ is satisfied by setting all variables to the same constant.*
- *Each relation in Γ is definable by a CNF formula in which each clause contains at most one positive literal.*
- *Each relation in Γ is definable by a CNF formula in which each clause contains at most one negative literal.*
- *Each relation in Γ is definable by a CNF formula in which each clause contains at most two literals.*
- *Each relation in Γ is the set of solutions of a system of linear equations over the field GF(2).*

Otherwise $\mathrm{CSP}(\Gamma)$ *is NP-complete.*

Hence in our constraint satisfaction framework, this result can be viewed as a complete classification of complexity of a constraint satisfaction problem over the domain with two elements.

Success in the classification of the complexity of the generalized satisfaction problem has motivated attempts to obtain a similar classification for related problems. For example, P. Hell and J. Nešetřil have characterized the complexity of the following problem:

Definition 2.7. GRAPH H-COLORING*: In this problem, we are given an undirected graph G and ask whether there exists a homomorphism from G to H.*

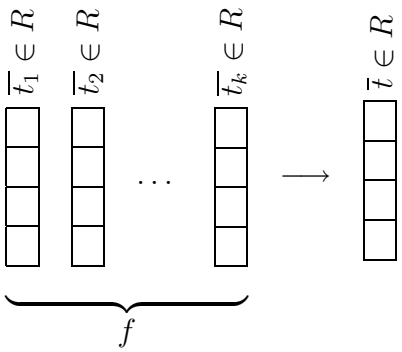


Figure 2.2: A k-ary polymorphism f

In their paper [54], they have showed that the problem is tractable if H is bipartite and NP-complete otherwise. The complexity of a similar problem for directed graphs is still open. In fact, in the light of the following theorem from [40] this is not a big surprise:

Theorem 2.3. *[40] Every* CSP(Γ) *with finite* Γ *is polynomial-time equivalent to* Graph H-coloring *for a suitable directed graph* H*.*

Finally, we would like to remark that recently, Bulatov managed to classify complexity of constraint satisfaction problems for all constraint languages over the domain of size three [20].

2.3 Polymorphisms and Expressibility

Generalizing the previous questions, we can ask, what the complexity of CSP(Γ) for given Γ is. Can we provide the characterization for all possible constraint languages? This has proved to be a quite complex task. Several methods have been used in determining complexity of some languages. Approach using general algebra and notion known as polymorphisms has turned out to be particularly fruitful.

Definition 2.8. *Let* D *be a set, and* O *be the set of* finitary operations *on* D*, i. e., functions from* D^k *to* D *for finite* k*'s. We say that a* k*-ary operation* $f \in O$ preserves *an* m*-ary relation* $R \subseteq D^m$ *if whenever* $R(x_1^i, \ldots, x_m^i)$ *holds in* Γ *for all* $1 \leq i \leq k$*, then* $R\big(f(x_1^1, \ldots, x_1^k), \ldots, f(x_m^1, \ldots, x_m^k)\big)$ *holds in* Γ*. If* f *preserves all relations of a constraint language* Γ*, we say that* f *is a* polymorphism *of* Γ*.*

See Figure 2.3 for a graphical explanation of a polymorphism.

Definition 2.9. *Let* Γ *be a constraint language over a domain* D*. Then* Pol(Γ) *denotes the set of all polymorphisms of* Γ*. Inversely, if* F *is a set of finitary*

operations, then $\mathrm{Inv}(F)$ *denotes the set of all relations that are preserved by all operations in* F*.*

In other words, a polymorphism f is a homomorphism from $\Gamma^k = \Gamma \times \cdots \times \Gamma$ to Γ, where $\Gamma_1 \times \Gamma_2$ is the (*categorical-* or *cross-*) *product* of the two relational τ-structures Γ_1 and Γ_2. Hence, the unary polymorphisms of Γ are the endomorphisms of Γ, and the unary bijective polymorphisms are the automorphisms of Γ.

To simplify the notation, we use $[k]$ to denote the set $\{1, \ldots, k\}$. To distinguish tuples of values from just plain elements of the domain or from names of variables, we use small Latin letters with a bar above as symbols for tuples — e. g. $\overline{t}$. We write $\overline{t}[i]$ for the i-th element of the tuple $\overline{t}$. If f is an l-ary function and $\overline{t_1}, \ldots, \overline{t_l}$ are k-tuples, we define $f(\overline{t_1}, \ldots, \overline{t_l})$ to be the k-tuple $(f(\overline{t_1}[1], \ldots, \overline{t_l}[1]), \ldots, f(\overline{t_1}[k], \ldots, \overline{t_l}[k]))$. When dealing with polymorphisms, we sometimes need the following definitions:

Definition 2.10. *A* k*-ary operation* $f : D^k \to D$ depends on the i-th argument*,* $i \in [k]$*, if there are* $x_1, \ldots, x_k \in D$ *and* $x_i' \in D$ *such that* $f(x_1, \ldots, x_k) \neq f(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k)$*. We say that* f *is* essentially l-ary*, if it depends on exactly* l *arguments.*

Now, we show an example to illustrate the notion of polymorphisms and also motivate further definitions:

Example 2.14. *Let* $R := \{(0,0), (1,2), (0,1), (1,1)\}$ *be a relation over the domain* $\{0,1,2\}$*. A ternary operation*

$$sp(x,y,z) := \begin{cases} y & \text{if } x = y \text{ or } y = z \text{ or } x = z \\ 1 & \text{otherwise.} \end{cases}$$

preserves R*. Let* $\overline{t_1}, \overline{t_2}, \overline{t_3}$ *be pairs from* R*. If the first case applies for both triples* $(\overline{t_1}[1], \overline{t_2}[1], \overline{t_3}[1])$*,* $(\overline{t_1}[2], \overline{t_2}[2], \overline{t_3}[2])$*, we obtain* $\overline{t_2}$*, which is from* R*. For the first triple, the second case never applies since* 2 *never occurs in the first triple. Hence, we have to check only the case that for the second triple the second case is applied. But the resulting pair is then either* $(0,1)$ *or* $(1,1)$*, which are both in* R*.*

In $\mathrm{CSP}(R)$*, we can also express other constraint relations. For example we can force relation* $R' := \{(0,0), (0,1), (0,2), (1,1)\}$ *on a pair of variables* (x,y) *by putting* R *on* (x,z) *and* (z,y)*, where* z *is some auxiliary variable. It is easy to check that* sp *also preserves* R'*. Actually, this is not just by chance, as we show in several following paragraphs.*

Definition 2.11. *A* k*-ary relation* R *can be* expressed *in a constraint language* Γ *over* D *if there exists an instance* $(V, D, \mathcal{C})$ *of* $\mathrm{CSP}(\Gamma)$*, and a list,* L*, of* k *variables, such that the solutions of* $(V, D, \mathcal{C})$ *restricted to* L *form the relation* R*. The set of all relations which can be expressed in* Γ *is called the* expressive power of Γ*,* $\langle\Gamma\rangle$*.*

A set of relations $\mathcal{R}$ that is closed under expression (i. e., anything that can be expressed using relations in $\mathcal{R}$ is in $\mathcal{R}$) is called a *relational clone*. Similarly, a set $\mathcal{F}$ of finitary operations over D that contains all projections (*projection* is a mapping $D^k \to D$ such that $f(x_1, \ldots, x_k) = x_i$ for all $x_1, \ldots, x_k \in D$ and some $i \in \{1, \ldots, k\}$) and is closed under compositions (a *composition* of $f : D^k \to D$ with $g_1, \ldots, g_k : D^l \to D$ is a function $h : D^l \to D$ defined as $h(x_1, \ldots, x_l) := f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l)))$ is called a *(polymorphism) clone*. In case D is infinite, we also require a clone $\mathcal{F}$ to be locally closed [98]. We say that $f : D^k \to D$ is *interpolated* by a set of operations $\mathcal{F}' \subseteq \mathcal{F}$ if for all finite $D' \subseteq D$ there is $f' \in \mathcal{F}'$ such that $f'(x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$ for all $x_1, \ldots, x_k \in D'$. The set of operations $\mathcal{F}$ is *locally closed* if any operation that is interpolated by operations from $\mathcal{F}$ is already in $\mathcal{F}$. Clones have been intensively studied in general algebra [90, 98] and the results provide essential tools for classification of complexity of constraint satisfaction problems. See for example Section 2.4.

The following standard lemma directly follows from the definitions:

Lemma 2.4. *For any constraint language Γ and any relation R from the expressive power of Γ, the problem $\mathrm{CSP}(\Gamma \cup \{R\})$ is reducible in polynomial time to the problem $\mathrm{CSP}(\Gamma)$.*

The fact that if f is a polymorphism for two constraints C_1, C_2, then it is a polymorphism for their conjunction give us another handy standard result:

Lemma 2.5. *For any constraint language Γ, any relation R from the expressive power of Γ and any polymorphism f of Γ, the relation R is preserved by f.*

This trivial result can be extended to a much more interesting theorem, that can be found for example in [87].

Theorem 2.6. *[87] For every constraint language Γ over a finite domain, the equality $\mathrm{Inv}(\mathrm{Pol}(\Gamma)) = \langle\Gamma\rangle$ holds.*

Theorem 2.6 together with Lemma 2.4 also gives us the following complexity result:

Theorem 2.7. *[58] For all constraint languages Γ, Γ' over a finite domain, if $\mathrm{Pol}(\Gamma) \subseteq \mathrm{Pol}(\Gamma')$, then $\mathrm{CSP}(\Gamma')$ is reducible in polynomial time to $\mathrm{CSP}(\Gamma)$.*

This result can also be viewed as follows: The set of polymorphisms of a constraint language Γ determines complexity of $\mathrm{CSP}(\Gamma)$ up to a polynomial time reduction. Actually, this result is not limited only to finite domains but also holds in some infinite cases as we describe below in Theorem 2.9.

Now, we turn our attention to the logical description of the expressive power of a constraint language. Let τ be a relational signature. A formula over τ (τ-formula) is called *primitive positive*, if it has the form

$$\exists x_1, \ldots, x_k : \phi_1 \wedge \ldots \wedge \phi_l,$$

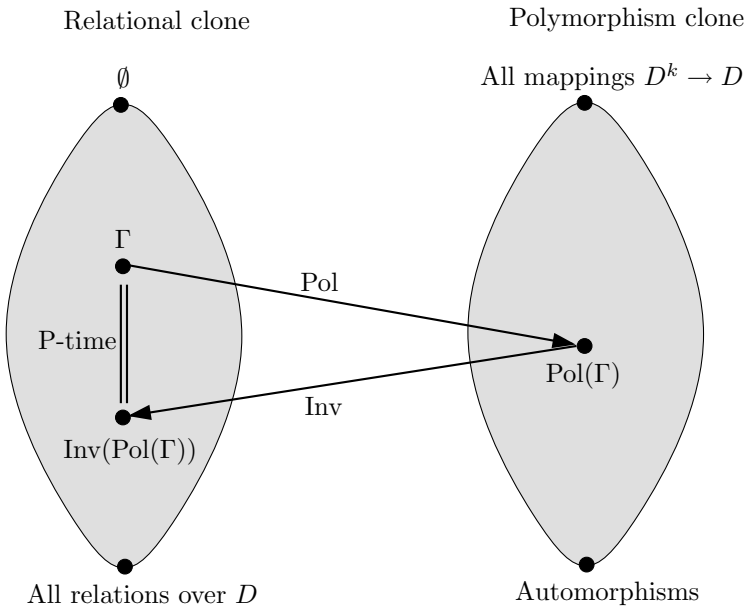


Figure 2.3: Operations Inv and Pol.

where $\phi_1, \ldots, \phi_l$ are atomic τ-formulae that may contain free variables and existentially quantified variables from $x_1, \ldots, x_k$. The atomic τ-formula may also be of the form $x = y$. A formula is called *existentially positive*, if it is a disjunctive combination of primitive positive formulae (i. e., it is a first order formula without negations and universal quantification). Every formula with k free variables naturally defines a k-ary relation over any τ-structure Γ. It is not hard to verify that relations which can be defined by primitive positive formulae over Γ are precisely those relations that can be expressed in Γ.

Generalization of Theorem 2.6 to infinite domains has been found in [11, 18]. First, we introduce a well-established notion from model theory:

Definition 2.12. *A constraint language Γ over a countable domain is called* ω-categorical *if all countable models of its first order theory are isomorphic to Γ.*

ω-categoricity can be characterized also differently:

Theorem 2.8. *[23] Let Γ be a constraint language over a domain D. Then the following is equivalent:*

- *Γ is ω-categorical.*
- *The automorphism group of Γ is oligomorphic (i. e., it has only finitely many orbits of k-tuples of elements from D for any $k \geq 1$).*
- *Every k-ary first-order definable relation in Γ is a union of a finite number of orbits of k-tuples of the automorphism group of Γ.*

Examples of ω-categorical languages are for example constraint languages over rational numbers with constraint relations defined by first order formulae using $=$ and $<$ as atomic relations. So $(\mathbb{Q}, \{<, \leq, =\})$ is ω-categorical. Automorphisms of such languages are certainly all increasing mappings $\mathbb{Q} \to \mathbb{Q}$. Hence, it follows that for any $k \geq 1$, there are at most $1 \cdot 3 \cdot \ldots \cdot (2k-1)$ orbits of k-tuples from $\mathbb{Q}$ in the automorphisms group. To demonstrate last characterization in Theorem 2.8, consider relation $R(x, y, z) := \{(x, y, z) \mid (x < y) \vee (x < z)\}$. Such relation is composed of seven orbits of triples. They are characterized by inequalities $x < y < z$, $x < z < y$, $y < x < z$, $z < x < y$, $x = y < z$, $x = z < y$, and $x < y = z$.

Now, we are ready to present result for infinite domains:

Theorem 2.9. *[11, 18] Let Γ be an ω-categorical constraint language. Then relations preserved by all polymorphisms of Γ are precisely those that have a primitive positive definition in Γ. I. e.,* $\mathrm{Inv}(\mathrm{Pol}(\Gamma)) = \langle\Gamma\rangle$.

This theorem theorem is fundamental to our further research of constraint satisfaction over infinite domains as it allows us to use the powerful machinery of polymorphisms and algebra also in the case of ω-categorical constraint languages.

Also this theorem allows us to extend a standard corollary of Theorem 2.6 to ω-categorical languages:

Corollary 2.10. *Let Γ be an ω-categorical constraint language and let R be a relation that does not have a primitive positive definition in Γ. Then there is $f \in \mathrm{Pol}(\Gamma)$ such that f does not preserve R.*

A similar counterpart on the side of polymorphisms holds even for arbitrary (not only ω-categorical) relational structures [98].

Proposition 2.11. *Let $\mathcal{F}$ be a clone over D and let g be a finitary operation over D such that $g \notin \mathcal{F}$. Then there is $R \in \mathrm{Inv}(\mathcal{F})$ such that R is not preserved by g.*

To slightly practice work with polymorphisms of a constraint language, we present here a useful lemma:

Lemma 2.12. *Let Γ be an ω-categorical constraint language and R be a k-ary relation that is a union of l orbits of k-tuples of* $\mathrm{Aut}(\Gamma)$. *If $R \notin \langle\Gamma\rangle$, then there is an at most l-ary $f \in \mathrm{Pol}(\Gamma)$ such that f is not preserving R.*

Proof. Recall that by Theorem 2.8, every relation first order definable in Γ is a union of finitely many orbits of $\mathrm{Aut}(\Gamma)$. If $R \notin \langle\Gamma\rangle$, Corollary 2.10 asserts that there is a polymorphism g that does not preserve R. If g is at most l-ary, we are done. Otherwise let m be the arity of g and $\overline{t_1}, \ldots, \overline{t_m}$ be tuples from R such that $g(\overline{t_1}, \ldots, \overline{t_m})$ is not in R. As R is a union of l orbits of $\mathrm{Aut}(\Gamma)$ and $l < m$, there are some tuples in $\overline{t_1}, \ldots, \overline{t_m}$ that are from the same orbit. In particular,

we can without loss of generality assume that for each tuple $\overline{t_{l+1}}, \ldots, \overline{t_m}$, there is some tuple from $\overline{t_1}, \ldots, \overline{t_l}$ that is in the same orbit. Let o_i be the number of the tuple that is in the same orbit as $\overline{t_i}$ for $l+1 \leq i \leq m$. Clearly, there are also automorphisms $\alpha_{l+1}, \ldots, \alpha_m$ of Γ such that $\alpha_i(\overline{t_{o_i}}) = \overline{t_i}$. Therefore the operation f defined as $f(x_1, \ldots, x_l) := g(x_1, \ldots, x_l, \alpha_{l+1}(x_{o_{l+1}}), \ldots, \alpha_m(x_{o_m}))$ also violates R and it is l-ary. □

We conclude this section with another example of a standard local generation argument.

Lemma 2.13. *Let Γ be such that $\mathrm{Aut}(\Gamma)$ has one orbit of 2-sets. If Γ has a non-injective endomorphism f, then Γ also has a constant endomorphism.*

Proof. Let f be an endomorphism of Γ such that $f(b) = f(b')$ for two distinct values b, b' from D. Let $a_1, a_2, \ldots$ be an enumeration of D. We construct an infinite sequence of endomorphisms $e_1, e_2, \ldots$, where e_i is an endomorphism that maps the values $a_1, \ldots, a_i$ to a_1. This suffices, since by local closure, the mapping defined by $e(x) := a_1$ for all x is an endomorphism of Γ.

For e_1, we take the identity map, which clearly is an endomorphism with the desired properties. To define e_i for $i \geq 2$, let α be an automorphism of Γ that maps $\{a_1, e_{i-1}(a_i)\}$ to $\{b, b'\}$ (such automorphism exists because of the assumption on $\mathrm{Aut}(\Gamma)$). Then the endomorphism $f(\alpha(e_{i-1}(x)))$ is constant on $a_1, \ldots, a_i$ (recall that $a_1 = e_{i-1}(a_1) = \ldots = e_{i-1}(a_{i-1})$). There is also an automorphism α' that maps $f(b)$ to a_1. Then $e_i(x) := \alpha'(f(\alpha(e_{i-1}(x))))$ is an endomorphism with the desired properties. □

2.4 Results for Finite Domains

To demonstrate the strength of the algebraic approach to constraint satisfaction we present here a few classic results for finite domains.

Let D be a finite domain in this section. Let f be a k-ary operation from $D^k \to D$.

- f is called *idempotent* if $f(d, \ldots, d) = d$ for all $d \in D$.
- f is called *essentially unary* (note that this definition coincides with the definition of essentially l-ary for $l = 1$) if there exists $i \in [k]$ and a non-constant $g : D \to D$ such that $f(d_1, \ldots, d_k) = g(d_i)$ for all $(d_1, \ldots, d_k) \in D^k$.
- f is called a *semiprojection* if $k \geq 3$, f is not a projection and there is $i \in [k]$ such that for all $d_1, \ldots, d_k \in D$ satisfying $|\{d_1, \ldots, d_k\}| < k$ it holds that $f(d_1, \ldots, d_k) = d_i$.

- f is called a *majority operation* if $k = 3$ and $f(d', d, d) = f(d, d', d) = f(d, d, d')$ for all $d, d' \in D$.
- f is called an *affine operation* if $k = 3$ and $f(d_1, d_2, d_3) = d_1 + d_2 - d_3$ for all $d_1, d_2, d_3 \in D$ where $(D, +, -)$ is an Abelian group [3].

The following theorem has been shown using results from general algebra [90, 98]:

Theorem 2.14. *[21] Let Γ be a constraint language such that it does not have a non-injective endomorphism. Then either*

1. $\mathrm{Pol}(\Gamma)$ *contains essentially unary operations only, or*
2. $\mathrm{Pol}(\Gamma)$ *contains an operation that is*
   (a) *a constant operation, or*
   (b) *a majority operation, or*
   (c) *an idempotent binary operation (which is not a projection), or*
   (d) *an affine operation, or*
   (e) *a semiprojection.*

Since for finite domains, we can always assume that Γ does not have a non-injective endomorphism (otherwise $\mathrm{CSP}(\Gamma)$ can be reduced in polynomial time to $\mathrm{CSP}(\Gamma')$ such that Γ' does not have a non-injective endomorphism [21]), Theorem 2.14 determines the complexity of a large class of constraint satisfaction problems. For example in case 1, it follows that the problem is NP-complete. On the other hand in cases 2a, 2b, 2d, the problem $\mathrm{CSP}(\Gamma)$ is shown to be tractable and hence the only unsettled cases are 2c and 2e.

Further results about constraint satisfaction problems for finite domains can be found for example in [20, 21, 29]. In [21], a conjecture about the complexity of constraint satisfaction problems over a finite domain has been posed. We state here its informal variant. Let I be an instance of $\mathrm{CSP}(\Gamma)$, $\overline{t_1}, \ldots, \overline{t_k}$ some l-tuples of variables from I and R a relation over the same domain as I. For a solution f to I, we can define a k-tuple $\overline{t_f}$ so that $\overline{t_f}[i] = 1$ if R holds on $f(\overline{t_i})$ and $\overline{t_f}[i] = 0$ otherwise. In this way, the instance I *simulates* some k-ary relation $S = \{\overline{t_f} \mid f \text{ a solution to } I\}$ over $\{0, 1\}$.

Conjecture 2.1. *If* $\mathrm{CSP}(\Gamma)$ *is not able to simulate any of the NP-complete cases of generalized satisfiability problem over a Boolean domain, then the problem* $\mathrm{CSP}(\Gamma)$ *is tractable.*

We refer the reader for a more formal formulation to [21].

Chapter 3

Equality Constraint Languages

In this chapter, we start a presentation of our results for constraint satisfaction problems over countably infinite domains. We start the research by studying the most symmetric languages. I. e., those that are preserved by all permutations of the domain. Actually, we define them as languages where each constraint relation can be defined by an $=$-formula (i. e., a Boolean combination of atoms of the form $x = y$) in Section 3.1 and show that the symmetry is an equivalent characterization. We call such languages *equality constraint languages*. The material of this chapter is mostly based on papers [15, 16] of the author.

First, let us show two examples of equality constraint languages:

Example 3.1. *Let* Γ *be a relational structure* $(\mathbb{N}, \{=, \neq\})$. *Then* $\mathrm{CSP}(\Gamma)$ *is the computational problem to determine for a given set of equality or inequality constraints on a finite set of variables whether the variables can be mapped to the natural numbers such that variables* x, y *with a constraint* $x = y$ *are mapped to the same value and variables* x, y *with a constraint* $x \neq y$ *are mapped to distinct values.*

Note that Example 3.1 is a tractable constraint language as we can consider the undirected graph on the variables of an instance I of $\mathrm{CSP}(\Gamma)$, where two variables x and y are joined if and only if there is a constraint $x = y$ in I. Then it is easy to see that I does not have a solution if and only if it contains an inequality-constraint $x \neq y$ such that y is reachable from x in the graph defined above. Clearly, such a reachability test can be performed in linear time in the size of the input.

Example 3.2. *Let* Γ *be the relational structure* $(\mathbb{N}, \{S\})$, *where* S *is the ternary relation* $S := \{ (x_1, x_2, x_3) \in \mathbb{N}^3 \mid (x_1 = x_2 \wedge x_2 \neq x_3) \vee (x_1 \neq x_2 \wedge x_2 = x_3)\}$.

For this language, we show that $\mathrm{CSP}(\Gamma)$ is NP-complete in Section 3.3.

The main result of this chapter is that we determine the complexity of constraint satisfaction problems for all equality constraint languages. In particular, we show that each problem is either tractable or NP-complete. The containment

in NP is easy to see: A non-deterministic algorithm can guess which variables in an instance I denote the same element in Γ and can verify whether there is a corresponding solution for I. To prove that certain equality constraint languages are NP-hard (Section 3.3), we apply the algebraic approach to constraint satisfaction.

For the tractable equality constraint languages, we present an algorithm that outperforms other algorithms that are on resolution or based on establishing relational consistency. Moreover, if the constraint relations are represented by formulae in DNF, then our algorithm is the first polynomial-time algorithm. We also discuss the complexity of the so-called meta problem of constraint satisfaction complexity, i. e., the question whether a given finite equality constraint language is tractable or not.

3.1 Equality Constraint Languages and Their Representations

We start with a formal definition of an equality constraint language:

Definition 3.1. *An equality constraint language is a constraint language* $\Gamma = (D, \{R_1, R_2, \dots\})$ *on a countably infinite domain* D *where all relations can be defined with Boolean combinations of the equality relation.*

Clearly, every permutation of D preserves all relations $R_1, R_2, \dots$ and hence equality constraint languages have a highly transitive automorphism group.

Conversely, suppose that R is a k-ary relation that is preserved by all permutations of D. Such a relation is a union of finitely many orbits of k-tuples with respect to the permutation group that contains all permutations. It is easy to see that the orbits of k-tuples in R can be described by a conjunction of equality and inequality relations. Hence, every relational structure that is preserved by all permutations is an equality constraint language.

For the rest of this chapter $\Gamma = (D, \{R_1, R_2, \dots\})$ always denotes an equality constraint language. There are different natural ways to represent the relations $R_1, R_2, \dots$ in instances of the CSP. If the constraint language is finite, the choice of the representation clearly does not affect the computational complexity. However, if the constraints of the language are specified in input, the representation of the constraint relations matters. The representation that dominated in the literature on the CSP over finite domains is the representation of a relation by the set of its tuples. However, note that there are other natural possibilities for the choice of the representation. In [28], for constraint satisfaction problems over a Boolean domain, representations of constraint relations by formulae in disjunctive normal form (DNF) have been studied as well.

We study two natural ways to represent relations in equality constraint languages, which are analogous to the two mentioned representations over a finite

domain: the representation by sets of tuples and the representation by a formulae in DNF. We show that in both cases the tractable constraint languages are globally tractable.

The first representation by *sets of tuples* is the closest analogue to representations of relations over a finite domain by the set of tuples in the relation. Clearly, a non-empty relation from an equality constraint language contains an infinite number of tuples, and therefore we cannot use such representations for equality constraint languages. However, as we have seen before, every k-ary relation in Γ is a union of a finite number of orbits of k-tuples of the automorphism group of Γ. Let $\overline{s}$ be a k-tuple from one of these orbits. Let ρ be the equivalence relation on the set $\{1, \ldots, k\}$ that contains those pairs $\{i, j\}$ where $s_i = s_j$. Clearly, all tuples of the same orbit lead to the same equivalence relation ρ. Hence, every k-ary relation R in Γ corresponds uniquely to a set of equivalence relations on $\{1, \ldots, k\}$, which we call the *representation* of R. Sometimes, we identify a relation R from Γ with its representation. For example, we freely write $\rho \in R$ if ρ is an equivalence relation from the representation of R. Let $|R|$ denote the number of orbits of k-tuples contained in R (i. e., the number of equivalence relations in the representation of R).

For algorithmic purposes, it is convenient to not use equivalence relations explicitly, but instead take from each orbit of R one tuple as its representant. Hence, in the algorithm, we rather speak about the representation of a relation by a *set of tuples* instead of the representation by a *set of equivalence relations*.

Example 3.3. *Consider again the ternary relation $S(x_1, x_2, x_3)$ from Example 3.2 that is defined by the $=$-formula $(x_1 = x_2 \wedge x_2 \neq x_3) \vee (x_1 \neq x_2 \wedge x_2 = x_3)$. The representation of S by equivalence relations consists of two equivalence relations, each containing exactly two equivalence classes. An example of a representation of S by a set of tuples is $\{(0, 0, 1), (0, 1, 1)\}$.*

Note that hardness results that we present in Section 3.3 hold independently from the representation. The algorithmic results in Section 3.4 do depend on the representation. Clearly, the meta problem of deciding whether a given constraint language is tractable discussed in Section 3.5 may also depend on the way how the relations of the language are represented.

3.2 Intersection-closed Relations

In this section, we study an important property that an equality constraint language might have.

Definition 3.2. *Let ρ and ρ' be equivalence relations on a set X. We say that ρ is* finer *than ρ', and write $\rho \subseteq \rho'$, if $\rho(x, y)$ implies $\rho'(x, y)$ for each $x, y \in X$. We also say that ρ' is* coarser *than ρ in this case. The* intersection *of two equivalence*

relations ρ and ρ', denoted by $\rho \cap \rho'$, is the equivalence relation σ such that $\sigma(x, y)$ if and only if $\rho(x, y)$ and $\rho'(x, y)$. Finally, let $c(\rho)$ denote the number of equivalence classes in ρ.

Lemma 3.1. *For a k-ary relation R in an equality constraint language on a countable set D, the following conditions are equivalent.*

1. *R is preserved by every injective binary operation on D;*
2. *R is preserved by an injective binary operation on D;*
3. *R is preserved by a binary operation f such that there are two k-element subsets S_1, S_2 of the domain such that f restricted to $S_1 \times S_2$ is injective;*
4. *The representation of R is* closed under intersections*, i. e., $\rho \cap \rho' \in R$ for all equivalence relations $\rho, \rho' \in R$;*
5. *R can be defined by an $=$-formula that is Horn, i. e., an $=$-formula in CNF where each clause contains at most one expression of the form $x = y$.*

Proof. The implication from 1 to 2 and from 2 to 3 is immediate. Let ρ and ρ' be two equivalence relations from the representation of R. Pick two k-tuples $\overline{s}$ and $\overline{s'}$ in R that lie in the orbits that are described by ρ and ρ'. Now, let f be a binary operation of D that is injective on its restriction to $S_1 \times S_2$ for two k-element subsets S_1, S_2. Let α_1 and α_2 be permutations of D that map the values of the k-tuples $\overline{s}$ and $\overline{s'}$ to S_1 and S_2, respectively. Then by the injectivity of f on $S_1 \times S_2$, the k-tuple $\overline{s''} := f(\alpha_1(\overline{s}), \alpha_2(\overline{s'}))$ satisfies $\overline{s''}[i] = \overline{s''}[j]$ if and only if $\rho(i, j)$ and $\rho'(i, j)$. Hence, we found a tuple in R that lies in the orbit that is described by $\rho \cap \rho'$, which is therefore also contained in the representation of R, and therefore 3 implies 4.

Every injection of D^2 into D preserves every relation with an intersection-closed representation, because it maps two tuples that correspond to equivalence relations ρ and ρ' to a tuple that corresponds to $\rho \cap \rho'$. We thus proved that 4 implies 1.

To show that 2 implies 5, let R be preserved by a binary injective operation f, and let $\phi(x_1, \ldots, x_k)$ be a formula in a CNF that defines R. By a CNF we mean that the formula ϕ is a conjunction of *clauses*, each clause is a disjunction of *literals*, and each literal is either *positive* (i. e., of the form $x = y$) or *negative* (i. e., of the form $x \neq y$). The formula ϕ is called in *reduced form* if it does not contain a clause or a literal such that removing this clause or literal from ϕ creates an equivalent formula. It is clear that every formula is equivalent to a reduced formula and hence we can assume that ϕ is in a reduced form. Suppose for contradiction that ϕ is not Horn, i. e., there exists a clause ϕ_α of ϕ which contains two equalities $x_i = x_j$ and $x_{i'} = x_{j'}$. Construct ϕ' from ϕ by removing the equality $x_i = x_j$, and ϕ'' by removing $x_{i'} = x_{j'}$. There exist $\overline{a}, \overline{b} \in X^n$ such

that $\phi(\overline{a})$ but not $\phi'(\overline{a})$, and $\phi(\overline{b})$ but not $\phi''(\overline{b})$. Clearly, $\overline{a}[i] = \overline{a}[j]$, $\overline{a}[i'] \neq \overline{a}[j']$, $\overline{b}[i] \neq \overline{b}[j]$, and $\overline{b}[i'] = \overline{b}[j']$. Set $\overline{c} := f(\overline{a}, \overline{b})$. Then $\overline{c}[i] \neq \overline{c}[j]$, $\overline{c}[i'] \neq \overline{c}[j']$, and in fact $\phi(\overline{c})$ does not hold. Hence, R is not preserved by f, a contradiction.

We finally show that 5 implies 2. Let R be a relation that can be defined by a Horn =-formula, and let f be a binary injective operation. Let $\overline{s}$ and $\overline{t}$ be two tuples in R. We verify that $f(\overline{s}, \overline{t})$ satisfies each clause C. First, consider the case that $\overline{s}$ or $\overline{t}$ satisfies a negative literal. Then $f(\overline{s}, \overline{t})$ satisfies this literal as well, by injectivity of f. In the other case, both $\overline{s}$ and $\overline{t}$ do not satisfy all negative literals. Hence, there must be a positive literal, and $\overline{s}$ and $\overline{t}$ must satisfy this literal. But then $f(\overline{s}, \overline{t})$ satisfies this literal as well. Therefore, f preserves R. □

Corollary 3.2. *An operation f generates an injective binary operation g if and only if every equality constraint relation that is preserved by f is intersection-closed.*

Proof. If f generates an injective binary operation g, then every relation R that is preserved by f is also preserved by g, and Lemma 3.1 shows that R is intersection closed. Conversely, if every equality constraint relation R preserved by f is intersection closed, we claim that f generates all injective binary operations. Suppose the contrary. By Proposition 2.11, there is an equality constraint relation R that is preserved by f but not by an injective binary operation g. Another application of Lemma 3.1 shows that R cannot be intersection closed, contradicting the assumption. □

3.3 A Generic Hardness Proof

In this section, we prove that every equality constraint language without a constant unary or an injective binary polymorphism has NP-hard constraint satisfaction problem. We start by noting that the automorphism group of every equality constraint language has one orbit of 2-sets and therefore Lemma 2.13 applies to all such languages.

Lemma 3.3. *If Γ does not have a constant endomorphism, then there is a primitive positive definition of the relation $x \neq y$ in Γ.*

Proof. Suppose Γ has a k-ary polymorphism f that does not preserve $\neq$, i. e., there are k-tuples $\overline{u}$ and $\overline{v}$ such that $\overline{u}[i] \neq \overline{v}[i]$ for all $i \in [k]$, but $f(\overline{u}[1], \ldots, \overline{u}[k])$ is equal to $f(\overline{v}[1], \ldots, \overline{v}[k])$. Let $\alpha_2, \ldots, \alpha_k$ be permutations of D such that α_i maps $\overline{u}[1]$ to $\overline{u}[i]$ and $\overline{v}[1]$ to $\overline{v}[i]$. Then the endomorphism g defined as $g(x) := f(x, \alpha_2(x), \ldots, \alpha_k(x))$ is not injective, because $g(\overline{u}[1]) = f(\overline{u}[1], \ldots, \overline{u}[k]) = f(\overline{v}[1], \ldots, \overline{v}[k]) = g(\overline{v}[1])$, and by Lemma 2.13 locally generates a constant, in contradiction to the assumptions. Hence, every polymorphism of Γ preserves $\neq$ and by Theorem 2.9 the relation $\neq$ has a primitive positive definition in Γ. □

Now comes the central argument.

Theorem 3.4. *Let f be a binary operation that depends on both arguments. Then f together with all permutations locally generates either a constant unary operation or a binary injective operation.*

Proof. Suppose that f does not locally generate a constant operation. We want to use Corollary 3.2 and show that every equality constraint relation R that is preserved by f is intersection closed, which implies that f locally generates a binary injective polymorphism. Suppose for contradiction that R is an n-ary equality constraint relation, $n \geq 2$, that is closed under f but not intersection closed, i. e., there are two equivalence relations ρ and ρ' in R such that $\rho \cap \rho'$ is not in R. Choose ρ and ρ' such that $(c(\rho), c(\rho'))$ is lexicographically maximal. Let $\overline{s}$ and $\overline{t}$ be n-tuples of D that define the equivalence relations ρ and ρ'. Because ρ is not finer than ρ' we can find indices p and q such that $\overline{s}[p] = \overline{s}[q]$, $\overline{t}[p] \neq \overline{t}[q]$. Let r be the number of equivalence classes of ρ that are contained in the equivalence class of p in ρ'. Choose p and q such that r is minimal.

Consider $2n-1$ distinct elements $a_1, \ldots, a_{2n-1}$ from D. By the infinite pigeonhole principle, there is an infinite subset S_1 of D such that $f(a_1, b) = f(a_1, b')$ for all $b, b' \in S_1$, or $f(a_1, b) \neq f(a_1, b')$ for all $b, b' \in S_1$. We apply the same argument to a_2 instead of a_1, and S_1 instead of D, and obtain an infinite subset S_2 of S_1. The argument can be iterated to obtain an infinite subset S_{2n-1} such that for all $a \in \{a_1, \ldots, a_{2n-1}\}$ we either have $f(a, b) \neq f(a, b')$ for all $b, b' \in S_{2n-1}$, or $f(a, b) = f(a, b')$ for all $b, b' \in S_{2n-1}$. Then there is also an n-element subset A of $\{a_1, \ldots, a_{2n-1}\}$ and an n-element subset B of S_{2n-1} such that either $f(a, b) \neq f(a, b')$ for all $a \in A$ and $b, b' \in B$, or $f(a, b) = f(a, b')$ for all $a \in A$ and $b, b' \in B$. Note that in the latter case, $f(a, b) \neq f(a', b)$ for all distinct elements $a, a' \in A$, and $b \in B$. Otherwise, if $f(a, b) = f(a', b)$, then f does not preserve the inequality relation, because there is $b' \in B$ such that $b' \neq b$ and $f(a, b) = f(a, b')$, and hence $f(a, b) = f(a', b')$, but $a \neq a'$ and $b \neq b'$. But this is impossible, because Lemma 2.13 shows that in this case f locally generates a constant operation. Therefore, we found two n-element sets A and B such that either $f(a, b) \neq f(a', b)$ for all $a, a' \in A$ and $b \in B$, or $f(a, b) \neq f(a, b')$ for all $a \in A$ and $b, b' \in B$. Without loss of generality we assume that the first case applies.

Since f cannot only depend on the first argument, there are elements u, v_1, and v_2 in D such that $v_1 \neq v_2$ and $f(u, v_1) \neq f(u, v_2)$. We can assume that v_2 is from B: For this, consider any element v' of B. If $f(u, v') \neq f(u, v_1)$, we choose v' instead of v_2 and are done. If $f(u, v') = f(u, v_1)$, then $f(u, v') \neq f(u, v_2)$, and we choose v' instead of v_2 and v_2 instead of v_1.

Let α_1 be a permutation of D that maps $\overline{s}[p] = \overline{s}[q]$ to u and the other entries in $\overline{s}$ to A. Let α_2 be a permutation of D that maps $\overline{t}[p]$ to v_1, $\overline{t}[q]$ to v_2, and the other entries in $\overline{t}$ to B. First, assume that we could also choose $u \in A$ and $v_1 \in B$ with $f(u, v_1) \neq f(u, v_2)$. Then it is easy to check that the equivalence

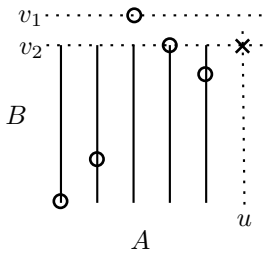


Figure 3.1: A binary operation f generating an injective binary operation from the proof of Theorem 3.4. Circles mark values for indices from equivalence classes of ρ contained in equivalence classes of ρ'. A cross marks the value $f(\alpha_1(\overline{s}[q]), \alpha_2(\overline{t}[q]))$.

relation σ of the tuple $f(\alpha_1(\overline{s}), \alpha_2(\overline{t}))$ has more equivalence classes than ρ and since f preserves R, we also have that $\sigma \in R$. Because σ is always coarser than $\rho \cap \rho'$, we easily obtain a contradiction to the maximality of $(c(\rho), c(\rho'))$.

So now assume that $f(a, v) = f(a, v')$ for all $a \in A$ and $v, v' \in B$ and fix $u \in D \setminus A, v_1 \in D, v_2 \in B$ with $f(u, v_1) \neq f(u, v_2)$. Now, the argument about σ is more refined. By Lemma 2.13, f preserves $\neq$ and hence $f(u, v) \neq f(a, b)$ for any $v \in D, a \in A, b \in B$. If $r = 0$, then for each equivalence class of ρ, there is an index $i \in [k]$ from this class such that $\overline{t}[i] \in B$. Therefore all these indices are in distinct equivalence classes of σ. Because index p was mapped to yet another equivalenc class, we see that σ has more equivalence classes than ρ. A contradiction to the maximality of $(c(\rho), c(\rho'))$. If $r \geq 1$, then σ has more equivalence classes than ρ', for the following reason (see Figure 3.1): Every equivalence class C of ρ' either consists of a union of equivalence classes from ρ, or contains an element from an equivalence class in ρ that is not contained in C. But also in the latter case, by the choice of p and q such that r is minimal, we can infer that C contains some equivalence class from ρ. Hence, in both cases, we can associate in that way one equivalence class from ρ to every class in ρ' and we can also pick an index contained in the contained equivalence class of ρ for each equivalence class of ρ'. By the choice of α_1, α_2 and because f preserves $\neq$, all these picked indices belong to distinct equivalence classes in σ. Moreover, $f(\alpha_1(\overline{s}[q]), \alpha_2(\overline{t}[q]))$ lies in yet another equivalence class of σ since no equivalence class of ρ' contains a class of elements equivalent to q in ρ. Thus, σ has more equivalence classes than ρ'. Since σ is not coarser than ρ, the existence of the relations ρ and σ then contradicts the choice of ρ and ρ' where $(c(\rho), c(\rho'))$ was lexicographically maximal. □

Now, we show hardness of a CSP for the relation S that was defined in Example 3.2.

Lemma 3.5. *If the relation S has a primitive positive definition in Γ, then* $\mathrm{CSP}(\Gamma)$ *is NP-hard.*

Proof. First observe that by identification of arguments x_1 and x_2, if S has a primitive positive definition in Γ, then the inequality relation has a primitive positive definition in Γ as well. We prove the NP-hardness by reduction from the NP-hard problem 3-COLORING [45]. Let $G = (V, E)$ be a graph that is an instance of 3-COLORING. We construct an instance of $\mathrm{CSP}(\Gamma)$ that has a polynomial size in $|V|$ and $|E|$ and is satisfiable if and only if G has a proper 3-coloring. Lemma 2.4 asserts we can use inequality constraints and the relation S to formulate this instance. The set of variables in this instance is $V \cup V' \cup \{c_1, c_2, c_3\}$, where V' is a copy of V, and c_1, c_2, c_3 are three new variables representing colors. We impose inequality constraints on each pair in c_1, c_2, c_3 and on each pair (u, v) for $uv \in E$. We impose the constraint S on (c_1, v', c_2) for each $v' \in V'$, and on (v', v, c_3) for each $v \in V$, where v' is the copy of v in V'. By construction, a solution to these constraints induces a proper 3-coloring of G. Conversely, a simple case analysis shows that any proper 3-coloring can be extended in a way that satisfies these constraints. □

As we already mentioned in the introduction, the constraint satisfaction problem for equality constraint languages is always contained in NP. By combining the results obtained in this chapter we can prove the following:

Theorem 3.6. *If Γ has no constant unary and no injective binary polymorphism, then* $\mathrm{CSP}(\Gamma)$ *is NP-complete.*

Proof. If the relation S can be expressed in Γ, we have by Lemma 3.5 and Lemma 2.4 that $\mathrm{CSP}(\Gamma)$ is NP-complete. If relation S cannot be expressed in Γ, Lemma 2.12 asserts that there is an at most binary polymorphism f of Γ violating S as S is a union of two orbits of triples. If f is essentially unary, then it generates a unary non-injective operation (because it violates S) and by Lemma 2.13 a constant unary polymorphism. Otherwise f is essentially binary and by Theorem 3.4 we are done. □

3.4 Tractable Constraint Languages

The case that Γ contains a constant unary polymorphism gives rise to trivially tractable constraint satisfaction problems: If an instance of such constraint satisfaction problem has a solution, then there is also a solution that maps all variables to a single value. In this case, an instance of $\mathrm{CSP}(\Gamma)$ is satisfiable if and only if it does not contain a constraint R, where R denotes the empty relation in Γ. Clearly, this can be decided efficiently. Such constraint languages are also called 0-valid in the literature, e. g. in [26].

To finish the classification of the complexity of equality constraint languages, we are left with the case that Γ has a binary injective polymorphism. We present an algorithm with polynomial running time that can adapted to work for representations of the constraint relations by sets of tuples and for representations by DNF formulae.

These two representations of the constraint relations in the input are different with respect to succinctness. It is easy to transform representations by sets of tuples into representations by DNF. For every relation R that is represented by a set of l k-tuples, we can find a formula ϕ of size at most $O(k^2 l)$ such that a tuple $\overline{a}$ satisfies ϕ if and only if $\overline{a}$ is in R. The formula can be found as follows: For each equivalence relation $\rho \in R$, we introduce one monomial to ϕ. The monomial contains $x_i = x_j$ if $(i, j) \in \rho$, and contains $x_i \neq x_j$ if $(i, j) \notin \rho$. Note that given a DNF formula $\phi(x_1, \ldots, x_k)$ that represents a k-ary relation R, the size of the representation of R by a set of tuples may be exponential in the size of ϕ. Hence, a polynomial time algorithm for instances where the constraints are represented in DNF implies a polynomial time algorithm for constraints represented as tuples, but not vice-versa.

We now review known algorithmic results for intersection-closed equality constraint languages. If the constraint relations in an instance of the CSP are explicitly given as Horn clauses, then the corresponding constraint satisfaction problem can be solved by a resolution-based algorithm that was developed by Bürkert and Nebel for temporal reasoning problems [22]. The worst-case running time of this algorithm is cubic in the size of the input. Our algorithm for representations by DNF formulae can be applied in this case as well, and has a significantly better running time. The algorithm of Nebel and Bürkert can also be applied, if the constraint language is finite, since we can in this case assume that the Horn representation of all the relations in the constraint language is known. However, their algorithm cannot be applied if the constraint language is infinite and the constraint relations are represented by sets of tuples or by formulae in DNF.

If an intersection-closed equality constraint language is finite or represented by sets of tuples, then the corresponding constraint satisfaction problem can be solved by an instantiation of the relational consistency algorithm as introduced in [31]. The worst-case running time of this algorithm is at most quadratic in the size of the input, and again the algorithm presented below has a significantly better running time.

We now present our new algorithm for intersection-closed equality constraints. The algorithm works both for representations by DNF formulae and representations by sets of tuples, only the implementation of some auxiliary procedures differs.

Algorithm 3.1. *Input: a set of variables X, a set of constraints $\mathcal{C}$*
// For each constraint s, $V(s)$ denotes a list of variables

```
// constrained by s.
// For each variable x, we construct a set C(x) that contains a pair (s, i)
// for all constraints s where the variable x appears at the i-th position.
for each x ∈ X do
  C(x) := ∅
for each s ∈ 𝒞 do
  for i := 1 to arity(s) do
    C(V(s)[i]) := C(V(s)[i]) ∪ {(s, i)}
for each x ∈ X do
  P(x) := |C(x)|
// Todo contains constraints that impose further contractions
Todo := 𝒞
// Contracted is a graph on the variables that contains edges for contracted
// pairs of variables
Contracted := (X, ∅)
while Todo ≠ ∅ do begin
  Let s be an arbitrary element from Todo
  Todo := Todo \ {s}
  // Compute new contractions forced by s
  c := ForcedContractions(s)
  for all {x, y} ∈ c do begin
    // Perform the contraction of x and y
    Add edge {x, y} to Contracted
    // Compute a list of constraints containing both x and y
    I := ∅
    if P(x) ≥ P(y) then begin
      for each (s', i) ∈ C(y) do
        if (s', ?) ∈ C(x) then
          I := I ∪ {s'}
    end
    else begin
      for each (s', i) ∈ C(x) do
        if (s', ?) ∈ C(y) then
          I := I ∪ {s'}
    end
    // Update all affected constraints
    for all s' ∈ I do begin
      // Compute a representation of s' ∧ x = y
      UpdateConstraint(s',x,y)
      if |s'| = 0 then
        Reject
      if s' changed then
        Todo := Todo ∪ {s'}
```

```
      end
      // Update the occurrences of the less frequent variable
      if P(x) ≥ P(y) then begin
        for each (s, i) ∈ C(y) do
          V(s)[i] := x
        C(x) := C(x) ∪ C(y)
        P(x) := P(x) + P(y)
      end
      else begin
        for each (s, i) ∈ C(x) do
          V(s)[i] := y
        C(y) := C(x) ∪ C(y)
        P(y) := P(x) + P(y)
      end
    end
  end
```
Assign a different value to each connected component in *Contracted*.
Assign to each variable the value of its component.

Algorithm 3.1 uses procedures `ForcedContractions` and `UpdateConstraint` that are not implemented in the pseudocode above. Their implementation depends on the representation of the constraints in the input, and will be presented later.

Recall that a k-tuple $\overline{s}$ *represents* an equivalence relation σ if $\sigma(x, y)$ if and only if $\overline{s}[x] = \overline{s}[y]$. We say that a k-tuple $\overline{s}$ is *finer* than a k-tuple $\overline{t}$ (and $\overline{t}$ is *coarser* than $\overline{s}$) if the equivalence relation represented by $\overline{s}$ is finer than the equivalence relation represented by $\overline{t}$. A k-tuple $\overline{u}$ is an *intersection* of two k-tuples $\overline{s}$ and $\overline{t}$ if $\overline{u}$ represents the intersection of the equivalence relations represented by $\overline{s}$ and $\overline{t}$. We say that a tuple is *consistent* with $x = y$ ($x \neq y$) if the value assigned to x is equal to (different from) the value assigned to y.

For representations of the constraints by formulae in DNF, we need the following definitions. A formula ϕ in DNF is a disjunction of *monomials*, and a *monomial* is a conjunction of *literals*. A *literal* is either a positive atom of the form $x = y$ or a negative atom of the form $x \neq y$, where x and y are variables from X. For example, the formula $(x = y \wedge x \neq z) \vee (x \neq y) \vee (y \neq z \wedge x = y)$ has three monomials.

Fix an enumeration $z_1, \ldots, z_n$ of the variables X. We say that a k-tuple $\overline{t}$ *satisfies* a monomial with k variables $z_{i_1}, \ldots, z_{i_k}$, where $1 \leq i_1 \leq \cdots \leq i_k \leq n$, if after the substitution of z_{i_j} by $\overline{t}[j]$ for all $j \in [k]$ all literals of the monomial are true. Observe that if two tuples satisfy a monomial, then their intersection also satisfies a monomial. Hence, there is a unique finest tuple among all tuples that satisfy the monomial. We say that a monomial M is *finer* than a monomial M' if the finest tuple satisfying M is finer than the finest tuple satisfying M'. We

say that a monomial M is *consistent* with $x = y$ ($x \neq y$) if there exists a tuple satisfying M that is consistent with $x = y$ ($x \neq y$).

First, we describe the implementation of procedures for the case that constraints are represented by sets of tuples. Procedure `ForcedContractions` selects the finest tuple $\overline{t}$ from its input constraint s. For each value v that appears in $\overline{t}$, the procedure adds to the returned list $\{V(s)[i_1], V(s)[i_j]\}$ for each $2 \leq j \leq l$, where $i_1, \ldots, i_l$ are the indices of entries that have value v in $\overline{t}$. The procedure `UpdateConstraint(`s`,`x`,`y`)` goes through the list of tuples in the representation of s and removes those tuples where x and y have different values.

If the constraints are represented by formulae in DNF, we need an additional data structure to achieve the desired running time. For each monomial M of a formula in DNF, we construct a graph $G(M)$ whose vertices are the variables constrained by the monomial. Note that for a k-ary constraint the number of such variables might be less than k, because some variables might occur several times in the list $V(s)$, or because a monomial uses less than k variables. The edges of the graph are the pairs $\{x, y\}$ such that the monomial contains the literal $x = y$. We arbitrarily select one vertex from each connected component of this graph, and call this vertex the *representant* of the component. Our data structure contains for each variable x constrained by the monomial a pointer $Rep(x)$ to the representant of the connected component containing x. Moreover, for each representant of a connected component, we maintain a hash table. For each term $x \neq y$ in the monomial, we insert $Rep(x)$ to the hash table of $Rep(y)$, and $Rep(y)$ to the hash table of $Rep(x)$.

The implementation of procedure `ForcedContractions(`s`)` selects the *finest* monomial M_0 in s, i. e., a monomial that is finer than all other monomials in the formula defining the constraint relation. Note that such finest monomial always exists, because there is the finest tuple satisfying the constraint, and this tuple satisfies at least one monomial, which is the finest monomial. Then `ForcedContractions` returns a list that contains all edges $\{x, Rep(x)\}$ for all vertices x of $G(M)$ satisfying $x \neq Rep(x)$.

The procedure `UpdateConstraint(`s`,`x`,`y`)` looks up $Rep(y)$ in the hash table of $Rep(x)$ for each monomial M. If $Rep(y)$ is found in the table, we remove the monomial M from s. Otherwise, suppose that the size of the hash table of $Rep(x)$ plus the number of variables having the representant $Rep(x)$ is larger than the corresponding number for $Rep(y)$. In this case, consider the set of all variables having the representant $Rep(y)$. We change the representant of these variables to $Rep(x)$ and rehash the hash table of $Rep(y)$ into the hash table of $Rep(x)$. In case that the size of the hash table of $Rep(x)$ plus the number of variables having the representant $Rep(x)$ is smaller or equal to the corresponding number for $Rep(y)$, we symmetrically change the representant of all variables having $Rep(x)$ as a representant to $Rep(y)$ and rehash the hash table of $Rep(x)$ to the hash table of $Rep(y)$.

Now, we show the correctness of Algorithm 3.1.

Lemma 3.7. *The procedure* `ForcedContractions(`s`)` *returns a list of pairs of variables* $\{x, y\}$ *such that* $x = y$ *holds in every assignment that satisfies the constraint* s *and that is also consistent with* $u = v$ *for every edge* $\{u, v\}$ *in* Contracted. *After the execution of the procedure* `UpdateConstraint(`s`,`x`,`y`)` *the constraint description contains exactly those tuples/monomials that are consistent with* $x = y$ *for each edge* $\{x, y\}$ *in* Contracted.

Proof. The lemma is easy to verify in the case that constraints are represented by sets of tuples. The procedure `UpdateConstraint` removes exactly tuples where $x \neq y$ and so its correctness is obvious. Also the correctness of the procedure `ForcedContractions` then immediately follows.

If constraints are represented by formulae in DNF, let M be a monomial in the formula representing s. By the construction of $G(M)$ variables having the same representant get the same value in any assignment satisfying M. Also $Rep(x)$ is in the hashtable of $Rep(y)$ if and only if values of $Rep(x)$ and $Rep(y)$ are different in any assignment satisfying M. Therefore, we find $Rep(x)$ in the hashtable of $Rep(y)$ in *UpdateConstraint* if and only if monomial M can never be satisfied and we have correctly removed it. So `UpdateConstraint` works as described in the statement of the lemma. Correctness of `ForcedContractions` then immediately follows. □

Lemma 3.8. *If* $\{x, y\}$ *is an edge in* Contracted, *then either the instance* $(X, \mathbb{N}, \mathcal{C})$ *has no solution, or* x *and* y *have the same value in every solution.*

Proof. Suppose the instance $(X, \mathbb{N}, \mathcal{C})$ has a solution. We prove the lemma by induction on the number of edges in *Contracted*. If *Contracted* has no edges, the lemma trivially holds. Now suppose there are $k+1$ edges in *Contracted*. There is some edge $\{x, y\}$ that was added last and by induction, we know that the variables in the first k edges must have the same value in every solution. The edge $\{x, y\}$ was added because it was returned by the procedure `ForcedContractions` for some constraint s'. Lemma 3.7 implies that then $x = y$ holds in every satisfying assignment to s' that is consistent with the equalities described by the first k edges. Hence, because s' must be satisfied, we conclude that x and y have the same value in every solution. □

Let Σ be the equivalence relation on X such that $\Sigma(x, y)$ if and only if x is connected to y by a path in *Contracted*. For $s \in \mathcal{C}$, let $\Sigma(s)$ denote the equivalence relation Σ restricted to variables constrained by s.

Lemma 3.9. *If Algorithm 3.1 rejects the instance* $(X, \mathbb{N}, \mathcal{C})$, *then it does not have a solution.*

Proof. By Lemma 3.7, every constraint s contains all tuples/monomials that are consistent with all equalities contained in $\Sigma(s)$. As Lemma 3.8 shows, if $\{x, y\}$ is an edge of *Contracted*, variables x, y get the same value provided that $(X, \mathbb{N}, \mathcal{C})$

has a solution. Hence, if there is no remaining tuple/monomial in s, the constraint cannot be satisfied and $(X, \mathbb{N}, C)$ has no solution. □

We finish the correctness proof with the following lemma.

Lemma 3.10. *The assignment created by the last two steps of Algorithm 3.1 is a solution to* $(X, \mathbb{N}, \mathcal{C})$.

Proof. We first prove the statement for the case where the constraints in the input are represented by sets of tuples. Then every constraint s can be seen as a set of equivalence relations as discussed in Section 3.1. We show that for each constraint $s \in \mathcal{C}$ the equivalence relation $\Sigma(s)$ is contained in s. First, suppose that there is no $\sigma \in s$ coarser than $\Sigma(s)$. Consider a moment when a pair $\{x, y\}$ was added to *Contracted* and hence there ceased to be a coarser equivalence relation in s than the equivalence relation defined by *Contracted*. Clearly, s had to constrain a variable from both the connected component of x and the connected component of y. But then s was in I, all the tuples were removed from s, and the instance was rejected. A contradiction. So s has to contain an equivalence relation σ coarser than $\Sigma(s)$. If s does not contain $\Sigma(s)$, consider the last moment when some equivalence relation was removed from s. At this moment, s was added to *Todo* and s already did not contain $\Sigma(s)$. Hence, when s was considered later when processing *Todo*, we had to add to *Contracted* some pairs that were not in $\Sigma(s)$ (the pairs present in some equivalence relation coarser than $\Sigma(s)$ that was the finest in s) — in contradiction to the definition of $\Sigma(s)$.

When constraints are represented by formulae in DNF, the proof is similar. We argue that for each constraint s, there is a monomial that can be satisfied when all the equalities from $\Sigma(s)$ hold. Then we conclude (in the same way as in the above paragraph) that among all such monomials there must be one that is satisfied even when all the inequalities from $\Sigma(s)$ hold. □

Now, we discuss the running time of Algorithm 3.1. We show that for both representations of the input, the algorithm has the running time $O(m \log m)$, where m is the size of the input. If the constraint relations are represented by sets of tuples, m is $\sum_{s \in \mathcal{C}} ar(s) \cdot |s|$. If constraints $s \in \mathcal{C}$ are represented by formulae $\phi(s)$ in DNF, then m is $\sum_{s \in \mathcal{C}} |\phi(s)|$, where $|\phi|$ is the length of the formula ϕ.

The initialization phase of the algorithm can clearly be implemented such that the initialization takes $O(m)$ time. To achieve the desired running time of procedure `ForcedContractions`, we have to describe how to efficiently find the finest tuple/monomial in the representation of a constraint relation. If the constraints are represented by sets of tuples, the idea is to sort the tuples of a constraint relation by the number of different values. Similarly, if the constraints are represented by formulae in DNF, we sort the monomials by the number of connected components in the graphs $G(M)$ defined for each monomial.

Observation 3.11. *Let R be an intersection-closed k-ary relation that is represented as a set of tuples. If we sort tuples in the representation of R inversely according to the number of different values (i. e., $\overline{s} < \overline{t}$ if $|\{\overline{s}[i] : 1 \leq i \leq k\}| > |\{\overline{t}[i] : 1 \leq i \leq k\}|$), then if $\overline{s}$ is finer than $\overline{t}$, it also holds that $\overline{s} < \overline{t}$.*

Proof. If $\overline{s}$ is finer than $\overline{t}$, then $\overline{s}$ must have more equivalence classes implying $\overline{s} < \overline{t}$. □

Observation 3.12. *Let R be an intersection-closed k-ary relation that is represented by a formula in DNF. If we sort the monomials inversely according to the number of connected components in the graph $G(M)$, then if M is finer than M' and M' is not finer than M, it also holds that $M < M'$.*

Proof. If M is finer than M' and M' is not finer than M, the finest tuple satisfying M must have more values than the finest tuple satisfying M'. Hence graph of $G(M)$ must have more connected components than graph $G(M')$, since all the vertices in the same connected component must get the same value. □

Observation 3.11 asserts that the first tuple in the sorted list is the finest one. Similarly, Observation 3.12 asserts that the first monomial in the sorted list is the finest one (as there is unique finest monomial). In both cases, we can use bucket sort such that the sorting is done in $O(m)$ time.

A constraint is added to *Todo* only if it has a new finest tuple/monomial (either in the beginning of the algorithm or after removal of an older tuple/monomial). Hence, the total time spent by processing the *Todo* list is in $O(m)$. The total time spent by procedure `ForcedContractions` can be also estimated by $O(m)$, because the finest tuple/monomial is the first in the list and because the number of contractions is linear in the size of a tuple/monomial.

What remains is to estimate the time spent by contracting pairs of variables. The computation of constraints containing both x and y and of $C(x) \cup C(y)$ can be implemented as follows. We represent each set $C(x)$ as a hash table in which a constraint containing x is the hashed key and the index of entry corresponding to x is a value associated with this key. For each hash table $C(x)$, we keep a potential $P(x)$. In the beginning, we assign each hash table a potential $P(x)$ equal to $|C(x)|$. Hence, the sum of all potentials equals m. If two tables are joined, we set the potential of $C(x) \cup C(y)$ to $P(x) + P(y)$. When we compute a list of constraints containing both x and y and also when we compute unions of hash tables, we always rehash the table with the smaller potential into the table with the larger potential. By rehashing, we obtain a hash table with $C(x) \cup C(y)$. The overall time spent for the computations of unions is clearly linear in the number of elements we had to rehash. The overall time spent by looking up constraints present in both hashtables is upperbounded by the time needed for the computation of unions and therefore we can neglect it. We can estimate the number of times an element of a table is rehashed as follows. If an element is

rehashed, the potential of the resulting table is at least twice the potential of the rehashed one, because we rehashed the table with the smaller potential. As the potential of a table can never exceed m, we immediately get that a single element can be rehashed at most $\log m$ times. Hence, the overall time spent in the computations of unions and intersections is in $O(m \log m)$. Similarly, we can estimate that a single element of a list $V(s)$ is relabeled at most $\log m$ times, and hence the time spent by relabeling these lists is also in $O(m \log m)$.

Finally, we need to estimate the time spent in procedure `UpdateConstraint` (the time spent for the remaining operations that perform contractions is clearly in $O(m)$). The test whether a tuple/monomial is consistent with $x = y$ can be performed in $O(1)$. If the constraints are represented by formulae in DNF, we have to further spent some time for updating representants and rehashing the hash tables inside `UpdateConstraints`. Here, we can use a similar potential argument as for the hash tables $C(x)$, and conclude that the total time spent by these update operations is in $O(m \log m)$.

We summarize the major results of this section in the following theorem.

Theorem 3.13. *Let Γ be an intersection-closed equality constraint language and $(X, \mathbb{N}, \mathcal{C})$ be an instance of $\mathrm{CSP}(\Gamma)$. If the constraint relations in $\mathcal{C}$ are represented by sets of tuples or by formulae in DNF, then Algorithm 3.1 computes a solution of $(X, \mathbb{N}, S)$ (or decides that there is no solution) in time $O(m \log m)$, where m denotes the size of the input.*

If the constraint language is finite, the size of an instance I can be measured by $i := \sum_{s \in \mathcal{C}} ar(s)$. The result for representations by sets of tuples implies that the constraint satisfaction problem can be solved in time $O(i \log i)$.

3.5 The Meta Problem for Tractability

Let Γ be a finite equality constraint language. In this section, we want to study the computational complexity of determining whether $\mathrm{CSP}(\Gamma)$ can be solved in polynomial time. The complexity of this task depends on the choice of the representation in which Γ is specified. As in the previous section, we focus on representations by sets of tuples, and representations by DNF formulae.

Let us first consider representations of the relations in Γ by sets of tuples. We first check whether all relations R in Γ admit the constant assignment. To determine whether R is intersection closed, we have to check whether for all pairs of tuples $\overline{r}, \overline{s}$ in the representation of a relation R in Γ the intersection of the tuples (the coarsest tuple that is finer than both tuples $\overline{r}$ and $\overline{s}$) is also in the representation of R. Clearly, these computations can be done in polynomial time.

We have thus shown the following.

Proposition 3.14. *If Γ is an equality constraint language that is represented by a set of tuples, then the meta problem can be solved in polynomial time.*

Now, suppose the relations in Γ are represented in DNF. Again, it can be checked easily whether CSP(Γ) is 0-valid. We can also set up an algorithm that non-deterministically verifies in polynomial time that for any two distinct equivalence relations in R their intersection is in R as well. Hence, the meta-problem for these two representations is in coNP. We now show that this problem is also coNP-hard.

Proposition 3.15. *It is coNP-complete to decide whether an equality constraint language Γ whose relations are represented by an $=$-formula in DNF is tractable.*

Proof. We have already seen that the problem is in coNP. To show hardness, let ϕ be a propositional formula in DNF. We create an $=$-formula Ψ as follows. For each propositional variable x in ϕ, we introduce a new pair of variables u_x and v_x. For each monomial $x_1 \wedge \cdots \wedge x_l \wedge \neg y_1 \wedge \cdots \wedge \neg y_k$ in ϕ, we introduce a monomial $u_{x_1} = v_{x_1} \wedge \cdots \wedge u_{x_l} = v_{x_l} \wedge u_{y_1} \neq v_{y_1} \wedge \cdots \wedge u_{y_k} \neq v_{y_k}$. Let ψ be the disjunction of all these monomials, and let Ψ be $\psi \vee u = v \vee u' = v'$, where u, u', v, and v' are new constraint variables that do not appear in ψ. We claim that Ψ is equivalent to a Horn $=$-formula if and only if ϕ is a boolean tautology. Clearly, if ϕ is a tautology, then ψ is a tautology as well, and hence Ψ is equivalent to `true` and so it is equivalent to a Horn formula.

Conversely, if Ψ is equivalent to a Horn formula, then it is intersection-closed by Lemma 3.1. The tuples that satisfy Ψ consist of the tuples where $u = v$ and the remaining variables are set arbitrarily, the tuples where $u' = v'$ and the remaining variables are set arbitrarily, and the tuples where ψ holds and u, u', v, and v' are set arbitrarily. If we compute intersection-closure of all tuples satisfying $u = v$ with all tuples satisfying $u' = v'$, we obtain (among other tuples) tuples having all possible configurations of equalities on the variables of Ψ and satisfying $u \neq v$ and $u' \neq v'$. It follows that ϕ is a tautology. The proposition then follows from the well-known fact that the problem whether a given boolean formula in DNF is a tautology is coNP-hard. □

Chapter 4

Temporal Constraint Languages

In this chapter, we present our results for another class of constraint languages over a countably infinite domain — *temporal constraint languages*. These are the languages that are definable over the dense linear order (for formal definition see Definition 4.1). We manage to fully classify the complexity of constraint satisfaction problems for these languages. In particular, we discover two new tractable languages and design algorithms for them. For example in Section 4.4, we present an algorithm solving constraint satisfaction problem for several languages strictly containing *Ord-Horn* language introduced by Bürckert and Nebel [22]. Running time of our algorithm meets the running time of the fastest algorithm designed for this class by Koubarakis [69]. We also show in Section 4.3 that unlike *Ord-Horn* languages, some of our languages cannot be solved by Datalog. Part of this chapter is based on the paper [17] of the author.

First, we define a temporal constraint language:

Definition 4.1. *A* temporal relation *is a relation that is first-order definable in an unbounded countable dense linear order. All such linear orders are isomorphic [55, 76], but for convenience, we always use* $(\mathbb{Q}, <)$*, i. e., the dense linear order on the rational numbers. An example of a temporal relation is the ternary Betweenness relation* $\{(x, y, z) \in \mathbb{Q}^3 \mid (x<y \wedge y<z) \vee (z<y \wedge y<x)\}$*. It is well-known that every temporal relation also has a* quantifier-free *definition [55, 76], i. e., we can define every temporal relation with a formula that is a Boolean combination of literals of the form* $x < y$ *or* $x = y$ *(as shown above in the case of Betweenness relation).*

A temporal constraint language *is a constraint language whose every constraint relation is temporal. As an example, consider the language* $\Gamma_0 := (\mathbb{Q}, \{\neq, \leq, <, =\})$*, with the obvious interpretation over* $(\mathbb{Q}, <)$*.*

Temporal constraint languages play an important role in artificial intelligence. Almost any area in AI — for instance common-sense reasoning, natural language processing, scheduling, planning — involves some sort of temporal reasoning. In 1993, Golumbic and Shamir [47] listed applications of temporal reasoning

problems in archeology, behavioral psychology, operations research, and circuit design. Since then, the area has been growing constantly, and temporal reasoning became one of the benchmark applications of constraint processing in general [30]. Contributions to the field have various background, e. g. database theory [79], constraint satisfaction complexity [74], the theory of relation algebras [75, 36], combinatorics [47], or the already mentioned artificial intelligence [41].

One of the most fundamental and well-known temporal constraint languages is the so-called *point algebra*. This language contains relation symbols for $=, <, \leq$, and $\neq$, interpreted over an infinite linear order. Vilain, Kautz and van Beek showed that consistency of a given set of constraints over this language can be decided in polynomial time by local consistency techniques [66]. Later, van Beek described an algorithm that runs in $O(n^2)$, where n is the number of variables [7].

A considerably larger tractable temporal constraint language is the already mentioned *Ord-Horn*. It strictly contains the point algebra and Bürkert and Nebel used resolution to show that consistency of a set of Ord-Horn constraints can be decided in $O(s^3)$, where s is the size of the input. They also showed that establishing path-consistency can be used to decide whether a given set of Ord-Horn constraints has a solution.

Ord-Horn is motivated by temporal reasoning tasks for constraints on *time intervals*. The study of constraints on intervals (which can be used to model information about events in time) was initiated by Allen [1], who introduced an algebra of binary constraint relations on intervals. The complexity to decide the consistency of a given set of constraints from Allen's algebra is NP-complete in general [1]. However, several fragments of Allen's interval algebra are tractable. All tractable fragments have been classified recently [35, 57]. It is well-known that every constraint on *intervals* can be translated into a constraint on *time points*. Hence, algorithmic results for temporal reasoning with time points can be used for reasoning with time intervals as well. Bürkert and Nebel used this translation to identify one of the tractable fragments of Allen's interval algebra, namely the set of all interval constraints that translate to Ord-Horn constraints on points.

There are temporal constraint languages for time points where one cannot expect a polynomial time algorithm. A well-known temporal constraint language with an NP-complete consistency problem consists of a single relation symbol for the *Betweenness* relation from Definition 4.1; another example of a constraint language with an NP-complete constraint satisfaction problem is the language containing the *cyclic ordering relation*, which is the ternary relation $\{(x, y, z) \mid x{<}y{<}z \vee y{<}z{<}x \vee z{<}x{<}y\}$. The constraint satisfaction problems for these two languages are listed as NP-complete in the book of Garey and Johnson [45]. We want to remark that the complexity of temporal constraint satisfaction problems for a *fixed and finite* number of time points was completed recently [27]; however, the restriction to a finite number of time points changes the nature of the problem considerably.

To conclude introduction to temporal constraint languages we would like to define some notation and make a few observations about temporal constraints. It is straightforward to check (due to properties of $\mathrm{Aut}(\mathbb{Q}, <)$ that whether or not an n-tuple $\overline{t}$ is a solution to an instance only depends on the weak linear order $tp(\overline{t})$ defined on $\{1, \ldots, n\}$ by $(i, j) \in tp(\overline{t})$ if and only if $\overline{t}[i] \leq \overline{t}[j]$. This immediately leads to an observation that a constraint satisfaction problem for every temporal constraint language is in NP. If the weak linear order $tp(\overline{s})$ is identical to $tp(\overline{t})$ for some n-tuple $\overline{s}$, we say that $\overline{s}$ *satisfies* $tp(\overline{t})$. This observation leads to a natural way of representing temporal relations. If R is a k-ary temporal relation, R can be represented by a set $\mathcal{R}$ of weak linear orders on $[k]$ as follows: For every k-tuple $\overline{t} \in R$, the weak linear order $tp(\overline{t})$ is contained in $\mathcal{R}$. Conversely, for every weak linear order $o \in \mathcal{R}$ there is a tuple $\overline{t} \in R$ such that $o = tp(\overline{t})$. As an example, the relation *Betweenness* defined above can be characterized as the set of all tuples satisfying either $tp((0, 1, 2))$ or $tp((2, 1, 0))$. As we can represent each weak linear order in $\mathcal{R}$ by a tuple over $\{1, \ldots, k\}$, we can also view $\mathcal{R}$ as a set of tuples over $\{1, \ldots, k\}$. We use this view especially in the algorithms, where we work with constraint relations as with finite sets of tuples of integers.

We say a k-tuple $\overline{t}$ is *injective*, if for all $i, j \in [k], i \neq j$ it holds that $\overline{t}[i] \neq \overline{t}[j]$. We say a relation R is *injective*, if for all $\overline{t} \in R$ the tuple $\overline{t}$ is injective.

4.1 Closure Properties of Temporal Constraints

In this section, we introduce several closure properties of temporal constraints used later. For the sake of brevity, we say that a set of operations F *generates* an operation g if F together with all automorphisms of $(\mathbb{Q}, <)$ locally generates g.

A k-ary operation f on $\mathbb{Q}$ defines a weak linear order on $\mathbb{Q}^k$. We can simply define that $\overline{x}$ is less or equal to $\overline{y}$ in this weak linear order if $f(\overline{x}) \leq f(\overline{y})$. The following observation easily follows from the properties of $\mathrm{Aut}(\mathbb{Q}, <)$:

Observation 4.1. *Let f and g be two k-ary operations that define the same weak linear order on $\mathbb{Q}^k$. Then f generates g and g generates f.*

We start by defining two unary operations neg and cyc on $\mathbb{Q}$. Operation neg is defined as $\mathrm{neg}(x) := -x$ and operation cyc is defined as $-1/(x-1)$ for $x \leq 0$ and $-1/(x+1)$ for $x > 0$. We continue with binary operations. Let lex be a binary operation on $\mathbb{Q}$ such that $\mathrm{lex}(a, b) < \mathrm{lex}(a', b')$ if either $a < a'$, or $a = a'$ and $b < b'$. Note that every operation lex satisfying these conditions is by definition injective. By Observation 4.1, it is easy to see that all such operations generate the same clone.

Let ll be a binary operation on $\mathbb{Q}$ such that $\mathrm{ll}(a, b) < \mathrm{ll}(a', b')$ if one of the following cases applies

- $a \leq 0$ and $a < a'$

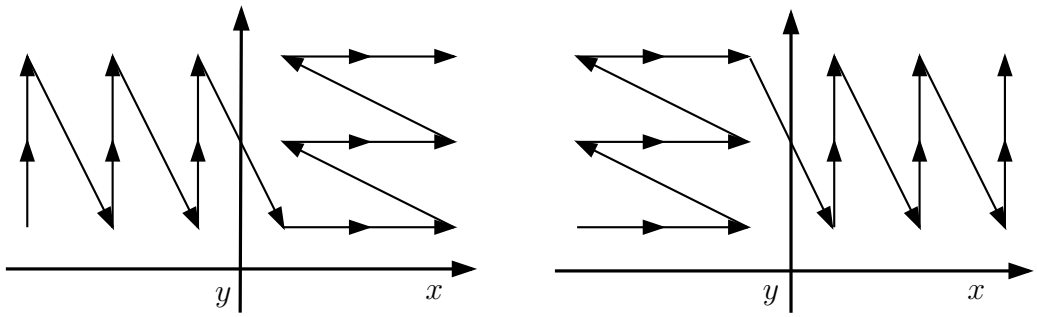


Figure 4.1: A visualization of ll (left) and dual-ll (right) operations

- $a \leq 0$ and $a = a'$ and $b < b'$
- $a > 0$ and $b < b'$
- $a > 0$ and $b = b'$ and $a < a'$

For an illustration, see Figure 4.1. In diagrams like this one, we draw a directed edge from (a, b) to (a', b') if $f(a, b) < f(a', b')$. Again, all operations satisfying these conditions are by definition injective and generate the same clone. It is also easy to see that ll generates lex. Similarly to the ll operation, we can define a dual-ll operation, as depicted in Figure 4.1.

Let pp be a binary operation on $\mathbb{Q}$ such that $\mathrm{pp}(a, b) \leq \mathrm{pp}(a', b')$ if one of the following cases applies:

- $a \leq 0$ and $a \leq a'$
- $a > 0$ and $b \leq b'$

For an illustration, see Figure 4.2. Clearly, pp is not injective. Similarly to the pp operation, we can define a dual-pp operation. Unoriented lines in rows and columns of the picture denote the pairs getting the same value.

Definition 4.2. *We say that a relation is* ll-closed*,* dual-ll-closed*,* pp-closed*, or* dual-pp-closed *if it is preserved by* ll*,* dual-ll*,* pp*, or* dual-pp*, respectively.*

To exercise properties of these operations, we show that neither of these operations generates one of the others. Recall that because of Proposition 2.11, we only need to find for each (ordered) pair of operations a relation that is preserved by the first operation but not by the second one.

Definition 4.3. *Let* $R^{\min}$ *be the ternary relation* $\{(x, y, z) \mid x{>}y \vee x{>}z\}$*, and* $R^{\max}$ *be* $\{(x, y, z) \mid x{<}y \vee x{<}z\}$*.*

These relations show that ll and dual-ll do not generate each other:

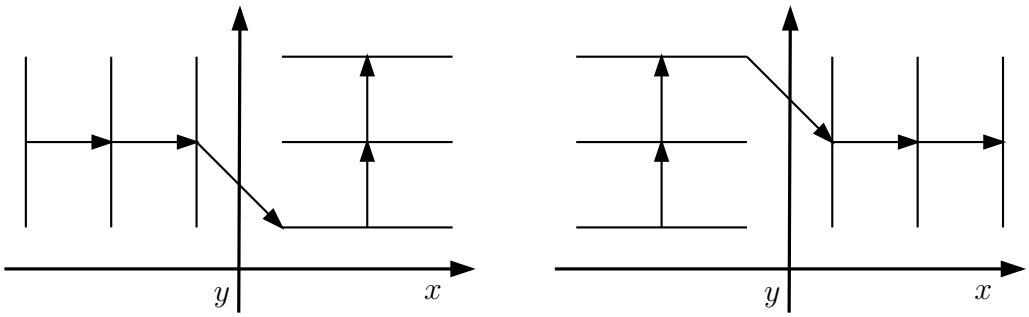


Figure 4.2: A visualization of pp (left) and dual-pp (right) operations

Lemma 4.2. *The relation* $R^{\min}$ *is preserved by* ll *and not preserved by* dual-ll. *Similarly, the relation* $R^{\max}$ *is preserved by* dual-ll *and not preserved by* ll.

Proof. We prove the lemma just for the $R^{\min}$ relation. The proof of the statement for $R^{\max}$ relation is analogous. We claim that the relation $R^{\min}$ is preserved by the ll operation: Let (x_1, x_2, x_3) and (y_1, y_2, y_3) be triples from the relation $R^{\min}$. Without loss of generality, $x_1 > x_2$ (note that the relation is symmetric in the second and the third argument). If in this case $y_1 \geq y_2$, then, because ll preserves $\leq$, we have that $\mathrm{ll}(x_1, y_1) \geq \mathrm{ll}(x_2, y_2)$, and because ll is injective, we have that $\mathrm{ll}(x_1, y_1) > \mathrm{ll}(x_2, y_2)$. Therefore $(\mathrm{ll}(x_1, y_1), \mathrm{ll}(x_2, y_2), \mathrm{ll}(x_3, y_3))$ is in $R^{\min}$, and we are done. So let us assume that $y_1 < y_2$, and therefore $y_1 > y_3$. By the same argument as above, we can show that $(\mathrm{ll}(x_1, y_1), \mathrm{ll}(x_2, y_2), \mathrm{ll}(x_3, y_3)) \in R^{\min}$ unless $x_1 < x_3$. So let us assume that $x_1 < x_3$. Now, in case $x_2 > 0$, the operation ll preserves $R^{\min}$, since in this case ll acts like a lexicographic order on the two triples. Otherwise, $x_2 \leq 0$. It is easy to check that then $\mathrm{ll}(x_2, y_2) < \mathrm{ll}(x_1, y_1)$ since $x_1 > x_2$.

However, $R^{\min}$ is not preserved by the dual-ll operation: Consider tuples $\overline{t_1} := (-1, 1, -2)$ and $\overline{t_2} := (-1, -2, 1)$ that are both in $R^{\min}$. If we apply the dual-lloperation to these two tuples, we obtain dual-ll$(-1, -1) <$ dual-ll$(-2, 1) <$ dual-ll$(1, -2)$, and hence the tuple dual-ll$(\overline{t_1}, \overline{t_2})$ is not in the relation $R^{\min}$. □

Similarly, we can define distinguishing relations for pp and dual-pp:

Definition 4.4. *Let* $S^{\min}$ *be the ternary relation* $\{(x, y, z) \mid x = y < z \vee x = z < y\}$, *and* $S^{\max}$ *be* $\{(x, y, z) \mid x = y > z \vee x = z > y\}$.

Lemma 4.3. *The relation* $S^{\min}$ *is preserved by* pp *and not preserved by* dual-pp. *Similarly, the relation* $S^{\max}$ *is preserved by* dual-pp *and not preserved by* pp.

Proof. Again, we prove the lemma only for $S^{\min}$ as the proof for $S^{\max}$ is analogous. Let (x_1, x_2, x_3) and (y_1, y_2, y_3) be triples from $S^{\min}$. Since pp preserves $\leq$ and $<$, if $x_1 = x_2 < x_3$ and $y_1 = y_2 < y_3$ (or similarly if $x_1 = x_3 < x_2$ and $y_1 = y_3 < y_2$),

the resulting triple $(z_1, z_2, z_3) := (\mathrm{pp}(x_1, y_1), \mathrm{pp}(x_2, y_2), \mathrm{pp}(x_3, y_3))$ satisfies $z_1 = z_2 < z_3$ ($z_1 = z_3 < z_2$, respectively) and hence is in $S^{\min}$. So suppose without loss of generality ($S^{\min}$ is symmetric in the second and the third argument) that $x_1 = x_2 < x_3$ and $y_1 = y_3 < y_2$. If $x_1 \leq 0$, then by the definition of pp it holds that $z_1 = z_2$ and $z_2 < z_3$. Hence $S^{\min}$ holds on the resulting tuple. Also if $x_1 > 0$, we have $z_1 = z_3$ by the definition of pp and $z_3 < z_2$ and so $S^{\min}$ again holds on the resulting tuple. We conclude that $S^{\min}$ is preserved by pp.

On the other hand, if we apply dual-pp to triples $(-1, -1, 1)$ and $(-1, 1, -1)$, we obtain a triple dual-pp$(-1, -1) <$ dual-pp$(-1, 1) <$ dual-pp$(1, -1)$. Hence, we obtained an injective triple, which clearly is not in $S^{\min}$. □

Actually, relations $S^{\min}$ and $S^{\max}$ are not preserved by ll or dual-ll because these operations are injective. Therefore we conclude:

Corollary 4.4. *The operations* pp *and* dual-pp *do not generate the operation* ll *or the operation* dual-ll*.*

Finally, to show that ll and dual-ll do not generate pp or dual-pp, we a introduce relation L:

Definition 4.5. *Let L be a 4-ary relation $\{(u, v, x, y) \mid u \neq v \vee x \neq y\}$.*

Lemma 4.5. *The relation L is preserved by* ll *and by* dual-ll *but not by* pp *or by* dual-pp*.*

Proof. The preservation of L by ll and by dual-ll directly follows from the fact that both operations are injective. The relation L is not preserved by pp, because for quadruples $(-1, -1, 1, 2)$ and $(0, 1, 2, 2)$ from L, the resulting tuple $(\mathrm{pp}(-1, 0), \mathrm{pp}(-1, 1), \mathrm{pp}(1, 2), \mathrm{pp}(2, 2))$ satisfies that $\mathrm{pp}(-1, 0) = \mathrm{pp}(-1, 1)$ and $\mathrm{pp}(1, 2) = \mathrm{pp}(2, 2)$. Similarly, it is easy to check that by applying dual-pp to quadruples $(-2, -1, 1, 1)$ and $(0, 0, 1, 2)$ from L we obtain a quadruple not from L. □

Now, we turn our attention to closure properties of a different kind that are useful for designing algorithms.

Definition 4.6. *The i-th entry in a k-tuple $\overline{t}$ is called* minimal *if $\overline{t}[i] \leq \overline{t}[j]$ for every $j \in [k]$. It is called* strictly minimal *if $\overline{t}[i] < \overline{t}[j]$ for every $j \in [k] \setminus \{i\}$.*

The i-th entry in a k-tuple $\overline{t}$ is called maximal *if $\overline{t}[i] \geq \overline{t}[j]$ for every $j \in [k]$. It is called* strictly maximal *if $\overline{t}[i] > \overline{t}[j]$ for every $j \in [k] \setminus \{i\}$.*

Definition 4.7. *Let R be a k-ary relation. A set of entries $S \subseteq [k]$ is called a* min-set for the i-th entry in R *if there exists a tuple $\overline{t} \in R$ such that the i-th entry is minimal in $\overline{t}$, and it holds that $\overline{t}[i] = \overline{t}[j]$ for every $j \in S$ and $\overline{t}[i] < \overline{t}[j]$ for every $j \in [k] \setminus S$. We say that $\overline{t}$ is a* witness *for this min-set. We simply say*

that a set of entries $S \subseteq [k]$ is a min-set in R if it is a min-set for some i-th entry.

Analogously, we define that a set of entries $S \subseteq [k]$ is a max-set for the i-th entry in R *if there exists a tuple $\overline{t} \in R$ such that the i-th entry is maximal in $\overline{t}$, and it holds that $\overline{t}[i] = \overline{t}[j]$ for every $j \in S$ and $\overline{t}[i] > \overline{t}[j]$ for every $j \in [k] \setminus S$.*

We also need a notion of a projection of a relation:

Definition 4.8. *Let R be a k-ary relation and $X \subset [k]$. A* projection of R *to $[k] \setminus X$ is a relation $p_{[k]\setminus X}(R) := \{(\overline{t}[i])_{i\in[k]\setminus X} \mid \overline{t} \in R\}$. An* ordered projection of R *to $[k] \setminus X$ is a relation $p^{<}_{[k]\setminus X}(R) := \{(\overline{t}[i])_{i\in[k]\setminus X} \mid \overline{t} \in R, \overline{t}[i] > \overline{t}[j]$ for all $i \in [k] \setminus X, j \in X\}$.*

Definition 4.9. *Let f be a binary operation preserving $<$. We say that f* provides a min-intersection closure *if $f(0,0) < f(0,x)$ and $f(0,0) < f(x,0)$ for all integers $x > 0$. We say that f* provides a min-union closure *if $f(0,0) = f(0,x) = f(x,0)$ for all integers $x > 0$. We say that f* provides a min-xor closure *if $f(0,0) > f(0,x) = f(x,0)$ for all integers $x > 0$.*

In a similar way, we can define a max-intersection, a max-union, and a max-xor closure provided by an operation. However, we mostly speak about min-properties in this chapter as the arguments can be straightforwardly dualized (e. g. by taking dual-ll instead of ll) to obtain similar results for max- properties.

Example 4.1. *For example operation* lex *provides a min-intersection closure and operation* min *provides a min-union closure. Operation*

$$f(x,y) := \begin{cases} \alpha(\min(x,y)) & \text{for } (x,y) \neq (0,0) \\ 0 & \text{for } x = y = 0 \end{cases}$$

where α is an automorphism of $(\mathbb{Q}, <)$ such that $\alpha(x) < 0$ for $x \leq 0$ and $\alpha(x) > 0$ for $x > 0$, provides a min-xor closure.

Definition 4.10. Closure properties of min-sets: *Let R be a temporal relation. We say that R has* intersection-closed/union-closed/xor-closed *min-sets if for every two min-sets A, B of R, it holds that:*

- *$A \cap B$ is a min-set (provided the intersection $A \cap B$ is nonempty) of R,*
- *$A \cup B$ is a min-set of R,*
- *$A \triangle B$ (the symmetric difference of A and B) is a min-set of R.*

Now, we relate the above properties of min-sets and operations providing some min- properties:

Lemma 4.6. *Let R be a temporal relation closed under an operation f that provides a min-intersection closure. Then R has intersection-closed min-sets.*

Proof. Let A and B be two min-sets and $\overline{t_A}$ and $\overline{t_B}$ two tuples witnessing these min-sets. Then we can take automorphisms α_A and α_B of $(\mathbb{Q}, <)$ such that α_A maps the minimal value of $\overline{t_A}$ to 0 and all other values of $\overline{t_A}$ to integers greater than 0. Similarly, α_B maps the minimal value of $\overline{t_B}$ to 0 and all other values of $\overline{t_B}$ to integers greater than zero. Consider a tuple $\overline{t_C} := f(\alpha_A(\overline{t_A}), \alpha_B(\overline{t_B}))$ witnessing some min-set C. Because $\alpha_A(\overline{t_A})$ is 0 for all entries from A, $\alpha_B(\overline{t_B})$ is 0 for all entries of B, $f(0,0) < f(0,x)$ and $f(0,0) < f(x,0)$ for all integers $x > 0$, it follows that in $\overline{t_C}$ all entries of $A \cap B$ have smaller value than all entries of $A \triangle B$. Because f preserves $<$, all entries of $A \cap B$ also have a smaller value than all entries not in $A \cup B$. We conclude that $C = A \cap B$. □

In the same straightforward, way we can prove the following two lemmas:

Lemma 4.7. *Let R be a temporal relation closed under an operation f that provides a min-union closure. Then R has union-closed min-sets.*

Proof. Let A and B be two min-sets and $\overline{t_A}$ and $\overline{t_B}$ two tuples witnessing these min-sets. Then we can take automorphisms α_A and α_B of $(\mathbb{Q}, <)$ such that α_A maps the minimal value of $\overline{t_A}$ to 0 and all other values of $\overline{t_A}$ to integers greater than 0. Similarly, α_B maps the minimal value of $\overline{t_B}$ to 0 and all other values of $\overline{t_B}$ to integers greater than zero. Consider a tuple $\overline{t_C} := f(\alpha_A(\overline{t_A}), \alpha_B(\overline{t_B}))$ witnessing some min-set C. Because $\alpha_A(\overline{t_A})$ is 0 for all entries from A, $\alpha_B(\overline{t_B})$ is 0 for all entries of B, $f(0,0) = f(0,x)$ and $f(0,0) = f(x,0)$ for all integers $x > 0$, it follows that in $\overline{t_C}$, all entries of $A \cup B$ have the same value. Because f preserves $<$, all entries of $A \cup B$ have also a smaller value than all entries not in $A \cup B$ (as they must have values greater than $f(0,0)$). We conclude that $C = A \cup B$. □

Lemma 4.8. *Let R be a temporal relation closed under an operation f that provides a min-xor closure. Then R has xor-closed min-sets.*

Proof. Let A and B be two min-sets and $\overline{t_A}$ and $\overline{t_B}$ two tuples witnessing these min-sets. Then we can take automorphisms α_A and α_B of $(\mathbb{Q}, <)$ such that α_A maps the minimal value of $\overline{t_A}$ to 0 and all other values of $\overline{t_A}$ to integers greater than 0. Similarly, α_B maps the minimal value of $\overline{t_B}$ to 0 and all other values of $\overline{t_B}$ to integers greater than zero. Consider a tuple $\overline{t_C} := f(\alpha_A(\overline{t_A}), \alpha_B(\overline{t_B}))$ witnessing some min-set C. Because $\alpha_A(\overline{t_A})$ is 0 for all entries from A, $\alpha_B(\overline{t_B})$ is 0 for all entries of B, $f(0,0) > f(0,x) = f(x,0)$ for all integers $x > 0$, it follows that in $\overline{t_C}$ all entries of $A \cap B$ have a larger value than all entries of $A \triangle B$, which have all the same value. Because f preserves $<$, all entries of $A \cap B$ have a smaller value than all entries not in $A \cup B$ (as they must have values greater than $f(0,0)$). We conclude that $C = A \triangle B$. □

A nice property of polymorphisms is that they are preserved under projections:

Observation 4.9. *Let R be a relation preserved by some k-ary operation f. Then arbitrary projection of R is also preserved by f.*

Proof. Let R' be a projection of R to X and $\overline{t'_1}, \ldots, \overline{t'_k} \in R'$. By the definition of R' there are $\overline{t_1}, \ldots, \overline{t_k} \in R$ such that $\overline{t'_i}$ is a projection of $\overline{t_i}$ for all $i \in [k]$. As R is preserved by f, the tuple $\overline{t} := f(\overline{t_1}, \ldots, \overline{t_k})$ is also in R and hence its projection $\overline{t'}$ to X is in R'. □

4.2 Ord-Horn and ll-closed Constraints

The class of Ord-Horn constraints was introduced by Bürckert and Nebel [22] to identify a tractable class of interval constraints. It is always possible to translate interval constraints into temporal constraints [66]. If the translation of an interval constraint language falls into a tractable temporal constraint language, the interval constraint language is tractable as well. Bürckert and Nebel showed that the class of *interval* constraints having a translation into Ord-Horn temporal constraints is a maximally tractable fragment of Allen's interval algebra. Note that this does not imply that the class of Ord-Horn constraints is a maximally tractable temporal constraint language on time points. Indeed, this is not the case, as we show in this section. Lemma 4.10 below shows that the class of Ord-Horn constraints is ll-closed. Since the relation $R^{\min}$ defined in Definition 4.3 is ll-closed but not Ord-Horn, the class of ll-closed constraints is *strictly* larger than Ord-Horn. Since we prove in Section 4.4 that ll-closed constraints are tractable, Ord-Horn is not a maximally tractable class of temporal constraints. We also present a characterization of ll-closed constraints in terms of structural restrictions on a formula in the conjunctive normal form.

Definition 4.11. *A temporal relation is contained in the temporal constraint language* Ord-Horn *if and only if it can be defined by a conjunction of formulae of the form*

$$(x_1 = y_1 \wedge \cdots \wedge x_k = y_k) \rightarrow x_0 \; O \; y_0 \; ,$$

where $O \in \{=, <, \leq, \neq\}$.

In the following proofs, if Φ is a formula on variables $x_1, \ldots, x_k$ and $\overline{t}$ is a k-tuple representing an assignment of values to $x_1, \ldots, x_k$, we write $\overline{t}[x_i]$ for the entry of $\overline{t}$ corresponding to the variable x_i of Φ. We also write $\Phi(\overline{t})$ for a formula in which each variable is substituted by its value defined by $\overline{t}$.

First, we show that all Ord-Horn temporal relations are preserved by the ll and the dual-ll operations:

Lemma 4.10. *All temporal relations in Ord-Horn are preserved by* ll *and by* dual-ll.

Proof. We give the argument for the ll operation only. The argument for dual-ll is analogous. It suffices to show that every relation that can be defined by a

formula Φ of the form $(x_1 = y_1 \wedge \cdots \wedge x_k = y_k) \to x_0\ O\ y_0$ is preserved by ll, where $O \in \{<, \leq, =, \neq\}$. Let $\overline{t_1}$ and $\overline{t_2}$ be two $2k+2$-tuples that satisfy Φ. Consider a $2k+2$-tuple $\overline{t_3}$ obtained by applying ll componentwise to $\overline{t_1}$ and $\overline{t_2}$. We distinguish two cases: Either there is $i \in [k]$ such that $\overline{t_1}[x_i] \neq \overline{t_1}[y_i]$ or $\overline{t_2}[x_i] \neq \overline{t_2}[y_i]$ — in this case, $x_i = y_i$ is not satisfied in $\overline{t_3}$ as well by the injectivity of ll and therefore the tuple $\overline{t_3}$ satisfies Φ. Or $\overline{t_1}[x_i] = \overline{t_1}[y_i]$ and $\overline{t_2}[x_i] = \overline{t_2}[y_i]$ holds for all $i \in [k]$. But then, as $\overline{t_1}$ and $\overline{t_2}$ satisfy Φ, the literal $x_0\ O\ y_0$ holds in both $\overline{t_1}$ and $\overline{t_2}$. Since ll preserves all relations in $\{<, \leq, =, \neq\}$, the literal $x_0\ O\ y_0$ holds in $\overline{t_3}$, and therefore $\overline{t_3}$ satisfies Φ as well. □

As we already mentioned above, relation $R^{\min}$ is ll-closed but not Ord-Horn. It is a natural question, whether Ord-Horn with addition of $R^{\min}$ is able to express all ll-closed relations. We show that this is not quite the case but Ord-Horn with addition of $R^{\min}_{=} := \{(x,y,z) \mid x > y \vee x > z \vee x = y = z\}$ is able to express all ll-closed relations. Analogously, it is the case that Ord-Horn with $R^{\max}_{=} := \{(x,y,z) \mid x < y \vee x < z \vee x = y = z\}$ is able to express all dual-ll-closed relations. We start by an auxiliary lemma:

Lemma 4.11. *Let R be a k-ary* ll*-closed injective temporal relation. Then R can be defined by a conjunction of formulae of the form $x > y_1 \vee x > y_2 \vee \ldots \vee x > y_k$.*

Proof. Let Φ be a formula in conjunctive normal form defining R. I. e., Φ is a conjunction of clauses, each clause is a disjunction of literals, each literal is of the form $x < y$, $x \leq y$, $x = y$, or $x \neq y$. Because R is injective, any literal of the form $x \neq y$ is always satisfied and hence we can without loss of generality assume that Φ does not contain such literals. Similarly, we can also assume that Φ does not contain literals of the form $x = y$ (as they are never satisfied). If Φ contains a literal of the form $x \leq y$, we can change it to $x < y$ and because of the injectivity of R, we obtain an equivalent formula. So in the following, we assume that Φ contains only literals of the form $x < y$. Among formulae defining Φ, we choose such formulae that have the least number of clauses of the form different from the statement of the lemma. Among these formulae, we choose a formula that has the least number of literals. We call such a formula *reduced*. All these choices are obviously without loss of generality.

Now suppose that Φ has a clause C of the form different from the form described in the statement of the lemma. That means there are literals $\ell_1 := x_1 > y_1$, $\ell_2 := x_2 > y_2$ in C such that x_1 and x_2 are distinct variables. Because Φ is reduced, there is a tuple $\overline{t_1}$ such that ℓ_1 is the only satisfied literal in $C(\overline{t_1})$. Also there is a tuple $\overline{t_2}$ such that ℓ_2 is the only satisfied literal in $C(\overline{t_2})$. If $\overline{t_1}$ can be chosen such that $\overline{t_1}[x_2]$ is smaller than $\overline{t_1}[x_1]$, $\overline{t_1}[y_1]$, and $\overline{t_1}[y_2]$, then we can choose automorphism α of $(\mathbb{Q}, <)$ such that $\overline{t_1}[x_2]$ is mapped to 0. Consider tuple $\overline{t} := \mathrm{ll}(\alpha(\overline{t_1}), \overline{t_2})$. Observe that if in both $\overline{t_1}$ and $\overline{t_2}$ a literal is not satisfied, then it is not satisfied in $\mathrm{ll}(\alpha(\overline{t_1}), \overline{t_2})$, because ll preserves $\leq$. Therefore only literals ℓ_1 or ℓ_2 can be satisfied by $\overline{t}$ in C. As $\overline{t}[x_2]$ is strictly smaller than $\overline{t}[y_2]$, the literal

ℓ_2 cannot be satisfied by $\overline{t}$. Because $\overline{t_2}[x_1] < \overline{t_2}[y_1]$, it also holds that $\overline{t}[x_1] < \overline{t}[y_1]$ and hence ℓ_1 is not satisfied in $\overline{t}$ either. So $\overline{t}$ is not a satisfying assignment for Φ. A contradiction with the fact that Φ expresses an ll-closed relation. Similarly, we can argue that $\overline{t_2}$ cannot be chosen such that $\overline{t_1}[x_1]$ is smaller than $\overline{t_1}[y_1]$, $\overline{t_1}[x_2]$, and $\overline{t_1}[y_2]$. Therefore the formula $\Psi := (x_1 > y_1 \vee x_1 > y_2) \wedge (x_2 > y_1 \vee x_2 > y_2)$ is satisfied for any satisfying assignment of Φ (recall that R is injective). Let Φ' be the formula Φ with clause C removed and with two clauses of Ψ added. Clearly, Φ' has less clauses of the form different from the one described in the statement of the lemma. Therefore if we show that Φ' is equivalent to Φ, we obtain a contradiction with the choice of Φ.

It is easy to see that any satisfying assignment of Φ is also satisfying Φ' — both added clauses are satisfied in any satisfying assignment of Φ and removal of a clause cannot make a satisfying assignment unsatisfying. To show the other direction consider some satisfying assignment $\overline{t}$ of Φ'. Clearly, all the clauses of Φ except for C are satisfied by $\overline{t}$ (as they are also present in Φ'). Tuple $\overline{t}$ also satisfies both clauses of Ψ. We can rewrite Ψ to $(x_1 > y_1 \wedge x_2 > y_1) \vee (x_1 > y_1 \wedge x_2 > y_2) \vee (x_1 > y_2 \wedge x_2 > y_1) \vee (x_1 > y_2 \wedge x_2 > y_2)$. If the first, the second, or the fourth conjunction is satisfied by $\overline{t}$, then $\overline{t}[x_1] > \overline{t}[y_1] \vee \overline{t}[x_2] > \overline{t}[y_2]$ and therefore $C(\overline{t})$ clearly holds. If the third conjunction is satisfied by $\overline{t}$ and the literal ℓ_1 does not hold (i. e., $\overline{t}[x_1] < \overline{t}[y_1]$), we have a chain of inequalities $\overline{t}[y_2] < \overline{t}[x_1] < \overline{t}[y_1] < \overline{t}[x_2]$ and hence $\overline{t}[x_2] > \overline{t}[y_2]$. We conclude that also in this last case C holds. □

In the following, we slightly abuse the notation and write $R^{\min}(x, y_1, \ldots, y_k)$ for a relation $\{(x, y_1, \ldots, y_k) \mid x > y_1 \vee \ldots \vee x > y_k\}$. We also use relation $R^{\min}_{=}(x, y_1, \ldots, y_k)$ defined as $R^{\min}_{=}(x, y_1, \ldots, y_k) := R^{\min}(x, y_1, \ldots, y_k) \vee x = y_1 = y_2 = \ldots = y_k$. First, we check that all these relations are preserved by ll:

Lemma 4.12. *Relations $R^{\min}(x, y_1, \ldots, y_k)$ and $R^{\min}_{=}(x, y_1, \ldots, y_k)$ are preserved by* ll.

Proof. First, we show preservation of $R^{\min}$. Let $\overline{t_1}$, $\overline{t_2}$ be two $k+1$-tuples from $R^{\min}$ and let $\overline{t_3} := \mathrm{ll}(\overline{t_1}, \overline{t_2})$. Since $\overline{t_1} \in R^{\min}$, there is $i \in [k]$ such that $\overline{t_1}[y_i] < \overline{t_1}[x]$. Also there is $j \in [k]$ such that $\overline{t_2}[y_j] < \overline{t_2}[x]$. Therefore $R^{\min}(\overline{t_1}[x], \overline{t_1}[y_i], \overline{t_1}[y_j])$ and $R^{\min}(\overline{t_2}[x], \overline{t_2}[y_i], \overline{t_2}[y_j])$ holds. As we have shown in Lemma 4.2, relation $R^{\min}$ on three variables is preserved by ll and so $R^{\min}(\overline{t_3}[x], \overline{t_3}[y_i], \overline{t_3}[y_j])$ holds too. Therefore $\overline{t_3}$ is from $R^{\min}(x, y_1, \ldots, y_k)$.

Now, we show the preservation of $R^{\min}_{=}$. Let $\overline{t_1}$, $\overline{t_2}$ be two $k+1$-tuples from $R^{\min}_{=}$ and let $\overline{t_3} := \mathrm{ll}(\overline{t_1}, \overline{t_2})$. If $\overline{t_1}, \overline{t_2} \in R^{\min}$, then $\overline{t_3}$ is from $R^{\min}$ and consequently it is also from $R^{\min}_{=}$. If $\overline{t_1}$ is a constant tuple, then the tuple $\overline{t_3}$ is just an automorphic image of $\overline{t_2}$ and therefore $\overline{t_3}$ also satisfies $R^{\min}_{=}$. The same argument holds if $\overline{t_2}$ is a constant tuple and we conclude that $R^{\min}_{=}$ is preserved by ll. □

Now, we are ready to show a general theorem:

Theorem 4.13. *A temporal relation R is* ll*-closed if and only if it can be defined by a conjunction of formulae (which we call* clauses*) of the form*

$$x_1 \neq y_1 \vee \cdots \vee x_k \neq y_k, \text{ or}$$
$$(x_1 = y_1 \wedge \cdots \wedge x_k = y_k) \rightarrow R^{\min}(z_0, z_1, \ldots, z_l), \text{ or}$$
$$(x_1 = y_1 \wedge \cdots \wedge x_k = y_k) \rightarrow R^{\min}_{=}(z_0, z_1, \ldots, z_l),$$

Proof. The implication from right to left is easy to verify. Similarly as in the proof of Lemma 4.10, we just need to argue for a single clause. The clauses of the first type are Ord-Horn and we have already verified the closure in Lemma 4.10. For clauses of the second and the third type, let $\overline{t_1}$ and $\overline{t_2}$ be two $2k+l+1$ tuples satisfying the clause and let $\overline{t_3} := \mathrm{ll}(\overline{t_1}, \overline{t_2})$. If there is $i \in [k]$ such that $\overline{t_1}[x_i] \neq \overline{t_1}[y_i]$ or $\overline{t_2}[x_i] \neq \overline{t_2}[y_i]$, then by the injectivity of ll $\overline{t_3}[x_i] \neq \overline{t_3}[y_i]$ and therefore $\overline{t_3}$ satisfies the clause. Otherwise, since $\overline{t_1}[x_i] = \overline{t_1}[y_i]$ and $\overline{t_2}[x_i] = \overline{t_2}[y_i]$ for all $i \in [k]$, we have that $R^{\min}(z_0, z_1, \ldots, z_l)$ ($R^{\min}_{=}(z_0, z_1, \ldots, z_l)$, respectively) holds in both the tuples. As ll preserves $R^{\min}$ and $R^{\min}_{=}$ by Lemma 4.12, $R^{\min}(\overline{t_3}[z_0], \overline{t_3}[z_1], \ldots, \overline{t_3}[z_l])$ ($R^{\min}_{=}(\overline{t_3}[z_0], \overline{t_3}[z_1], \ldots, \overline{t_3}[z_l])$, respectively) holds as well and therefore $\overline{t_3}$ satisfies the clause.

For the implication from the left to the right, we proceed by induction on the arity n of R. For $n = 2$ the theorem holds as if we identify second and the third argument in $R^{\min}(z_0, z_1, z_2)$ ($R^{\min}_{=}(z_0, z_1, z_2)$), we obtain a relation $z_0 > z_1$ ($z_0 \geq z_1$, respectively). So assume $n > 2$. Let $i, j \in [n], i < j$ be two entries of R. A *contraction of i, j in R* is a relation $R_{ij}(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$ defined as $R(x_1, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n)$ (i. e., we force entries i and j of R to be equal and then project the relation to $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n$). It is straightforward to verify that if R is ll-closed, then R_{ij} is ll-closed too. We also define $\mathrm{inj}(R)$ to be the relation containing all the injective tuples of R. Again, it is straightforward to verify that if R is ll-closed, then so is $\mathrm{inj}(R)$.

We construct the formula Φ defining R as follows: For each pair $1 \leq i < j \leq n$ of entries, let $\Phi_{ij}(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$ be the formula defining R_{ij}. By induction, all its clauses have the form as described in the statement of the theorem. We add to each clause a literal $x_i \neq x_j$ (the form of the clauses clearly remains of the form described in the statement) and add all the modified clauses to Φ. Finally, let Φ_{inj} be the formula defining $\mathrm{inj}(R)$ as stated in Lemma 4.11. We take each clause C of Φ_{inj}. If there is a tuple $\overline{t} \in R$ such that $\overline{t}$ is constant on all variables $z_0, z_1, \ldots, z_{l_C}$ of C, then we add to Φ a clause $C \vee z_0 = z_1 = \ldots = z_{l_C}$ (which is equivalent to $R^{\min}_{=}(z_0, z_1, \ldots, z_{l_C})$). Otherwise we add to Φ a clause C (which is equivalent to $R^{\min}(z_0, z_1, \ldots, z_{l_C})$). It is easy to check that all added clauses are of one of allowed types and therefore Φ has a form described in the statement of the theorem.

What remains to verify is that Φ really defines R. First, let $\overline{t}$ be a k-tuple such that $\overline{t} \notin R$. If $\overline{t}$ is injective, then $\overline{t} \notin \mathrm{inj}(R)$ and hence Φ_{inj} is not satisfied by $\overline{t}$. No literal of the form $z_0 = z_1 = \ldots = z_{l_C}$ can be satisfied by $\overline{t}$ and so Φ is not

satisfied either. If $\overline{t}$ is not injective, there is $1 \leq i < j \leq n$ such that $\overline{t}[i] = \overline{t}[j]$ and $\overline{t_{ij}} := (\overline{t}[1], \ldots, \overline{t}[j-1], \overline{t}[j+1], \ldots, \overline{t}[n]) \notin R_{ij}$. Therefore some clause C of Φ_{ij} is not satisfied by $\overline{t_{ij}}$ and consequently clause $C \vee x_i \neq x_j$ of Φ is not satisfied by $\overline{t}$. So in this case $\overline{t}$ does not satisfy Φ too.

Now, we verify that for all $\overline{t} \in R$, the n-tuple $\overline{t}$ satisfies Φ. If $\overline{t}$ is injective, all the clauses arising from formulae Φ_{ij} are satisfied, because these clauses contain $x_i \neq x_j$, which is satisfied. Because in this case $\overline{t} \in \mathrm{inj}(R)$, $\overline{t}$ also satisfies Φ_{inj} and so $\overline{t}$ satisfies all the clauses of Φ. If $\overline{t}$ is not injective, let C be a clause of Φ created from some clause of Φ_{ij}. If $\overline{t}[i] \neq \overline{t}[j]$, C is satisfied by $\overline{t}$ as C contains $x_i \neq x_j$. If $\overline{t}[i] = \overline{t}[j]$, then $(\overline{t}[1], \ldots, \overline{t}[j-1], \overline{t}[j+1], \ldots, \overline{t}[n]) \in R_{ij}$ and thus this tuple satisfies Φ_{ij}. We also infer that $\overline{t}$ satisfies C. Finally, let C be a clause of Φ created from some clause of Φ_{inj}. The clause C is of the form $x > y_1 \vee \ldots \vee x > y_m$ and possibly has a literal $x = y_1 = \ldots = y_m$. If $\overline{t}$ is constant on the variables of C, then, by the construction of Φ, C contains the literal $x = y_1 = \ldots = y_m$ and so C is satisfied. So suppose $\overline{t}$ is not constant on the variables of C and assume for contradiction that $\overline{t}[x] \leq \overline{t}[y_i]$ for all $i \in [m]$ (otherwise we see that $\overline{t}$ satisfies C). As $\overline{t}$ is not constant on the variables of C, there is $j \in [m]$ such that $\overline{t}[x] < \overline{t}[y_j]$. Because Φ_{inj} was reduced, there is tuple $\overline{t'} \in \mathrm{inj}(R)$ that satisfies only the literal $x > y_j$ in C. Consider tuple $\overline{t''} := \mathrm{lex}(\overline{t}, \overline{t'})$. Because $\overline{t'}$ is injective and lex is an injective operation, the tuple $\overline{t''}$ is injective too. Since $\overline{t}[x] \leq \overline{t}[y_i]$ for all $i \in [m]$, $\overline{t'}[x] < \overline{t'}[y_i]$ for all $i \in [m] \setminus \{j\}$, and lex preserves $\leq$, we get that $\overline{t''}[x] < \overline{t''}[y_i]$ for all $i \in [m] \setminus \{j\}$. Finally, as $\overline{t}[x] < \overline{t}[y_j]$, we also have that $\overline{t''}[x] < \overline{t''}[y_j]$ by the properties of lex. Hence, $\overline{t''}[x] < \overline{t''}[y_i]$ for all $i \in [m]$ and therefore $\overline{t''}$ does not satisfy C and thus Φ_{inj}. But $\overline{t''}$ is injective and is from R (recall that R is ll-closed) and therefore $\overline{t''}$ satisfies $\mathrm{inj}(R)$. A contradiction with the fact that Φ_{inj} defines $\mathrm{inj}(R)$. □

Note that the above theorem gives us a nice list of relations, that are enough to express all ll-closed relations. Actually, we do not need the whole power of primitive positive definitions, because we do not use existential quantification. If we use existential quantification, our list of relations can be even shorter:

Corollary 4.14. *A temporal relation R is* ll*-closed if and only if it can be expressed in a temporal constraint language Γ containing relations $<$, $R^{\min}_{=}(z_0, z_1, z_2)$ and $E(x_1, y_1, z_1, y_2) := \{(x_1, y_1, z_1, z_2) \mid x_1 \neq y_1 \vee z_1 = z_2\}$.*

Proof. Clearly, the first two relations can be defined by formulae of the form described in Theorem 4.13. The third relation can be defined by

$$(x_1 = y_1 \rightarrow R^{\min}_{=}(z_1, z_2, z_2)) \wedge (x_1 = y_1 \rightarrow R^{\min}_{=}(z_2, z_1, z_1))$$

Therefore all the relations in Γ are by Theorem 4.13 ll-closed and consequently all relations expressible in Γ are ll-closed by Theorem 2.9.

For the other implication, it is enough to show that all the relations defined by clauses of the type described in Theorem 4.13 can be expressed in Γ. First, we

show that relation $F(x_1, y_1, x_2, y_2, z_1, z_2) := \{(x_1, y_1, x_2, y_2, z_1, z_2) \mid (x_1 = y_1 \wedge x_2 = y_2) \rightarrow z_1 = z_2\}$ can be expressed in Γ. Let u_1, u_2, u_3 be three new variables. We impose constraints $E(x_1, y_1, u_1, u_2)$, $E(x_2, y_2, u_2, u_3)$, and $E(u_1, u_3, z_1, z_2)$. Clearly, if $x_1 = y_1$ and $x_2 = y_2$, then $u_1 = u_2 = u_3$ and so $z_1 = z_2$. On the other hand, if $x_1 \neq y_1$ or $x_2 \neq y_2$, there are values for u_1, u_2, u_3 that satisfy all the constraints and $u_1 \neq u_3$. Hence, values of z_1 and z_2 are unrestricted. We have just shown that described instance expresses F. Now suppose R is defined by $x_1 \neq y_1 \vee \ldots \vee x_k \neq y_k$. We prove that R can be expressed in Γ by induction on k. If $k = 1$, we can express R by $E(x_1, y_1, z, z') \wedge z < z'$, where z, z' are two new variables. Now assume $k > 1$ and we can express all relations of this type for smaller k. Let z, z' be two new variables and force $z \neq z' \vee x_3 \neq y_3 \vee \ldots \vee x_k \neq y_k$ (we can express this relation by induction) and $F(x_1, y_1, x_2, y_2, z, z')$. It is straightforward to check that the relation expressed on $x_1, y_1, \ldots, x_k, y_k$ is R.

Next, we show that $R^{\min}_{=}(z_0, \ldots, z_l)$ can be expressed for every $l \geq 2$. Again, we proceed by induction on l. For $l = 2$ the statement is obvious as we have $R^{\min}_{=}(z_0, z_1, z_2)$. So let us assume that $l > 2$ and that we can express all relations of this type for smaller l. We express $R^{\min}_{=}(z_0, \ldots, z_l)$ by $R^{\min}_{=}(z_0, z_1, x) \wedge R^{\min}_{=}(x, z_2, \ldots, z_l)$. It is easy to check that whenever $z_0, \ldots, z_l$ satisfies $R^{\min}_{=}$, we can choose a value of x to satisfy also our formula. On the other hand, if our formula is satisfied, then either $z_1 < z_0$, $x < z_0$, or $z_0 = z_1 = x$. In the first case, $R^{\min}_{=}(z_0, \ldots, z_l)$ is satisfied. In the second case, as $R^{\min}_{=}(x, z_2, \ldots, z_l)$ holds, it follows there is $2 \leq i \leq l$ such that $z_i \leq x$ and therefore $R^{\min}_{=}(z_0, \ldots, z_l)$ is satisfied too. In the third case, it either holds that $x = z_2 = \ldots = z_l$ and $R^{\min}_{=}(z_0, \ldots, z_l)$ is satisfied or there is $2 \leq i \leq l$ such that $z_i < x = z_0$ and so $R^{\min}_{=}(z_0, \ldots, z_l)$ is satisfied as well.

It is easy to see that $R^{\min}(z_0, z_1, z_2)$ is expressed by $R^{\min}_{=}(z_0, z_1, z_2) \wedge (z_0 \neq z_1 \vee z_0 \neq z_2)$. For larger l, we express $R^{\min}(z_0, \ldots, z_l)$ by $R^{\min}(z_0, z_1, x) \wedge R^{\min}(x, z_2, \ldots, z_l)$. It is straightforward to check that this expression is correct.

Finally, we show how to express relation $(x_1 = y_1 \wedge \ldots \wedge x_k = y_k) \rightarrow R(z_0, \ldots, z_l)$ if we are able to express $R(z_0, \ldots, z_l)$. Let $x'_2, \ldots, x'_k$, $y'_2, \ldots, y'_k$ and $z'_0, \ldots, z'_l$ be new variables. We impose constraints $F(x_1, y_1, x_2, y_2, x'_2, y'_2)$, and $F(x'_i, y'_i, x_{i+1}, y_{i+1}, x'_{i+1}, y'_{i+1})$ for all $2 \leq i \leq k-1$. From the definition of F it follows that if $x_i = y_i$ for all $i \in [k]$, then also $x'_k = y'_k$. We further impose constraints $R(z'_0, \ldots, z'_l)$ and $E(x'_k, y'_k, z_j, z'_j)$ for all $0 \leq j \leq l$. Therefore if $x_i = y_i$ for all $i \in [k]$, then $z_j = z'_j$ for all $0 \leq j \leq l$ and so R is imposed on $z_0, \ldots, z_l$. If there is $i \in [k]$ such that $x_i \neq y_i$, there exists satisfying assignment to all imposed constraints with F as a constraint relation where $x'_k \neq y'_k$. Therefore $z_0, \ldots, z_l$ are not constrained by any of the newly imposed constraints and we conclude that the newly imposed constraints really express relation $(x_1 = y_1 \wedge \ldots \wedge x_k = y_k) \rightarrow R(z_0, \ldots, z_l)$. □

4.3 ll-closed Constraints and Datalog

In this section, we introduce Datalog, and then prove that the constraint satisfaction problem for ll-closed constraints cannot be solved by Datalog programs. This result should not be confused with the weaker fact that establishing k-consistency does not imply global consistency for any k. This was shown for Ord-Horn in [69]. But recall that Ord-Horn *can* be solved by a Datalog program [22].

All constraint satisfaction problems studied in the literature so far where one can show that they cannot be solved by Datalog have the *ability to count* [40]. It is easy to verify that the temporal constraint language that only contains the ternary relation $R^{\min}$ does *not* have the ability to count. However, we present a proof that CSP($R^{\min}$) cannot be solved by a Datalog program.

We now define Datalog. Our definition will be purely operational; for the standard semantical approach to the evaluation of Datalog programs see [38]. A Datalog program is a finite set of Horn clauses, i. e., clauses of the form $\psi \leftarrow \phi_1, \ldots, \phi_l$, where $l \geq 1$ and where $\psi, \phi_1, \ldots, \phi_l$ are atomic formulae of the form $R(\overline{x})$. The formula ψ is called the *head* of the rule, and $\phi_1, \ldots, \phi_l$ are called the *body*. We assume that all variables in the head also occur in the body. The relation symbols occurring in the head of some clause are called *intentional*, and all other relation symbols in the clauses are called *extensional*.

If Γ is a finite temporal constraint language, we might use Datalog programs to solve CSP(Γ) as follows. Let Π be a Datalog program whose extensional symbols are from Γ, and let L be the set of intentional relation symbols of Π. We assume that there is one distinguished 0-ary intentional relation symbol `false`. Now suppose we are given an instance I of CSP(Γ). An *evaluation* of Π on I proceeds in steps $i = 0, 1, \ldots$. At each step i, we maintain a set of literals I^i with relation symbols from L; it always holds that $I^i \subset I^{i+1}$. Each clause of Π is understood as a rule that may derive a new literal (with a relation symbol from L) from the literals in I^i. Initially, we have $I^0 := I$. Now suppose that $R_1(x_1^1, \ldots, x_{k_1}^1), \ldots, R_l(x_1^l, \ldots, x_{k_l}^l)$ are literals in I^i, and $R_0(y_1^0, \ldots, y_{k_0}^0) \leftarrow R_1(y_1^1, \ldots, y_{k_1}^1), \ldots, R_l(y_1^l, \ldots, y_{k_l}^l)$ is a rule from Π, where $y_j^i = y_{j'}^{i'}$ if and only if $x_j^i = x_{j'}^{i'}$. Then $R_0(x_1^0, \ldots, x_l^0)$ is the newly derived literal in I^{i+1}, where $x_j^0 = x_{j'}^0$ if and only if $y_j^0 = y_{j'}^0$. The procedure stops if no new literal can be derived. We say that Π solves CSP(Γ), if for every instance I of CSP(Γ) there exists an evaluation of Π on I that derives `false` if and only if I has no solution.

We want to remark that the so-called method of *establishing path-consistency*, which is very well-known and frequently applied in Artificial Intelligence, can be formulated with Datalog programs where the intentional symbols are at most binary and all rules use at most three variables in the body.

We prove that already for the temporal language that only consists of $R^{\min}$ there is no Datalog program that solves the corresponding constraint satisfaction problem. We use a pebble-game characterization of the expressive power of

Datalog, which was originally shown in [40] and [68] for finite domain constraint satisfaction, and which holds for a wide variety of infinite domain constraint languages as well, including qualitative temporal constraint satisfaction (see the journal version of [13]).

Let Γ be a temporal constraint language, and let I be an instance of $\mathrm{CSP}(\Gamma)$. Then the *existential k-pebble game on I* is the following game between the players *Spoiler* and *Duplicator*. Spoiler has k pebbles $p_1, \ldots, p_k$. He places his pebbles on variables from I. Initially, no pebbles are placed. In each round of the game, Spoiler picks some of these pebbles. If they are already placed on I, then Spoiler first removes them from I. He then places the pebbles on variables from I, and Duplicator responds by assigning elements from $\mathbb{Q}$ to these variables. This assignment has to satisfy all the constraints C from I where all variables in C are pebbled, otherwise Spoiler wins the game. Duplicator wins, if the game continues *forever*, i. e., if Spoiler can never win the game. Note that if Spoiler removes some pebble from I, we can also erase the value assigned to the corresponding variable (as its value plays any role only when a pebble is placed on it again and at that moment, we can assign it a new value).

Theorem 4.15. *[13] Let Γ be a temporal constraint language. There is no Datalog program that solves $\mathrm{CSP}(\Gamma)$ if and only if for every k there exists an inconsistent instance I of $\mathrm{CSP}(\Gamma)$ such that Duplicator wins the existential k-pebble game on I.*

The rest of this section is devoted to the proof of the following theorem.

Theorem 4.16. *There is no Datalog program that solves $\mathrm{CSP}(R^{\min})$.*

Proof. Let k be an arbitrary number. To apply Theorem 4.15, we have to construct an inconsistent instance I of $\mathrm{CSP}(R^{\min})$ such that Duplicator wins the existential k-pebble game on I.

For this, let G be a 4-regular graph of girth $2k+1$, i. e., all cycles in G have more than $2k$ vertices. It is known and easy to see that such graphs exist; it is even known that there are such graphs of size exponential in k [56]. Orient the edges in G such that there are exactly two outgoing and two incoming edges for each vertex in G. Since G is 4-regular, there exists an Euler tour for G (see e. g. [32]), which shows that such an orientation exists.

Now, we can define our instance I of $\mathrm{CSP}(R^{\min})$ as follows. The variables of I are the vertices from G. The instance I contains the constraint $R^{\min}(w, u, v)$ if and only if uw and vw are the two incoming edges at vertex w. We claim that I does not have a solution: If there was a solution, some variable w must denote the minimal value. But for every variable w we find a constraint $R^{\min}(w, u, v)$ in I, and this constraint is violated since either u or v must be strictly smaller than w.

Now, we show that Duplicator has a winning strategy for the existential k-pebble game on this instance. Let r be a vertex of G. Consider a subgraph G'_r of

G induced by vertices of G from which there is an oriented path of length at most k to r. Because G has a girth $2k+1$, G'_r is a binary (every vertex has outdegree zero or two) tree. We call a binary subtree of G'_r a *dominated tree for* r if it has the root r and all its leaves are pebbled. Since there are at most k pebbles in G, a dominated tree for any vertex has at most $2k$ vertices. Conversely, any binary tree in G that has all its leaves pebbled is a dominated tree for its root as it can have at most k leaves and thus can have depth at most k.

Duplicator always maintains the property that whenever the root r in a dominated tree for r is pebbled during the game, then the value assigned to r is strictly larger than the minimum of all the values assigned to the leaves. Clearly, this property is satisfied at the beginning of the game.

Suppose that during the game Spoiler pebbles the variable u. Let $T_1, \ldots, T_s$ be those newly created dominated trees for $r_1, \ldots, r_s$ that have variable u as a leaf and have pebbled their roots $r_1, \ldots, r_s$, for $s \geq 0$. If $s > 0$, let r_i be the root that received the minimal value a among all the roots $r_1, \ldots, r_s$. We claim that if u is the root of a dominated tree T, then a is strictly larger than the minimum b of all the values assigned to the leaves of T. Otherwise, the graph $T \cup T_i$ was a dominated tree that violates the invariant even before the variable u has been pebbled, a contradiction. Therefore in this case Duplicator can choose a value c between b and a for the variable u. Since c is smaller than a, in all the new dominated trees $T_1, \ldots, T_s$ in G the value assigned to $r_1, \ldots, r_s$ is strictly larger than c, and hence the invariant is preserved. Particularly, if $R^{\min}(w, u, v)$ (or $R^{\min}(w, v, u)$) is a constraint in Φ where w and v have been pebbled, then this constraint is satisfied by the assignment.

Since c is larger than b, this choice also guarantees that if v, v' are pebbled variables, then any constraint of the form $R^{\min}(u, v, v')$ is satisfied, because in this case the variables u, v, v' induce a dominated tree with the root u in G.

If there is no dominated tree T for u, then Duplicator assigns a value to u that is smaller than all values assigned to other variables. The remaining case is that $s = 0$. In this case, Duplicator plays a value that is larger than all values assigned to other variables. In both cases, it is easy to check that Duplicator maintains the invariant and satisfies all constraints C of I where all variables are pebbled. By induction, we have shown that Duplicator has a winning strategy for the existential k-pebble game on Φ. □

4.4 An Algorithm for ll-closed Languages

In this section, we present an algorithm for ll-closed constraints. It is straightforward to dualize all arguments and the algorithm, and we will therefore also obtain an algorithm for dual-ll-closed constraints.

One of the underlying ideas of the algorithm is to use a subroutine that tries to find a solution where every variable has a different value. If this is impossible, the

subroutine should return a set of at least two variables that denote the same value in all solutions. It is one of the fundamental properties of ll-closed constraints that this is always possible.

To formally introduce our algorithm, we need the following definition. Let $I = (V, D, \mathcal{C})$ be an instance of CSP(Γ) and $X \subset V$. A *projection of* I to $V \setminus X$ is an instance of CSP(Γ), $I' = (V \setminus X, D, \mathcal{C}')$, where $\mathcal{C}' = \{p_{V \setminus X}(C) \mid C \in \mathcal{C}\}$. Observe that any operation preserving all constraint relations in I also preserves all constraint relations in I' (by Observation 4.9). Also note that if I' does not have a solution, then I does not have a solution either.

Because lex provides min-intersection closure, any relation R preserved by lex has intersection-closed min-sets (by Lemma 4.6). Hence, if the i-th entry has min-sets $S_1, \ldots, S_l$, for $l \geq 1$, then there is some $j \in [l]$ such that S_j is a subset of all min-sets for the i-th entry. We call such a min-set the *minimal min-set* for the i-th entry in R.

Lemma 4.17. *Let R be a k-ary relation preserved by* lex, *$\overline{t} \in R$, $i \in [k]$, and let S be the minimal min-set for the i-th entry in R. If $\overline{t}[j] \geq \overline{t}[i]$ for every $j \in S$, then $\overline{t}[i] = \overline{t}[j]$ for every $j \in S$.*

Proof. Let $\overline{t'} \in R$ be the tuple that witnesses the minimal min-set S. Suppose there is a tuple $\overline{t} \in R$ such that not all entries in S are equal in $\overline{t}$ (in particular, $|S| > 1$). Consider the tuple $\overline{t''} := lex(\overline{t'}, \overline{t})$. By the properties of lex it holds that $\overline{t''}[i] < \overline{t''}[j]$ for every $j \in [k] \setminus S$. Furthermore, $\overline{t''}[i] \leq \overline{t''}[j]$ for $j \in S$ if and only if $\overline{t}[i] \leq \overline{t}[j]$ for all $j \in S$. Thus, unless $\overline{t''}$ witnesses a smaller min-set for i in R (which would be a contradiction), we have that $\overline{t''}[i] > \overline{t''}[j]$ for some $j \in S$ and so $\overline{t}[i] > \overline{t}[j]$. □

To develop our algorithm, we use a specific notion of a *constraint graph* of a temporal CSP instance, defined as follows:

Definition 4.12. *The* constraint graph *G_I of a temporal* CSP *instance I is a directed graph (X, E) defined on the variables X of I. For each constraint of the form $R(x_1, \ldots, x_k)$ from I, we add a directed edge $x_i x_j$ to E if in every tuple from R where the i-th entry is minimal the j-th entry is minimal as well.*

Note that if the i-th entry is not minimal in any tuple, there are directed edges from x_i to all x_j, $j \in [k]$.

Example 4.2. *Let $R(x, y, u, v)$ be the 4-ary temporal relation defined by $(x{=}y \wedge y{<}u \wedge u{=}v) \vee (x{<}y \wedge y{<}u \wedge u{<}v)$. Consider an instance $I := (\{x_1, x_2, y_1, y_2, y_3\}, \mathbb{Q}, \{R(x_1, x_2, y_1, y_2), R(x_1, x_2, y_2, y_3), R(x_1, x_2, y_3, y_1)\})$. The constraint graph G_I for the instance I has vertices x_1, x_2, y_1, y_2, y_3, edges from y_1, y_2, y_3 to all other variables, and an edge from x_2 to x_1.*

Definition 4.13. *If I contains a constraint C imposed on y such that C does not admit a solution where y denotes the minimal value, we say that y is* blocked (by C).

We would like to use the constraint graph to identify variables that have to denote the same value in all solutions, and therefore introduce the following concepts.

Definition 4.14. *A strongly connected component K of the constraint graph G_I for a temporal* CSP *instance Φ is called a* sink component *if no edge in G_I leaves K, and no variable in K is blocked. A vertex of G_I that belongs to a sink component of size one is called a* sink*.*

In the previous example, the variables y_1, y_2, y_3 are blocked, and x_1 and x_2 are not blocked. The set of vertices $\{y_1, y_2, y_3\}$ forms a strongly connected component, which is not a sink component, because there are outgoing edges. Moreover, the variables in K are blocked. The singleton-set $\{x_1\}$ is a strongly connected component without outgoing edges and without blocked vertices, and thus x_1 is a sink.

The following lemma shows an important consequence of lex-closure of constraints.

Lemma 4.18. *Let K be a sink component of the graph G_I for an instance I with* lex-*closed constraints. Then all variables from K must have the same value in all solutions of I.*

Proof. We assume that I has a solution, and that K has at least two vertices (otherwise the lemma is trivial). Let $\overline{t}$ be a solution of I, and let $M \subseteq K$ be the set of variables that have in $\overline{t}$ the minimal value among the variables of the sink component K. If $M = K$, we are done.

Otherwise, because K is a strongly connected component, there is an edge in G_I from some vertex $u \in M$ to some vertex $v \in K \setminus M$. By the definition of G_I, there is a constraint C in I such that whenever u denotes the minimal value of a solution of C, then v has to denote the minimal value as well. By permuting arguments, we can assume without loss of generality that C is of the form $R(w_1, \ldots, w_k)$ where $w_1 = u$ and $w_2 = v$. Because K is a sink component, the variable u cannot be blocked, and hence there is a minimal min-set S for the first entry in R. Clearly, S contains 2, because v is the second argument of C and uv is an edge of G_I.

Note that G_I contains an edge from u to w_i for all $i \in S$. Since K is a sink component, all these variables w_i are in K. Because u has in $\overline{t}$ the minimal value among the variables in K, there is no variable w_i, $i \in S$, which has a smaller value than u in $\overline{t}$. This contradicts Lemma 4.17, because the value for u in $\overline{t}$ is different than the value for v. □

Lemma 4.18 immediately implies that we can add constraints of the type $x = y$ for all variables x, y from the same sink component K. Equivalently, we can consider the CSP instance I' where all the variables in K are *contracted*, i. e., where all variables from K are replaced by the same variable. In some

cases, a solution to a projected instance with ll-closed constraints can be used to construct a solution to the original instance.

Lemma 4.19. *Let I be an instance of the* CSP *with variables X and* ll*-closed constraints. Let x be a sink in G_I. If the projection I' of I to $X \setminus \{x\}$ has an injective solution, then I has an injective solution as well.*

Proof. Let $\overline{s}$ be an injective solution to I'. Consider a constraint C from I that is imposed on x. By the definition of I', there is a tuple $\overline{t} \in R(C)$ such that $\overline{t}$ agrees with $\overline{s}$ on $V(C) \setminus \{x\}$. Because x is a sink, there is a tuple $\overline{t'} \in R(C)$ such that the entry corresponding to x is strictly minimal. Let β be an automorphism of $(\mathbb{Q}, <)$ mapping $\overline{t'}[x]$ to 0. It is now easy to check that there is an automorphism α of $(\mathbb{Q}, <)$ such that the tuple $\overline{t''} := \alpha(ll(\beta(\overline{t'}), \overline{t}))$ agrees with $\overline{s}$ on $X \setminus \{x\}$, and the entry $\overline{t''}[x]$ is strictly minimal. As $R(C)$ is ll-closed, $\overline{t''} \in R(C)$. Thus we see that for each constraint C imposed on x, there is a tuple in $R(C)$ where the entry corresponding to x is strictly minimal, and the rest of the tuple agrees with $\overline{s}$ on $X \setminus \{x\}$. Hence, we can extend $\overline{s}$ by assigning to x a value smaller than any value used in $\overline{s}$, and the lemma readily follows. □

We are ready to state our algorithm for instances with ll-closed constraints.

Algorithm 4.1.

```
Spec(I) {
// Input: Instance I with variables X
// Output: If I has no solution, then return false.
//   If I has an injective solution, then return true.
//   Otherwise return S ⊆ X, |S| ≥ 2, s.t. for all
//     x, y ∈ S we have x = y in all solutions of I.
  G := ConstructGraph(I)
  Y := ∅, I' := I
  while G contains a sink s do begin
    Y := Y ∪ {s}
    I' := projection of I' to X \ Y
    G := ReconstructGraph(I')
  end
  if Y = X then
    return true
  else if G has a sink component S then
    return S
  else
    return false
}
```

Algorithm 4.2.

```
Solve(I):  {
// Input: Instance I with variables X
// Output: true or false
  S := Spec(I)
  if S = false then
    return false
  else if S = true then
    return true
  else begin
    Let I' be the contraction of S in I
    return Solve(I')
  end
}
```

Theorem 4.20. *The procedure* `Solve`(I) *in Algorithm 4.2 decides whether a given instance I of a constraint satisfaction problem with* ll*-closed constraints has a solution. There is an implementation of the algorithm that runs in time $O(nm)$, where n is the number of variables of I and m is the weighted number of constraints in I.*

Proof. The correctness of the procedure `Spec` immediately implies the correctness of the procedure `Solve`. In the procedure `Spec`, after iterated deletion of sinks in G, we have to distinguish three cases.

In the first case, $Y = X$. We inductively construct an injective solution of I as follows. Let $x_1, \dots, x_n$ be the elements from Y in the order in which they were included into Y. For $0 \leq i \leq n$, let I_i be the instance I projected to $X \setminus \{x_1, \dots, x_i\}$. Note that $I_0 = I$, and that $I_n = I'$ is the projection of I to the empty set, which trivially has an injective solution. So assume $i < n$ and I_j has an injective solution for all $i < j \leq n$. Then Lemma 4.19 applied to x_i, the instance I_i, and the injective solution to I_{i+1} implies that also I_i has an injective solution. By induction, I_i has an injective solution for all $0 \leq i \leq n$, and in particular $I_0 = I$ has an injective solution. Therefore, the output `true` of `Spec` is correct.

Otherwise, in the second case, G contains a sink component S with $|S| \geq 2$. We claim that for all variables $x, y \in S$ we have $x = y$ in all solutions to I. Lemma 4.18 applied to the projection of I to $X \setminus Y$ implies that whenever some variables are in the same sink component, they must have the same value in every solution, and hence the output is correct in this case as well.

In the third case, $Y \neq X$, but G does not contain a sink component. Note that in every solution to I' some variable must denote the minimal value. However, by the construction of G, if x gets the minimal value, then all variables reachable from x must get the same value and thus all vertices of some sink component

must get the minimal value. Since each strongly connected component without outgoing edges contains a blocked vertex, there is no variable that can denote the minimal value, and hence I' has no solution. Because I' is a projection of I to $X \setminus Y$, the instance I is inconsistent as well.

Since in each recursive call of `Solve`, the instance in the argument has at least one variable less, `Solve` is executed at most n times. It is not difficult to implement the algorithm such that the total running time is cubic in the input size. However, it is possible to implicitly represent the constraint graph and to implement all sub-procedures such that the total running time is in $O(nm)$. We describe an implementation of the procedure `Spec` that is linear in the input size.

The constraint graph G_I is not computed explicitly, but instead we create the following data structure in the beginning of the procedure `Spec`. For each constraint in the instance, we sort the tuples according to their number of different values. Now, the data structure contains for each variable v and each constraint that is imposed on v a reference to the tuple $\overline{t}$ in this constraint such that v has the least value in $\overline{t}$ and $\overline{t}$ has the largest number of different values. Moreover, for each value in each tuple, we create a list that contains the entries where this value appears in the tuple. Finally, for each variable v, we also have a list that contains the constraints that are imposed on v and block v. With bucket sort, the total time to set up this data structure is linear in the input size. Even though the constraint graph G is not explicitly represented, it is possible to use the above data structure to compute the strongly connected components of G in linear time, using depth-first search.

Now, we describe how the algorithm finds sinks, how the data structure is updated after projections, and how the algorithm finds sink components if there is no sink left and not all variables have been projected out. To find sink and sink components, we also have to be able to determine efficiently whether a node is blocked or not.

Initially, because we have computed the strongly connected components, and because we know which variables are blocked, we can create a list that contains all sinks of the initial instance. Suppose that s is a sink of G at some iteration of the while-loop. We then first compute the projection of I' to $X \setminus Y$ by updating only the constraints imposed on s in I'. If new unblocked sinks are created by removing s, we add them to the list of sinks. At this step, we can also determine whether a constraint no longer blocks a variable v (a new value became minimal in some tuple $\overline{t}$ after projecting out s and v is among the entries with this value in $\overline{t}$), and in this case we can update the list of blocking constraints for v. As soon as this list becomes empty, we know that v is no longer blocked. In this case, if v does not have any outgoing edges in the current constraint graph, which we can determine efficiently using our updated data structure, we add v to the list of sinks. The total number of operations we have to perform in all iterations of the while-loop is then bounded by m.

Finally, if there is no sink left, but not all variables have been projected

out, then we can compute the strongly connected components of the resulting constraint graph (again, this can be done in linear time using depth-first search on our data structure), and since we know which variables are blocked, we can also find the sink components.

Note that we can assume that n is smaller than m. Otherwise, the instance is not connected (we use the notion of connectivity for instances of the CSP as e. g. in [53]). We can in this case use the same implementation, analyze the running time for each of the connected components separately, and will get the same result. This shows that the algorithm can be implemented so that it runs in time $O(nm)$. □

4.5 An Algorithm for Some pp-closed Languages

In this section, we present a polynomial time algorithm that solves constraint satisfaction problem for languages that are pp-closed and also closed under an operation providing a min-intersection closure, a min-union closure, or a min-xor closure. Although at the first sight the languages closed under pp and an operation providing a min-intersection closure may seem to be a subclass of ll-closed languages (recall that we have shown in Lemma 4.5 that there is a relation preserved by the ll operation but not by the pp operation), this is not the case. Consider for example operation f defined as:

$$f(x,y) := \begin{cases} \alpha(\min(x,y)) & \text{if } x \neq y \\ \beta(x) & \text{if } x = y \end{cases}$$

where α and β are two automorphisms of $(\mathbb{Q}, <)$ such that $\alpha(x) > \beta(x) > \alpha(x-\varepsilon)$ for all $x \in \mathbb{Q}$ and all $0 < \varepsilon \in \mathbb{Q}$ (see Figure 4.3). This operation provides a min-intersection closure and there is a relation $R(x_1,x_2,x_3,x_4) := \{(x_1,x_2,x_3,x_4) \mid x_1 = x_2 < x_3 < x_4 \vee x_1 > x_2 > x_3 = x_4 \vee x_1 = x_2 < x_3 = x_4 \vee x_1 = x_2 > x_3 = x_4\}$, which is pp-closed and also closed under operation f. But it is straightforward to check that this relation is not even lex-closed and therefore it cannot be ll-closed. So the class of ll-closed constraint languages is incomparable to the class of languages that are pp-closed and have a polymorphism providing min-intersection closure. As usual, we concentrate only on pp and polymorphisms providing min-intersection/xor/union closure properties. All the arguments can be easily reformulated for dual-pp and polymorphisms providing max-intersection/xor/union closure properties.

Algorithms for these languages have a few similar ideas as the algorithm for ll-closed languages. We also look for a variable that can have the minimal value in a solution. Unlike the algorithm for ll-closed languages, we impose all equalities and inequalities that are forced by such variable being minimal and then consider how the rest of the solution can look like on the other variables. We later show

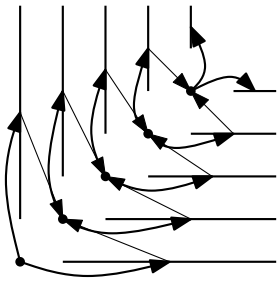


Figure 4.3: An operation providing min-intersection closure.

that if the instance has a solution, it also has a solution that satisfies all these additional constraints.

To formally introduce our algorithm, we need the following definition. Let $I = (V, D, \mathcal{C})$ be an instance of CSP(Γ) and $X \subset V$. A *ordered projection of* I to $V \setminus X$ is an instance of CSP(Γ), $I' := (V \setminus X, D, \mathcal{C}')$, where $\mathcal{C}' := \{p^{<}_{V \setminus X}(C) \mid C \in \mathcal{C}\}$. Observe that any operation preserving all constraint relations in I and relation $<$ also preserves all constraint relations in I' (by Observation 4.9 and the fact that if an operation preserves two relations, it also preserves their conjunction). Therefore the instance I' has also all constraints pp-closed and closed under an operation providing some min- closure property.

Let $I = (X, \mathbb{Q}, \mathcal{C})$ be an instance of CSP over $\mathbb{Q}$. We call a set of variables $S \subset X$ a *min-set of* I, if for all constraints $C \in \mathcal{C}$ satisfying $S \cap V(C) \neq \emptyset$, the set $S \cap V(C)$ is a min-set of $R(C)$.

Now, we show a lemma demonstrating usefulness of min-sets:

Lemma 4.21. *Let $I = (X, \mathbb{Q}, \mathcal{C})$ be an instance of temporal* CSP *such that all constraint relations are closed under* pp *operation and let S be a min-set of I. Then I has a solution if and only if the ordered projection I' of I to $X \setminus S$ has a solution.*

Proof. First suppose I' has a solution $\overline{s'}$ and consider some constraint $C \in \mathcal{C}$ such that $V(C) \cap S \neq \emptyset$ and $V(C) \not\subseteq S$. By the definition of ordered projection, there is a tuple $\overline{t_1} \in R(C)$ that coincides with $\overline{s'}$ on $V(C) \cap (X \setminus S)$. Also by the definition of min-set, there is a tuple $\overline{t_2} \in R(C)$ that witnesses a min-set $V(C) \cap S$ of $R(C)$. Let α be an automorphism of $(\mathbb{Q}, <)$ that maps the minimal value of $\overline{t_2}$ to 0. As the relation $R(C)$ is pp-closed, the tuple $\overline{t_3} := \mathrm{pp}(\alpha(\overline{t_2}), \overline{t_1})$ is in $R(C)$. It is easy to check that this tuple also witnesses the min-set $V(C) \cap S$ and after mapping by a proper automorphism it also coincides with $\overline{s'}$ on $V(C) \cap (X \setminus S)$. As we can find such tuple for all the constraints in $\mathcal{C}$ such that $V(C) \cap S \neq \emptyset$, we conclude that a solution $\overline{s'}$ of I' can be extended to a solution $\overline{s}$ of I by setting all the variables of S to a value smaller than any other value in $\overline{s'}$. Clearly, all the constraints in $\mathcal{C}$ with $V(C) \cap S = \emptyset$ or $V(C) \subset S$ are satisfied by $\overline{s}$ too.

Now suppose that I has a solution $\overline{s}$. We show that it has a solution $\overline{s'}$ where S is the set of variables with minimal value and s' coincides with $\overline{s}$ on $X \setminus S$. Such solution $\overline{s'}$ restricted to $X \setminus S$ is a solution to the ordered projection I' of I to $X \setminus S$ as it satisfies all the inequalities imposed by the ordered projection. So we only have to verify that $\overline{s'}$ really is a solution of I. Let $C \in \mathcal{C}$ be a constraint of I such that $V(C) \cap C \neq \emptyset$ (all other constraints are satisfied by $\overline{s'}$ trivially) and $\overline{t_1}$ be $\overline{s}$ restricted to $V(C)$. Clearly, $\overline{t_1} \in R(C)$ as $\overline{s}$ is a solution of I. By the definition of the min-set S, there is $\overline{t_2} \in R(C)$ witnessing the min-set $S \cap V(C)$ of $R(C)$. Similarly as in the previous paragraph, let α be an automorphism of $(\mathbb{Q}, <)$ that maps the minimal value of $\overline{t_2}$ to 0. Because the relation $R(C)$ is pp-closed the tuple $\overline{t_3} := \mathrm{pp}(\alpha(\overline{t_2}), \overline{t_1})$ is in $R(C)$. It is easy to check that this tuple witnesses the min-set $V(C) \cap S$ and after mapping by a proper automorphism it also coincides with $\overline{s}$ on $X \setminus S$. Therefore C is satisfied by $\overline{s'}$. □

The above lemma basically asserts that if we are able to identify a min-set for every instance of CSP from some language in polynomial time, we have a polynomial time algorithm for finding a solution of those instances. The algorithm is as follows:

Algorithm 4.3.

```
Solve(I) {
// Input: Instance I of a temporal CSP with variables X and
//   constraints C whose constraint relations are pp-closed
// Output: A solution s to I or false if it does not exist.
  i := 0
  while X ≠ ∅ do begin
    S := FindMinSet(I)
    if S = false then
      return false
    for each x ∈ S do
      s[x] := i
    i := i + 1
    I := ordered projection of I to X \ S
  end
  return s
}
```

We only require that the function `FindMinSet` returns a min-set of I if I has a solution (otherwise it can return either a min-set or `false`) and that all constraint relations of I are pp-closed. The running time of the algorithm is $O(n \cdot (t(\texttt{FindMinSet}) + m) + n)$, where $t(\texttt{FindMinSet})$ is the running time of the function `FindMinSet`, $n = |X|$, and m is the weighted number of constraints in I.

Now, we concentrate on finding a min-set of I. Firstly, we deal with the case that we have an operation providing a min-intersection closure. Similarly as in the case of ll-closed constraints, we can define a notion of a minimal min-set: Let R be a k-ary constraint relation in I and $S \subseteq [k]$ be a set of entries such that there is a min-set A of R with $S \subseteq A$. Because R has intersection-closed min-sets, we have that there is also a min-set B of R such that B is a subset of every min-set containing S. We call such a min-set the *minimal min-set of R containing S*.

Now, we are ready to state our function for finding a min-set:

Algorithm 4.4.

```
FindMinSetIC(I) {
// Input: Instance I of CSP with variables X and constraints C whose
//   constraint relations have intersection-closed min-sets
// Output: Return min-set S ⊆ X of I.
//   If there is no such set S, return false.
  for all x ∈ X do begin
    S := {x}
    recheck := true
    correct := true
    while recheck ∧ correct do begin
      recheck := false
      for all C ∈ C do begin
        if there is no min-set of R(C) containing S ∩ V(C) then
          correct := false
        else begin
          S := S ∪ the minimal min-set of R(C) containing S ∩ V(C)
          if S changed then recheck := true
        end
      end
    end
    if correct then return S
  end
  return false
}
```

It is straightforward to check that the above algorithm runs in time $O(n^2m)$ where n is the number of variables of I and m is the weighted number of constraints in I.

Lemma 4.22. *If I has a solution, then the function* `FindMinSetIC` *in Algorithm 4.4 returns a set S of variables such that for all constraints $C \in \mathcal{C}$ such that $S \cap V(C) \neq \emptyset$ it holds that $S \cap V(C)$ is a min-set of $R(C)$.*

Proof. If I has a solution, there is some set S' of variables that have the minimal value in this solution. Consider a run of the while loop in the function `FindMinSetIC` for some variable $x \in S'$. In the beginning, it holds that $S = \{x\} \subseteq S'$. Because for each $C \in \mathcal{C}$, we have that $S' \cap V(C)$ is a min-set of $R(C)$ if $S' \cap V(C)$ is non-empty, *correct* cannot be set to `false` while $S \subseteq S'$. Because we always add to S only variables of the **minimal** min-set of $R(C)$ containing $S \cap V(C)$, all these variables are always in S' and therefore S remains a subset of S' all the time. At some moment, the while loop terminates. Because *correct* is `true`, we return a set S. When the while loop terminated, *recheck* was `false` and so for all $C \in \mathcal{C}$, the set S did not change. That means that for all $C \in \mathcal{C}$, the minimal min-set of $R(C)$ containing $S \cap V(C)$ was a subset of S, i. e., it was equal to $S \cap V(C)$. □

Theorem 4.23. *There is an algorithm running in time $O(n^3m)$ solving* CSP(Γ) *for Γ closed under* pp *and a polymorphism providing a min-intersection closure.*

Proof. Lemma 4.21 and Lemma 4.22 directly imply correctness of the Algorithm 4.3 using function `FindMinSetIC` from Algorithm 4.4. □

Secondly, we consider languages preserved under operation f providing min-union closure. This case has some similarities to the case of operation providing min-intersection closure. Let R be a k-ary relation closed under f and $S \subseteq [k]$. Let $A_1, \ldots, A_l$ be all min-sets of R such that $A_i \subseteq S$ for all $i \in [l]$. Because R has union-closed min-sets, there is some $A_j, j \in [l]$ such that $A_i \subseteq A_j$ for all $i \in [l]$. We call such min-set the *maximal min-set of R contained in S*. Note that this can be an empty set for some S.

Now, we are ready to state our function for finding a min-set:

Algorithm 4.5.

```
FindMinSetUC(I) {
// Input: Instance I of a CSP with variables X and constraints C whose
//   constraint relations have union-closed min-sets
// Output: Return min-set S ⊆ X of I.
//   If there is no such set S, return false.
  S := X
  recheck := true
  while recheck do begin
    recheck := false
    for all C ∈ C do begin
      if S ∩ V(C) ≠ ∅ then begin
        S := (S \ V(C)) ∪ the maximal min-set of R(C)
                 contained in S ∩ V(C)
        if S changed then recheck := true
      end
```

```
    end
  end
  if S ≠ ∅ then
    return S
  else
    return false
}
```

As in the case of intersection-closed min-sets, it is straightforward to check that the function `FindMinSetUC` has a running time $O(mn)$, where n is the number of variables of I and m is the weighted number of constraints in I. Now, we prove correctness of the function `FindMinSetUC`:

Lemma 4.24. *If I has a solution, then the function* `FindMinSetUC` *in Algorithm 4.5 returns a set S of variables such that for all constraints $C \in \mathcal{C}$ such that $S \cap V(C) \neq \emptyset$, it holds that $S \cap V(C)$ is a min-set of $R(C)$.*

Proof. If I has a solution, there is some set S' of variables that have the minimal value in this solution. In the beginning of the function, S is set to X and therefore $S \supseteq S'$. We show that $S \supseteq S'$ during the whole run of the function. Let $C \in \mathcal{C}$ be some constraint of I. Because $S' \cap V(C)$ is a min-set of $R(C)$ contained in S, the maximal min-set added to $S \setminus V(C)$ certainly contains $S' \cap V(C)$. Therefore after this modification to S, it still holds that $S \supseteq S'$. At some moment the function terminates and because $\emptyset \neq S' \subseteq S$, we return the set S. At the moment the while-loop was terminated, *recheck* has been set to `false`. Therefore for all $C \in \mathcal{C}$ such that $S \cap V(C) \neq \emptyset$, it holds that $S \cap V(C)$ is equal to (the maximal) min-set contained in S. □

Theorem 4.25. *There is an algorithm running in time $O(n^2 m)$ solving* $\mathrm{CSP}(\Gamma)$ *for Γ closed under* pp *and a polymorphism providing a min-union closure.*

Proof. Lemma 4.21 and Lemma 4.24 directly imply correctness of the Algorithm 4.3 using the function `FindMinSetUC` from Algorithm 4.5. □

Finally, we consider languages preserved under operation f providing a min-xor closure. Let R be a k-ary relation preserved by f. For a tuple $\overline{t} \in R$, we define $\chi_{\min}(\overline{t})$ to be a vector from $\{0,1\}^k$ such that $\chi_{\min}(\overline{t})[i] = 1$ if and only if $\overline{t}[i]$ is minimal in $\overline{t}$. We define $\chi_{\min}(R)$ as $\{\chi_{\min}(\overline{t}) \mid \overline{t} \in R\}$. As R is closed under f, the set $\chi_{\min}(R)$ is closed under addition of distinct vectors over $GF(2)$ and so $\chi_{\min}(R) \cup \{\{0\}^k\}$ is also closed under the operation $g(x,y,z) := x + y + z$ over $GF(2)$. Such operation is called *affine* (usually, affine operations are defined as $x + y - z$ but for $GF(2)$ this is equivalent) and a set of vectors closed under the affine operation is exactly the set of solutions of a system of linear equations [28]. Moreover this system can be constructed in polynomial time. So if we have an instance $I = (V, \mathbb{Q}, \mathcal{C})$ of a temporal CSP such that the constraint relation of

every $C \in \mathcal{C}$ is closed under f, we implement function `FindMinSetX` as follows: It constructs a system S of linear equations over $GF(2)$ with the set of variables $x_v, v \in V$, and equations as described above for each $C \in \mathcal{C}$. Then it finds a solution of S distinct from $\{0\}^k$. If S has such solution, then the variables getting value 1 in this solution form a min-set of I, which function `FindMinSetX` returns. If S has no such solution, then I has no min-set and function returns `false`. We can summarize the results of this paragraph in the following theorem:

Theorem 4.26. *There is a polynomial algorithm solving* $\mathrm{CSP}(\Gamma)$ *for* Γ *closed under* pp *and a polymorphism providing a min-xor closure.*

4.6 A Classification of Complexity of pp-closed TCSPs

In this section, we give a complete classification of the complexity of temporal constraint satisfaction problems that have the operation pp as a polymorphism. In particular, we show that this class of CSPs has a dichotomy — i. e., each problem is either tractable or NP-complete.

Theorem 4.27. *Let* Γ *be a temporal constraint language that is able to express the relation* $S^{\min}(x, y, z)$ *from Definition 4.4. Then* $\mathrm{CSP}(\Gamma)$ *is NP-complete.*

Proof. By Lemma 2.4, it is enough to show that $\mathrm{CSP}(S^{\min})$ is NP-complete. We reduce 1-in-3-SAT [45] to the problem $\mathrm{CSP}(S^{\min})$. Let ϕ be an input formula for 1-in-3-SAT with variables $x_1, \ldots, x_n$. We create the following instance I of $\mathrm{CSP}(S^{\min})$:

- our instance will have a distinguished variable a.
- for each variable x_i of ϕ, we introduce two variables v_i^1, v_i^2 and a constraint $S^{\min}(a, v_i^1, v_i^2)$
- for each clause C with variables x_i, x_j, x_k, we add a new variable v_C and introduce the following constraints: $S^{\min}(v_C, v_i^1, v_j^1)$, $S^{\min}(a, v_C, v_k^1)$

We show that I has a solution if and only if ϕ was 1-in-3 satisfiable. For the implication from right to left suppose we have some 1-in-3 satisfying truth assignment s. We assign a value 0 and for each variable x_i of ϕ, we assign 0 to v_i^1 and 1 to v_i^2 if x_i is true in s and $i+1$ to v_i^1 and 0 to v_i^2 if x_i is false in s. Clearly, the constraints $S^{\min}(a, v_i^1, v_i^2)$ are satisfied for all $i \in [n]$. For each clause C containing variables x_i, x_j, x_k, exactly one of variables is true in s. If it is x_k, we assign $\min(i+1, j+1)$ to v_C and see that both constraints $S^{\min}(v_C, v_i^1, v_j^1)$ and $S^{\min}(a, v_C, v_k^1)$ are satisfied (as v_k^1 is assigned 0). If it is x_i or x_j, we assign 0 to v_C and again see that both constraints are satisfied.

For the implication from left to right, let $\overline{s}$ be a solution to I. Without loss of generality we can assume that a has value 0 (otherwise we can apply appropriate automorphism of $(\mathbb{Q}, <)$ to $\overline{s}$). Because of the constraints $S^{\min}(a, v_i^1, v_i^2)$, exactly one of v_i^1, v_i^2 is 0 and the other variable is greater than 0 for all $i \in [n]$. We set variable x_i to true if v_i^1 is 0. Otherwise we set x_i to false. Now, we have to check that there is exactly one variable set to true in each clause. Let C be some clause of ϕ with variables x_i, x_j, x_k. First, we check that at most one variable is true in each clause. If x_k is true, it means that v_k^1 is 0 and so v_C must be greater than 0. Consequently both v_i^1 and v_j^1 must be greater than 0 and so x_i and x_j are false. If x_i or x_j is true, then it is so because v_i^1 or v_j^1 are 0, respectively. From the properties of $S^{\min}$ it follows that at most one of these variables is 0 (and hence at most one of x_i, x_j is true) and v_C must be 0 too. Hence, v_k^1 must be greater than 0 and so x_k is false. What remains to check is that at least one variable is set to true in each clause. If all variables x_i, x_j, x_k are false, it means that v_i^1, v_j^1, v_k^1 are all greater than zero. Therefore v_C is greater than zero and the constraint $S^{\min}(a, v_C, v_k^1)$ cannot be satisfied; a contradiction. So there is exactly one true variable in each clause and so the truth assignment we have defined is 1-in-3 satisfying. □

Now, we show that if the relation $S^{\min}$ cannot be expressed in Γ (i. e., by Theorem 2.9, there is some polymorphism of Γ that does not preserve $S^{\min}$), then there is a polymorphism of Γ providing special closure properties. First, we show two auxiliary lemmas:

Lemma 4.28. *Let f be a binary polymorphism preserving $<$ such that there is an infinite sequence $x_1 < x_2 < \ldots$ and y such that $f(x_1, y) > f(x_2, y) < f(x_i, y)$ for all $i > 2$. Then f generates an operation providing a min-intersection closure.*

Proof. Because f preserves $<$, we have that for arbitrary infinite sequence $y_1 < y_2 < \ldots$, $y_1 > y$, it holds that $f(x_2, y_i) > f(x_1, y)$. Hence operation $f(\alpha(x), \beta(y))$, where α is an automorphism of $(\mathbb{Q}, <)$ mapping $0, 1, \ldots$ to $x_2, x_3, \ldots$ and β is an automorphism of $(\mathbb{Q}, <)$ mapping $0, 1, 2, \ldots$ to $y, y_1, y_2, \ldots$, provides min-intersection closure. □

Lemma 4.29. *Let f be a binary polymorphism preserving $<$ such that there are infinite sequences $x_1 < x_2 < \ldots$ and $y_1 < y_2 < \ldots$ with the property that for all $j > 1$ there is some $i_0 > 1$ such that for all $i \geq i_0$, it holds that $f(x_i, y_1) < f(x_1, y_j)$. Furthermore, we require that $f(x_1, y_1) > f(x_i, y_1) > f(x_j, y_1)$ for all $1 < i < j$ or $f(x_1, y_1) > f(x_i, y_1) = f(x_j, y_1)$ for all $1 < i < j$ (cf. Figure 4.4). Then $\{f, \mathrm{pp}\}$ generates an operation providing a min-intersection closure.*

Proof. First, we show that there is a sequence of operations $f_1, f_2, \ldots$ generated by $\{f, \mathrm{pp}\}$ such that for each f_k it holds that $f_k(0, 0) < f_k(x, 0)$ and $f_k(0, 0) <$

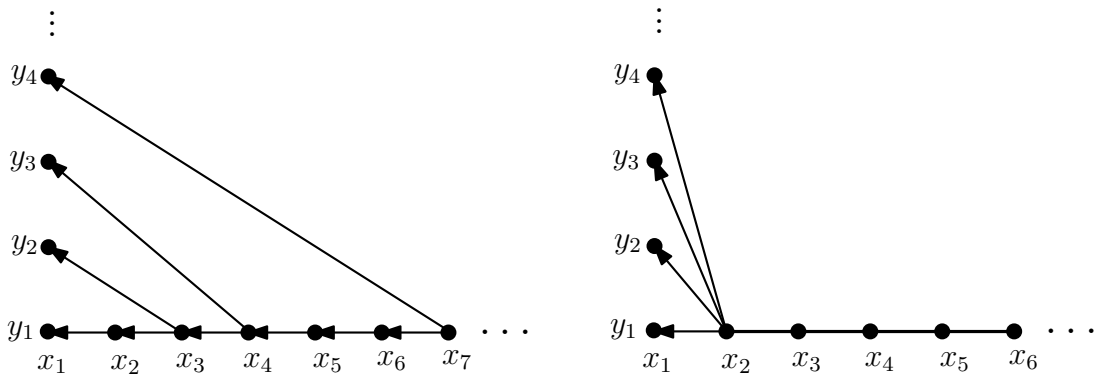


Figure 4.4: Two possibilities of the operation from Lemma 4.29.

$f_k(0,x)$ for all integers $x, 0 < x < k$. Then we use this sequence to generate an operation g providing a min-intersection closure.

So let $k > 0$ be fixed integer. By the assumptions of the lemma, there is $i_0 > 1$ such that for all $i \geq i_0$ it holds that $f(x_i, y_1) < f(x_1, y_k)$. So we apply an automorphism of $(\mathbb{Q}, <)$ to the first argument of f so that $0, 1, 2, \ldots$ is mapped to $x_1, x_{i_0}, x_{i_0+1}, \ldots$ and another automorphism of $(\mathbb{Q}, <)$ to the second argument of f so that $0, 1, \ldots$ is mapped to $y_1, y_2, \ldots$. Let f' denote the resulting operation. If $f(x_i, y_1) > f(x_j, y_1)$ for all $1 < i < j$, we choose an automorphism α of $(\mathbb{Q}, <)$ such that $f'(x, 0)$ is mapped to $k - x$ for all integers $x, 0 \leq x \leq k$. In case $f(x_i, y_1) = f(x_j, y_1)$ for all $1 < i < j$, we choose an automorphism α of $(\mathbb{Q}, <)$ so that $f'(0, 0)$ is mapped to k and $f'(1, 0)$ is mapped to 0. We set $f_k(x, y) := f'(\alpha(f'(x, y)), y)$ and we have to check that $f_k(0, 0) < f_k(x, 0)$ and $f_k(0, 0) < f_k(0, x)$ for all integers $x, 0 < x < k$. So let x be an integer from $\{1, \ldots, k-1\}$. From the choice of α, it holds that $f_k(0, 0) = f'(\alpha(f'(0, 0)), 0) = f'(k, 0)$ and

- $f_k(x, 0) = f'(\alpha(f'(x, 0)), 0) = f'(k - x, 0)$ if $f(x_i, y_1) > f(x_j, y_1)$ for all $1 < i < j$, or
- $f_k(x, 0) = f'(0, 0)$ if $f(x_i, y_1) = f(x_j, y_1)$ for all $1 < i < j$.

From the properties of f, we have that $f'(k, 0) < f'(k - x, 0)$ for all integers $x, 0 < x \leq k$ and so $f_k(0, 0) < f_k(x, 0)$ in the first case. In the second case, we have that $f'(k, 0) < f'(0, 0)$ and therefore $f_k(0, 0) < f_k(x, 0)$ for all integers $x, 0 < x < k$. Also, from the properties of f and the definition of f', it follows that $f'(0, x) > f'(1, 0)$ for all $x, 0 < x < k$. So we have $f_k(0, x) = f'(\alpha(f'(0, x)), x) > f'(0, 0) > f'(k, 0)$ for all $x, 0 < x < k$ from the properties of f and because f' preserves $<$ (observe that $\alpha(f'(0, x)) > \alpha(f'(1, 0)) \geq 0$). So we conclude that $f_k(0, 0) < f_k(0, x)$ for all $x, 0 < x < k$ too.

Now, we concentrate on generating g from $f_1, f_2, \ldots$. Fix some enumeration $x_1, x_2, \ldots$ of $\mathbb{Q}$. Clearly, for every k, there are only finitely many orderings of the set $\{(x_i, x_j) \mid i, j \in [k]\}$. Therefore there are (up to an automorphism of $(\mathbb{Q}, <)$) only finitely many operations on $\{x_1, \ldots, x_k\} \times \{x_1, \ldots, x_k\}$. Hence, for every k, there are infinitely many operations among $f_1, f_2, \ldots$ that are up to an automorphism of $(\mathbb{Q}, <)$ the same on $\{x_1, \ldots, x_k\} \times \{x_1, \ldots, x_k\}$. It immediately follows, that we can choose a subsequence $g_1, g_2, \ldots$ of $f_1, f_2, \ldots$ and automorphisms $\alpha_1, \alpha_2, \ldots$ of $(\mathbb{Q}, <)$ so that $\alpha_i(g_i)$ is the same as $\alpha_j(g_j)$ on $\{x_1, \ldots, x_i\} \times \{x_1, \ldots, x_i\}$ for all $j > i$. Now, we define $g(x, y)$ to be $\alpha_i(g_i(x, y))$ for arbitrary i such that $x, y \in \{x_1, \ldots, x_i\}$. It immediately follows from the definition of local generation that $\alpha_1(g_1), \alpha_2(g_2), \ldots$ generate g and therefore $f_1, f_2, \ldots$ generate g. By the choice of $f_1, f_2, \ldots$, it also follows that $g(0, 0) < g(0, x)$ and $g(0, 0) < g(x, 0)$ for all integers $x > 0$ and so g provides a min-intersection closure. □

Now, we are ready to prove the crucial lemma:

Lemma 4.30. *Let f be a binary polymorphism that preserves $<$ and violates the relation $S^{\min}$. Then $\{f, \mathrm{pp}\}$ generate an operation providing a min-intersection, a min-union, or a min-xor closure.*

Proof. Because f preserves $<$ and violates $S^{\min}$, we can assume without loss of generality (possibly after swapping arguments) that there are $x_1, x', y_1, y' \in \mathbb{Q}$ such that $x_1 < x'$, $y_1 < y'$ and $(f(x_1, y_1), f(x', y_1), f(x_1, y')) \notin S^{\min}$. We fix x_1, y_1 for the rest of the proof. Now, we distinguish several cases:

1. There are infinite sequences $x_2 < x_3 < \ldots$ and $y_2 < y_3 < \ldots$ such that $x_1 < x_2$, $y_1 < y_2$, $f(x_i, y_1) > f(x_1, y_1)$, and $f(x_1, y_i) > f(x_1, y_1)$ for all $i > 1$. In this case, an operation $f(\alpha(x), \beta(y))$, where α is an automorphism of $(\mathbb{Q}, <)$ mapping $0, 1, \ldots$ to $x_1, x_2, \ldots$ and β is an automorphism of $(\mathbb{Q}, <)$ mapping $0, 1, \ldots$ to $y_1, y_2, \ldots$, provides a min-intersection closure and we are done.

2. There are infinite sequences $x_2 < x_3 < \ldots$ and $y_2 < y_3 < \ldots$ such that $x_1 < x_2$, $y_1 < y_2$, and $f(x_i, y_1) = f(x_1, y_1) = f(x_1, y_i)$ for all $i > 1$. In this case, f after applying automorphisms of $(\mathbb{Q}, <)$ mapping $0, 1, \ldots$ to $x_1, x_2, \ldots$ and $0, 1, \ldots$ to $y_1, y_2, \ldots$ to its arguments provides min-union closure.

3. There is an infinite sequence $x_2 < x_3 < \ldots$ such that $x_1 < x_2$, $f(x_2, y_1) < f(x_1, y_1)$, and $f(x_i, y_1) > f(x_2, y_1)$ for all $i > 2$. In this case, we can use Lemma 4.28 and obtain that f generates an operation providing a min-intersection closure.

4. There is an infinite sequence $y_2 < y_3 < \ldots$ such that $y_1 < y_2$, $f(x_1, y_2) < f(x_1, y_1)$, and $f(x_1, y_i) > f(x_1, y_2)$ for all $i > 2$. In this case, we can just swap arguments and proceed as in case 3.

5. There is $x_2 > x_1$ such that $f(x_2, y_1) < f(x_1, y_1)$ and we are not in case 3 or 4. Then there must be infinite sequence $x_2 < x_3 < \ldots$ such that $x_1 < x_2$ and $f(x_i, y_1) \geq f(x_j, y_1)$ for all $2 \leq i < j$. Either this sequence has a subsequence $x'_2 < x'_3 < \ldots$ such that $f(x'_i, y_1) = f(x'_j, y_1)$ for all $2 \leq i < j$, or it has a subsequence $x'_2 < x'_3 < \ldots$ such that $f(x'_i, y_1) > f(x'_j, y_1)$ for all $2 \leq i < j$. Fix a subsequence $x'_2, x'_3, \ldots$ with one of these properties. Now, we distinguish three subcases:

   (a) There is an infinite sequence $y_2 < y_3 < \ldots$ such that $y_1 < y_2$ and for each $i > 1$ there is $j \geq 2$ satisfying $f(x_1, y_i) > f(x'_j, y_1)$. In this case, we can apply Lemma 4.29 and obtain that $\{f, \mathrm{pp}\}$ generates an operation providing a min-intersection closure.

   (b) There is an infinite sequence $y_2 < y_3 < \ldots$ such that $y_1 < y_2$ and $f(x_1, y_i) = f(x'_2, y_1)$ for all $i > 1$. In this case, we can assume that $f(x'_i, y_1) = f(x'_j, y_1)$ for all $2 \leq i < j$ as otherwise we can apply case 5a. Hence for chosen $x_1, x'_2, x'_3, \ldots$ and $y_1, y_2, \ldots$, it holds that $f(x_1, y_1) > f(x_1, y_i) = f(x'_i, y_1)$ for all $i > 1$ and therefore f after applying automorphisms of $(\mathbb{Q}, <)$ mapping $0, 1, 2, \ldots$ to $x_1, x'_2, x'_3, \ldots$ and $0, 1, \ldots$ to $y_1, y_2, \ldots$ to its arguments provides a min-xor closure.

   (c) The only remaining case is (recall that we are not in case 4 or 5a) that there is an infinite sequence $y_2 < y_3 < \ldots$ such that $y_1 < y_2$ and $f(x_1, y_i) \geq f(x_1, y_j)$ for all $2 \leq i < j$. If there is $i > 1$ such that for all $j \geq 2$ it holds that $f(x_1, y_i) < f(x'_j, y_1)$, we can just swap arguments and apply case 5a. Otherwise it holds that $f(x_1, y_i) > f(x_1, y_j)$ and $f(x'_i, y_1) > f(x'_j, y_1)$ for all $2 \leq i < j$. Furthermore, for every $j > 1$, there is $i_0 > 1$ such that for all $i \geq i_0$, we have $f(x'_i, y_1) < f(x_1, y_j)$ (otherwise the condition from the second sentence of the paragraph is satisfied). Hence, we can apply Lemma 4.29 and obtain that $\{f, \mathrm{pp}\}$ generates an operation providing a min-intersection closure.

6. There is $y_2 > y_1$ such that $f(x_1, y_2) < f(x_1, y_1)$ and we are not in case 3 or 4. In this case, we can just swap arguments and apply case 5.

Now, we deal with the remaining possibilities. So suppose that none of the previous cases applied. First, note that if there was $x_2 > x_1$ such that $f(x_2, y_1) < f(x_1, y_1)$, then one of the cases 3, 4, or 5 would apply. Similarly, if there was $y_2 > y_1$ such that $f(x_1, y_2) < f(x_1, y_1)$, then one of the cases 3, 4, or 6 would apply. Hence, in the remaining cases, we know that for all $x > x_1$ and $y > y_1$ it holds that $f(x, y_1) \geq f(x_1, y_1)$ and $f(x_1, y) \geq f(x_1, y_1)$. Because case 1 or case 2

did not apply, we can (possibly after swapping arguments) assume that there are no $x_2 < x_3 < \ldots$ such that $x_1 < x_2$ and $f(x_i, y_1) = f(x_1, y_1)$ for all $i > 1$ and there are no $y_2 < y_3 < \ldots$ such that $y_1 < y_2$ and $f(x_1, y_i) > f(x_1, y_1)$ for all $i > 1$. If there is at least $x_2 > x_1$ such that $f(x_2, y_1) = f(x_1, y_1)$, then because f preserves $<$, f provides a min-intersection closure after applying automorphisms of $(\mathbb{Q}, <)$ mapping $0, 1, 2, \ldots$ to $x_2, x_3, x_4, \ldots$ (where $x_3, x_4, \ldots$ are chosen so that $f(x_i, y_1) > f(x_2, y_1) = f(x_1, y_1)$ for all $i > 2$) and $0, 1, \ldots$ to $y_1, y_2, \ldots$ to the arguments of f. Similarly, if there is at least $y_2 > y_1$ such that $f(x_1, y_2) > f(x_1, y_1)$, then because f preserves $<$, the assumptions of Lemma 4.29 are satisfied (after swapping arguments) for $y_2 < y_3 < y_4 < \ldots$ (where $y_3, y_4, \ldots$ are chosen so that $f(x_1, y_i) = f(x_1, y_1)$ for all $i > 2$) and $x_1 < x_2 < \ldots$. Hence, the only remaining possibility is that for any $y > y_1$, it holds that $f(x_1, y) = f(x_1, y_1)$ and for any $x > x_1$, it holds that $f(x, y_1) > f(x_1, y_1)$. But then $(f(x_1, y_1), f(x, y_1), f(x_1, y)) \in S^{\min}$ for any $x > x_1, y > y_1$, which is a contradiction to the choice of x_1, y_1 in the beginning of the proof. □

Now, we have to deal with the case that Γ is not able to express the $<$ relation. An easy corollary of Lemma 2.12 is:

Corollary 4.31. *Let Γ be a temporal constraint language that cannot express $<$. Then Γ has a unary polymorphism that violates $<$.*

The standard result of Cameron [23], Theorem 3.10, characterizing highly transitive permutation groups on a countable set implies the following characterization of the reducts of $(\mathbb{Q}, <)$ (i. e., relational structures over $\mathbb{Q}$ that are closed under first-order definitions and whose all relations are first-order definable in the dense linear order $(\mathbb{Q}, <)$). An alternative proof of the characterization of reducts of $(\mathbb{Q}, <)$ is given in [59] and we use the formulation of the theorem from this paper.

Theorem 4.32. *Up to first-order interdefinability, there are exactly five reducts of $(\mathbb{Q}, <)$:*

- *The dense linear order $(\mathbb{Q}, <)$ itself,*
- *The structure $(\mathbb{Q}, Betweenness)$,*
- *The structure $(\mathbb{Q}, Cyclic)$, where Cyclic is a ternary relation defined as $Cyclic(x, y, z) := \{(x, y, z) \mid (x < y < z) \vee (y < z < x) \vee (z < x < y)\}$,*
- *The structure $(\mathbb{Q}, Sep)$, where Sep is a 4-ary relation $Sep(x_1, y_1, x_2, y_2) := \{(x_1, y_1, x_2, y_2) \mid (x_1 < x_2 < y_1 < y_2) \vee (x_1 < y_2 < y_1 < x_2) \vee (y_1 < x_2 < x_1 < y_2) \vee (y_1 < y_2 < x_1 < x_2) \vee (x_2 < x_1 < y_2 < y_1) \vee (x_2 < y_1 < y_2 < x_1) \vee (y_2 < x_1 < x_2 < y_1) \vee (y_2 < y_1 < x_2 < x_1)\}$,*
- *The structure $(\mathbb{Q}, \emptyset)$.*

Note that in the case of $(\mathbb{Q}, \emptyset)$, the relation $\neq$ has a primitive positive definition in the structure. The following result of Bodirsky and Nešetřil allows us to translate the result directly into the language of polymorphisms:

Theorem 4.33. *[11, 18] Let Γ be an ω-cathegorical relational structure. Then a relation R has a first-order definition in Γ if and only if R is preserved by all automorphisms of Γ.*

Proposition 4.34. *Let Γ be a temporal constraint language (recall that it can be viewed as a relational structure over $\mathbb{Q}$). Then one of the following happens:*

- *All unary polymorphisms of Γ preserve $<$, or*
- *there is a non-injective unary operation that is a polymorphism of Γ, or*
- *the clone generated by all unary polymorphisms of Γ is the clone generated by* neg*, or*
- *the clone generated by all unary polymorphisms of Γ is the clone generated by* cyc*, or*
- *the clone generated by all unary polymorphisms of Γ is the clone generated by* neg *and* cyc*, or*
- *all permutations are polymorphisms of Γ.*

Proof. If Γ has a non-injective unary polymorphism, we are done. Otherwise we know that every unary polymorphisms of Γ is an automorphism of Γ. Thus if there is a unary polymorphism f of Γ violating $<$, then, by Theorem 4.33, $<$ does not have a first-order definition in Γ. Thus one of the last four cases of Theorem 4.32 apply. All unary polymorphisms of Γ generate some locally closed clone that is the clone generated by unary polymorphisms of one of the structures $(\mathbb{Q}, \mathit{Betweenness})$, $(\mathbb{Q}, \mathit{Cyclic})$, $(\mathbb{Q}, \mathit{Sep})$, or $(\mathbb{Q}, \emptyset)$ by Theorem 4.32. In the first case, neg is a polymorphism of the structure $(\mathbb{Q}, \mathit{Betweenness})$ and cyc is not a polymorphism of the structure. Thus the clone generated by all unary polymorphisms of $(\mathbb{Q}, \mathit{Betweenness})$ is the clone generated by neg. Similarly, in the second case, we get that the clone generated by all unary polymorphisms of $(\mathbb{Q}, \mathit{Cyclic})$ is the clone generated by cyc. In the third case, both neg and cyc are polymorphisms of the structure $(\mathbb{Q}, \mathit{Sep})$ and the structure is not preserved by all permutations. So the clone generated by all unary polymorphisms of the structure is the clone generated by neg and cyc. In the fourth case, all the permutations of $\mathbb{Q}$ are polymorphisms of the structure and so we are done. □

We continue by showing that if $\mathrm{pp} \in \mathrm{Pol}(\Gamma)$ and either neg or cyc are also polymorphisms of Γ, then Γ also has a binary polymorphism f that preserves $<$ but violates $S^{\min}$. We can then plug this polymorphism into Lemma 4.30 and obtain the same result as if Γ was able to express $<$.

Lemma 4.35. *The operation* pp *together with* neg *or* cyc *generate a binary operation f that preserves $<$ and violates $S^{\min}$.*

Proof. First, we show that $\{\mathrm{neg}, \mathrm{pp}\}$ generates the binary operation f. We set $f(x,y) := \mathrm{neg}(\mathrm{pp}(\mathrm{neg}(x), \mathrm{neg}(y)))$. We verify that f preserves $<$. Let $x_1 < x_2$ and $y_1 < y_2$. Clearly, $\mathrm{neg}(x_1) > \mathrm{neg}(x_2)$ and $\mathrm{neg}(y_1) > \mathrm{neg}(y_2)$. As pp preserves $<$, it follows that $\mathrm{pp}(\mathrm{neg}(x_1), \mathrm{neg}(y_1)) > \mathrm{pp}(\mathrm{neg}(x_2), \mathrm{neg}(y_2))$. Therefore $f(x_1, y_1) < f(x_2, y_2)$ and we conclude that f preserves $<$. It is also easy to see that f violates $S^{\min}$. Consider tuples $\overline{t_1} := (-1, -1, 0)$ and $\overline{t_2} := (-1, 0, -1)$ and let $\overline{t_3} := f(\overline{t_1}, \overline{t_2})$. From the definition of f and pp it follows that $\overline{t_3}[1] < \overline{t_3}[2]$ and $\overline{t_3}[1] < \overline{t_3}[3]$. We conclude that $\overline{t_3}$ does not satisfy $S^{\min}$.

For the operation cyc, we define $f(x,y) := \mathrm{cyc}(\mathrm{pp}(\mathrm{cyc}(x), y))$. In this definition we pick operation pp defined as $\mathrm{pp}(x,y) = x$ if $x \leq 0$ and $\mathrm{pp}(x,y) = 2 - 1/(y+1)$ for $x > 0$. Again, we first verify that f preserves $<$. Let $x_1 < x_2$, $y_1 < y_2$ and $z_1 := \mathrm{pp}(\mathrm{cyc}(x_1), y_1)$, $z_2 := \mathrm{pp}(\mathrm{cyc}(x_2), y_2)$. If $x_1 > 0$ or $x_2 \leq 0$, it holds that $\mathrm{cyc}(x_1) < \mathrm{cyc}(x_2)$ and because pp preserves $<$, we have that $z_1 < z_2$. By our choice of pp, it also follows that either $z_1 > 0$ or $z_2 \leq 0$ and so $f(x_1, y_1) < f(x_2, y_2)$. So suppose that $x_1 \leq 0$ and $x_2 > 0$. The choice of pp implies that $z_1 > 0$ and $z_2 < 0$ and so $\mathrm{cyc}(z_1) < \mathrm{cyc}(z_2)$. Again $<$ is preserved. Finally, we have to check that f violates $S^{\min}$. Consider tuples $\overline{t_1} := (-1, -1, 1)$ and $\overline{t_2} := (-1, 1, -1)$ and let $\overline{t_3} := f(\overline{t_1}, \overline{t_2})$. By the definition of cyc the tuple $\mathrm{cyc}(\overline{t_1})$ is $(1/2, 1/2, -1/2)$. Therefore after application of pp we obtain a triple $(\mathrm{pp}(1/2, -1), \mathrm{pp}(1/2, 1), \mathrm{pp}(-1/2, -1))$. It is easy to check that these are three distinct values and as cyc is injective, $\overline{t_3}$ also contains three distinct values. We have just shown that f violates $S^{\min}$. □

Results in this section and Section 4.5 directly imply the following theorem:

Theorem 4.36. *Let Γ be a temporal constraint language having the polymorphism* pp*. If $S^{\min} \in \langle\Gamma\rangle$, $\mathrm{CSP}(\Gamma)$ is NP-complete. Otherwise $\mathrm{CSP}(\Gamma)$ is tractable.*

Proof. If Γ is able to express $S^{\min}$, then by Theorem 4.27 $\mathrm{CSP}(\Gamma)$ is NP-complete. Otherwise Lemma 2.12 implies that there is an at most binary polymorphism f of Γ that violates $S^{\min}$. If this polymorphism also violates $<$, Proposition 4.34 asserts that either neg or cyc are polymorphisms of Γ and so by Lemma 4.35 there is a binary polymorphism g of Γ that preserves $<$ and violates $S^{\min}$. Using this polymorphism (or directly polymorphism f if it preserves $<$), we obtain using Lemma 4.30 that there is a binary polymorphism h of Γ such that h provides a min-intersection closure, a min-union closure, or a min-xor closure. In each of these cases (as Γ also has the pp polymorphism), Theorem 4.23, Theorem 4.25, or Theorem 4.26 give us that $\mathrm{CSP}(\Gamma)$ is tractable. □

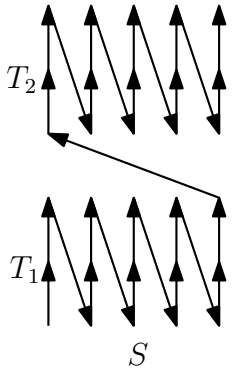


Figure 4.5: The operation from Lemma 4.37.

4.7 Operations Generating ll or dual-ll

This section contains a sequence of results gradually weakening assumptions needed for an operation to generate ll or dual-ll operation. We use these results in the next section to finish a classification of the complexity of a constraint satisfaction problem for temporal constraint languages.

To speak about properties of an operation on some restricted sets, the following definitions are useful: If $S_1, \ldots, S_d$ are sets, we call a set of the form $S_1 \times \cdots \times S_d$ a *grid*, and also write S^d for a product of the form $S \times \cdots \times S$ with d factors. A $[k]^d$*-subgrid* of a grid $S_1 \times \cdots \times S_d$ is a subset of $S_1 \times \cdots \times S_d$ of the form $S'_1 \times \cdots \times S'_d$, where S'_i is a k-element subset of S_i. We say that a *k-ary operation f behaves as a k-ary operation g on $S \subset \mathbb{Q}^k$* if the (non-strict) ordering induced by f on the tuples in S is the same as the ordering induced on these tuples by g.

We start by several technical lemmas about generation of ll.

Lemma 4.37. *Let f be a binary operation such that there are $c, d \in \mathbb{Q}$ and sets $S^{(i)}, T_1^{(i)}, T_2^{(i)} \subseteq \mathbb{Q}$ for each $i > 0$ such that $|S^{(i)}| = |T_1^{(i)}| = |T_2^{(i)}| = i$, $t_1 < d < t_2$ for all $t_1 \in T_1^{(i)}$, $t_2 \in T_2^{(i)}$ and f behaves as* lex *on $S^{(i)} \times T_1^{(i)}$ and on $S^{(i)} \times T_2^{(i)}$ and that $f(s, t_1) < c < f(s, t_2)$ for all $s \in S^{(i)}, t_1 \in T_1^{(i)}, t_2 \in T_2^{(i)}$ (see Figure 4.5 for an illustration of the operation f). Then f generates* ll.

Proof. We show that for all positive integers k, f generates a binary operation g_k such that there are sets S', T'_1, T'_2, $|S'| \geq k$, $|T'_1| \geq k$, $|T'_2| \geq k$ satisfying:

- $g_k(x, y)$ behaves as $\mathrm{lex}(x, y)$ on $S' \times T'_2$,
- $g_k(x, y)$ behaves as $\mathrm{lex}(y, x)$ on $S' \times T'_1$, and
- $g_k(s, t_1) < c < g_k(s, t_2)$ for all $s \in S'$, $t_1 \in T'_1$, $t_2 \in T'_2$.

This already implies that f generates ll.

So fix k a positive integer, choose $l := (3k)^k$, and pick arbitrary $T_1' \subset T_1^{(l)}$, $T_2'' \subset T_2^{(l)}$, and $S' \subset S^{(l)}$ such that $T_1' = \{b_1, \ldots, b_k\}$, $b_1 < b_2 < \ldots < b_k$, $T_2'' = \{c_1, \ldots, c_{2k}\}$, $c_1 < c_2 < \ldots < c_{2k}$, and S' is formed by the k smallest elements of $S^{(l)}$. We also define $T_2' := \{c_1, \ldots, c_k\}$. There is an automorphism β of $(\mathbb{Q}, <)$ that maps the set $\{f(s,t) \mid s \in S'', t \in T_1' \cup T_2''\}$ to $S^{(l)}$, where S'' contains the $l/3k$ smallest elements of $S^{(l)}$. Let $\alpha_1, \ldots, \alpha_k$ be automorphisms of $(\mathbb{Q}, <)$ such that α_i maps $b_1, \ldots, b_i$ to $b_1, \ldots, b_i$, and $\{b_{i+1}, \ldots, b_k\} \cup T_2'$ to T_2''. We define

$$g_k(x,y) := f(\beta(f(\beta(f(\ldots\beta(f(\beta(f(x,\alpha_1(y))),\alpha_2(y)))\ldots)),\alpha_{k-1}(y))),\alpha_k(y))$$

Operation g_k behaves as lex on $S' \times T_2'$ as f behaves as lex on $S^{(l)} \times T_2^{(l)}$, automorphisms $\alpha_1, \ldots, \alpha_k$ map T_2' into T_2'', and so the automorphism β always maps its argument back into $S^{(l)}$ on $S' \times T_2'$. Also the outermost application of f in the definition of $g_k(x,y)$ implies that $g_k(s,t_1) < c < g_k(s,t_2)$ for all $s \in S'$, $t_1 \in T_1'$, $t_2 \in T_2'$. It remains to check that $g_k(x,y)$ behaves as $\mathrm{lex}(y,x)$ on $S' \times T_1'$. We show that for arbitrary $u_1, u_2 \in S'$, $v_1, v_2 \in T_1'$ such that $v_1 \leq v_2$, $u_1 < u_2$ if $v_1 = v_2$, it holds that $g_k(u_1,v_1) < g_k(u_2,v_2)$. If $u_1 \leq u_2$ and $v_1 \leq v_2$, then because f preserves $\leq$ on $S^{(l)} \times (T_1' \cup T_2'')$, we immediately obtain that $g_k(u_1,v_1) \leq g_k(u_2,v_2)$ and by injectivity of f on $S^{(l)} \times (T_1' \cup T_2'')$, that $g_k(u_1,v_1) < g_k(u_2,v_2)$. The only remaining case is $u_1 > u_2$ and $v_1 < v_2$. Let $i \in [k]$ be such that $v_1 = b_i$. By the choice of α_i, we know that α_i maps v_1 to b_i and v_2 to T_2''. Therefore $f(s,\alpha_i(v_1)) < c < f(s',\alpha_i(v_2))$ for all $s, s' \in S^{(l)}$. If $i = 1$, we are done as this is also the result of g_k. So assume $i > 1$. The function f on the level $i-1$ from outside evaluates $f(s,\alpha_{i-1}(v_1))$ and $f(s',\alpha_{i-1}(v_2))$, respectively, where $s, s' \in S'', s < s'$. As f preserves $<$ on $S \times (T_1' \cup T_2'')$, we get that $f(s,\alpha_{i-1}(v_1)) < f(s',\alpha_{i-1}(v_2))$. We can now argue in the same way for the function on the level $i-2$ from outside. The ordering of the first argument is thus transferred up to the first level and we conclude that $g_k(u_1,v_1) < g_k(u_2,v_2)$. □

In the following lemmas, we call operations $g(x,y) = \mathrm{lex}(x, \mathrm{neg}(y))$ and $g(x,y) = \mathrm{lex}(y, \mathrm{neg}(x))$ *weakly negated* lex. If for any $x, x' \in \mathbb{Q}$, $x \neq x'$, it holds that $g(x,y) < g(x',y')$ for all $y, y' \in \mathbb{Q}$, or $g(x,y) > g(x',y')$ for all $y, y' \in \mathbb{Q}$, we say that the *first argument of g is its major argument* (an example of such operation is lex). Analogously, we define that the *second argument of g is its major argument.*

Lemma 4.38. *Let f be a binary operation preserving $<$ such that there are $c, d \in \mathbb{Q}$ and sets $S_1^{(i)}, S_2^{(i)}, T^{(i)} \subset \mathbb{Q}$ for each $i > 0$ such that $|S_1^{(i)}| = |S_2^{(i)}| = |T^{(i)}| = i$, $s_1 < d < s_2$ for all $s_1 \in S_1^{(i)}$, $s_2 \in S_2^{(i)}$, $f(s_1,t) < c < f(s_2,t)$ for all $s_1 \in S_1^{(i)}$, $s_2 \in S_2^{(i)}$, and $t \in T^{(i)}$, and f behaves as:*

- *(weakly negated)* lex *with major the first argument on* $S_1^{(i)} \times T^{(i)}$ *and as (weakly negated)* lex *with major the second argument on* $S_2^{(i)} \times T^{(i)}$*, or*
- *as (weakly negated)* lex *with major the second argument on* $S_1^{(i)} \times T^{(i)}$ *and as (weakly negated)* lex *with major the first argument on* $S_2^{(i)} \times T^{(i)}$*.*

Then f *generates* ll *or* dual-ll*.*

Proof. We assume that f behaves as (weakly negated) lex having the first argument major on $S_1^{(i)} \times T^{(i)}$ and as (weakly negated) lex having the second argument major on $S_2^{(i)} \times T^{(i)}$ for all $i > 0$. The other case is symmetric. Similarly as in the previous lemma, we show that for all positive integers k, f generates an operation g_k such that there are sets S_1', S_2', T', $|S_1'| \geq k$, $|S_2'| \geq k$, $|T'| \geq k$, such that

- $g_k(x, y)$ behaves as $\mathrm{lex}(x, y)$ on $S_1' \times T'$,
- $g_k(x, y)$ behaves as $\mathrm{lex}(y, x)$ on $S_2' \times T'$, and
- $g_k(s_1, t) < c < g_k(s_2, t)$ for all $s_1 \in S_1'$, $s_2 \in S_2'$, $t \in T'$.

It then immediately follows that f generates ll. So fix k a positive integer, choose $l := 64k^6$, and let S_1' be formed by the k smallest elements of $S_1^{(l)}$, S_2' be formed by the k smallest elements of $S_2^{(l)}$, and T' be formed by the k smallest elements of $T^{(l)}$. We also define S_1'' to be the $l/2k$ smallest elements of $S_1^{(l)}$, S_2'' to be the $l/2k$ smallest elements of $S_2^{(l)}$, and T'' to be the $l/2k$ smallest elements of $T^{(l)}$. It follows that there is an automorphism α of $(\mathbb{Q}, <)$ mapping the set $\{f(s,t) \mid s \in S_1'', t \in T'\}$ to $S_1^{(l)}$ and the set $\{f(s,t) \mid s \in S_2'', t \in T'\}$ to $S_2^{(l)}$. Also there is an automorphism β of $(\mathbb{Q}, <)$ mapping the set $\{f(s,t) \mid s \in S_1' \cup S_2', t \in T''\}$ to $T^{(l)}$.

Having these automorphisms we define

$$g_k(x, y) := f(\alpha(f(x, \beta(f(x, y)))), \beta(f(\alpha(f(x, y)), y)))$$

We show that g_k has the desired properties. Operation g_k preserves $<$ as f preserves $<$. Hence, it is enough to verify the correct ordering of $g_k(u_1, v_1)$ and $g_k(u_2, v_2)$ for $u_1, u_2 \in S_1' \cup S_2'$, $u_1 \leq u_2$ and $v_1, v_2 \in T'$, $v_1 \geq v_2$. We distinguish four cases:

- Firstly, suppose $u_1 \in S_1'$ and $u_1 < u_2$. We know that $f(u_1', v_1') < f(u_2', v_2')$ for all $u_1' \in S_1^{(l)}$, $u_2' \in S_1^{(l)} \cup S_2^{(l)}$, $u_1' < u_2'$, $v_1', v_2' \in T^{(l)}$ — either $u_2' \in S_1^{(l)}$ and then the inequality follows from the fact that f has major the first argument in $S_1^{(l)} \times T^{(l)}$, or $u_2' \in S_2^{(l)}$ and then we have the inequality because $f(u_1', v_1') < c < f(u_2', v_2')$. Because also $\alpha(f(u_1, v_1')) \in S_1^{(l)}$ and $\alpha(f(u_2, v_2')) \in S_1^{(l)} \cup S_2^{(l)}$ for all $v_1', v_2' \in T''$, it is straightforward to verify that there are $u_1' \in S_1^{(l)}$, $u_2' \in S_1^{(l)} \cup S_2^{(l)}$, $u_1' < u_2'$ and $v_1', v_2' \in T^{(l)}$

such that $u_1' = \alpha(f(u_1, \beta(f(u_1, v_1))))$, $u_2' = \alpha(f(u_2, \beta(f(u_2, v_2))))$, $v_1' = \beta(f(\alpha(f(u_1, v_1)), v_1))$, and $v_2' = \beta(f(\alpha(f(u_2, v_2)), v_2))$. Therefore we have that $g_k(u_1, v_1) = f(u_1', v_1') < f(u_2', v_2') = g_k(u_2, v_2)$ and we are done.

- Secondly, suppose $u_1 \in S_1'$ and $u_1 = u_2$. We have that $v_1 > v_2$, as if $v_1 = v_2$, we have nothing to prove. If f behaves as lex on $S_1^{(l)} \times T^{(l)}$, we get that $f(u_1, \beta(f(u_1, v_1))) > f(u_2, \beta(f(u_2, v_2)))$ as f also preserves $\leq$ in $S_1^{(l)} \times T^{(l)}$. If f behaves as weakly negated lex on $S_1^{(l)} \times T^{(l)}$, we have that $f(u_1, v_1) < f(u_2, v_2)$, and thus $f(u_1, \beta(f(u_1, v_1))) > f(u_2, \beta(f(u_2, v_2)))$. As f has major the first argument in $S_1^{(l)} \times T^{(l)}$, we conclude that $g_k(u_1, v_1) > g_k(u_2, v_2)$, which we wanted to prove.

- Thirdly, suppose $u_1 \in S_2'$ and $v_1 > v_2$. It follows that $u_2 \in S_2'$. Therefore $\alpha(f(u_1, v_1')) \in S_2^{(l)}$ for all $v_1' \in T''$ and $\alpha(f(u_2, v_2')) \in S_2^{(l)}$ for all $v_2' \in T''$. Set $u_1' := \alpha(f(u_1, \beta(f(u_1, v_1))))$, $u_2' := \alpha(f(u_2, \beta(f(u_2, v_2))))$. Because f has major the second argument in $S_2^{(l)} \times T^{(l)}$, we have that $f(u_1'', v_1) > f(u_2'', v_2)$ for all $u_1'', u_2'' \in S_2^{(l)}$. Thus if we set $v_1' := \beta(f(\alpha(f(u_1, v_1)), v_1))$, $v_2' := \beta(f(\alpha(f(u_2, v_2)), v_2))$, we know that $v_1' > v_2'$. Giving these facts together, we get that $g_k(u_1, v_1) = f(u_1', v_1') > f(u_2', v_2') = g_k(u_2, v_2)$.

- Finally, suppose $u_1 \in S_2'$ and $v_1 = v_2$. We see that $u_2 \in S_2'$ and $u_2 > u_1$ as otherwise there is nothing to prove. Similarly as in the previous case, we see that $u_1' := \alpha(f(u_1, \beta(f(u_1, v_1))))$, $u_2' := \alpha(f(u_2, \beta(f(u_2, v_2))))$, are both in $S_2^{(l)}$. Let $v_1' := \beta(f(\alpha(f(u_1, v_1)), v_1))$ and $v_2' := \beta(f(\alpha(f(u_2, v_2)), v_2))$. If f behaves as lex on $S_2^{(l)} \times T^{(l)}$, we see that $v_1' < v_2'$ and thus $g_k(u_1, v_1) = f(u_1', v_1') < f(u_2', v_2') = g_k(u_2, v_2)$. If f behaves as weakly negated lex on $S_2^{(l)} \times T^{(l)}$, we have that $f(u_1, v_1) > f(u_2, v_2)$. By the choice of α we know that $\alpha(f(u_1, v_1)), \alpha(f(u_2, v_2)) \in T_2^{(l)}$ and consequently $v_1' < v_2'$. We again conclude that $g_k(u_1, v_1) = f(u_1', v_1') < f(u_2', v_2') = g_k(u_2, v_2)$.

□

Lemma 4.39. *Let f be a binary operation preserving $<$ such that there are $c, d \in \mathbb{Q}$ and sets $S^{(i)}, T_1^{(i)}, T_2^{(i)} \subset \mathbb{Q}$ for each $i > 0$ such that $|S^{(i)}| = |T_1^{(i)}| = |T_2^{(i)}| = i$, $t_1 < d < t_2$ for all $t_1 \in T_1^{(i)}$, $t_2 \in T_2^{(i)}$, and f behaves as (weakly negated)* lex *on $S^{(i)} \times T_1^{(i)}$ and on $S^{(i)} \times T_2^{(i)}$, it has major the first argument in these two subgrids and $f(s, t_1) < c < f(s, t_2)$ for all $s \in S^{(i)}$, $t_1 \in T_1^{(i)}$, and $t_2 \in T_2^{(i)}$. Then f generates* ll.

Proof. Similarly as in the previous lemmas, we show that for all positive integers k, f generates an operation g_k such that there are sets S', T_1', T_2', $|S'| \geq k$, $|T_1'| \geq k$, $|T_2'| \geq k$, such that

- g_k behaves as lex on $S' \times T_1'$,

- g_k behaves as lex on $S' \times T'_2$, and
- $g_k(s,t_1) < c < g_k(s,t_2)$ for all $s \in S'$, $t_1 \in T'_1$, $t_2 \in T'_2$.

Such operations $g_1, g_2, \ldots$ generate an operation g which behaves as lex on $S \times T_1$ and on $S \times T_2$ for $S = \mathbb{Q}$, $T_1 = \{y \in \mathbb{Q} \mid y < 0\}$, $T_2 = \{y \in \mathbb{Q} \mid y > 0\}$ and which satisfies $g(s,t_1) < c < g(s,t_2)$ for all $s \in S$, $t_1 \in T_1$, $t_2 \in T_2$. Such operation satisfies conditions of Lemma 4.37 and thus we get that f generates ll. So fix k a positive integer, choose $l := k^2$, and let S' be formed by the k smallest elements of $S^{(l)}$, T'_1 be formed by the k smallest elements of $T_1^{(l)}$, and T'_2 be formed by the k smallest elements of $T_2^{(l)}$. It follows that there is an automorphism α of $(\mathbb{Q}, <)$ mapping the set $\{f(s,t) \mid s \in S', t \in T'_1\}$ to T_1 and the set $\{f(s,t) \mid s \in S', t \in T'_2\}$ to T_2.

Having this automorphism we define $g_k(x,y) := f(x, \alpha(f(x,y)))$. Clearly, g_k preserves $<$. It also holds that $g_k(s,t_1) < c < g_k(s,t_2)$ for all $s \in S'$, $t_1 \in T'_1$, and $t_2 \in T'_2$, because α mapped $f(x,y)$ to T_1 if $y \in T'_1$ and to T_2 if $y \in T'_2$ for all $x \in S'$. It remains to show that g_k behaves on both grids $S' \times T'_1$ and $S' \times T'_2$ as lex. If f behaves as lex on some grid, then so does g_k because of the property of α described above. So assume that f behaves as weakly negated lex on at least one of the grids. Let it be $S' \times T'_1$. For $S' \times T'_2$, the argument is the same. We show that for arbitrary $u_1, u_2 \in S'$, $v_1, v_2 \in T'_1$ such that $u_1 \leq u_2$, $v_1 < v_2$ if $u_1 = u_2$, it holds that $g_k(u_1,v_1) < g_k(u_2,v_2)$. If $u_1 < u_2$, this immediately follows due to the outer application of f, which preserves $<$ and has major the first argument in $S \times T_1$. So assume that $u_1 = u_2$ and $v_1 < v_2$. But then $f(u_1,v_1) > f(u_2,v_2)$, and consequently $f(u_1, \alpha(f(u_1,v_1))) < f(u_2, \alpha(f(u_2,v_2)))$. □

Now, we are ready for proving the main result of this section. Its proof uses so called *product Ramsey theorem (PRT)* [49], which can be easily derived from the classical infinite Ramsey theorem; see [82] for a general introduction to Ramsey theory.

Theorem 4.40. *[49] For all positive integers d, r, m, and k, there is a positive integer $\mathcal{R}(d,r,m,k)$ such that for all sets $S_1, \ldots, S_d$, $|S_i| \geq \mathcal{R}(d,r,m,k)$ for all $i \in [d]$, and an arbitrary coloring of $[m]^d$ subgrids of $S_1 \times \ldots \times S_d$ with r colors, there exists a $[k]^d$ subgrid of $S_1 \times \ldots \times S_d$ such that all $[m]^d$ subgrids of the $[k]^d$ subgrid have the same color.*

As we use the above theorem for $d = m = 2$ and r is also obvious from the context, we use just $\mathcal{R}(k)$ instead of $\mathcal{R}(d,r,m,k)$ in our notation. We apply the theorem to prove the following:

Lemma 4.41. *Let f be a binary injective operation preserving $<$ and violating Betweenness. Then f generates* ll *or* dual-ll.

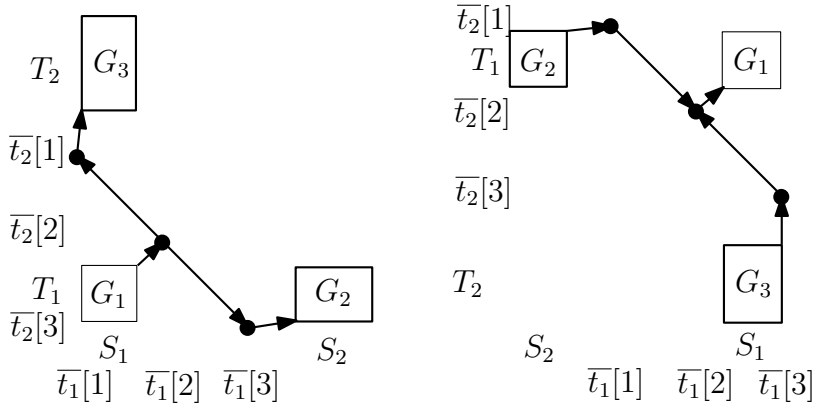


Figure 4.6: Grids chosen for application of product Ramsey theorem. Depicted ordering on values of f follows from the choice of $\overline{t_1}, \overline{t_2}$ and because f preserves $<$.

Proof. We identify three subgrids of $\mathbb{Q} \times \mathbb{Q}$ and show using Theorem 4.40 on these grids that f generates ll or dual-ll.

As f is injective, there are $\overline{t_1}, \overline{t_2} \in$ *Betweenness* such that $\overline{t} := f(\overline{t_1}, \overline{t_2}) \notin$ *Betweenness* and $\overline{t}$ is injective. As f preserves $<$, we can without loss of generality assume that $\overline{t_1}[1] < \overline{t_1}[2] < \overline{t_1}[3]$ and $\overline{t_2}[1] > \overline{t_2}[2] > \overline{t_2}[3]$ (otherwise, we can consider f with swapped arguments).

There are two possibilities for triple $\overline{t}$. Either $\overline{t}[1] > \overline{t}[2] < \overline{t}[3]$ or $\overline{t}[1] < \overline{t}[2] > \overline{t}[3]$. Depending on this condition, we choose grids we apply product Ramsey theorem to. In the first case, let $S_1 := \{x \in \mathbb{Q} \mid \overline{t_1}[1] < x < \overline{t_1}[2]\}$, $S_2 := \{x \in \mathbb{Q} \mid \overline{t_1}[3] < x\}$, $T_1 := \{y \in \mathbb{Q} \mid \overline{t_2}[3] < y < \overline{t_2}[2]\}$, and $T_2 := \{y \in \mathbb{Q} \mid \overline{t_2}[1] < y\}$. Similarly, in the second case let $S_1 := \{x \in \mathbb{Q} \mid \overline{t_1}[2] < x < \overline{t_1}[3]\}$, $S_2 := \{x \in \mathbb{Q} \mid x < \overline{t_1}[1]\}$, $T_1 := \{y \in \mathbb{Q} \mid \overline{t_2}[2] < y < \overline{t_2}[1]\}$, and $T_2 := \{y \in \mathbb{Q} \mid y < \overline{t_2}[3]\}$. See Figure 4.6 for an illustration of chosen grids.

Now, we apply the product Ramsey theorem (Theorem 4.40) to grid $S_1 \times T_1$. We use the theorem for $d = m = 2$ and arbitrarily large k. We color the $[2] \times [2]$ subgrids of the grid according to the strict linear order (recall that f is injective) appearing on the $[2] \times [2]$ subgrid. As our sets S_1, T_1 are infinite, we can obtain subsets $U^{(k)} \subseteq S_1$ and $V^{(k)} \subseteq T_1$ of arbitrary size from the theorem, in particular such that $|U^{(k)}| \geq \mathcal{R}(k)$, $|V^{(k)}| \geq \mathcal{R}(k)$ and all the $[2] \times [2]$ subgrids of $U^{(k)} \times V^{(k)}$ are colored by the same color. Next, we apply the product Ramsey theorem the to grid $U^{(k)} \times T_2$ with the same coloring as above and obtain subsets $S_1^{(k)} \subseteq U^{(k)}$ and $T_2^{(k)} \subseteq T_2$. Finally, we apply the theorem to $S_2 \times V^{(k)}$ and obtain subsets $S_2^{(k)} \subseteq S_2$ and $T_1^{(k)} \subseteq V^{(k)}$. Both grids $S_1^{(k)} \times T_2^{(k)}$ and $S_2^{(k)} \times T_1^{(k)}$ have the same strict linear ordering induced on all their $[2] \times [2]$ subgrids, $|S_1^{(k)}| \geq k$, $|S_2^{(k)}| \geq k$, $|T_1^{(k)}| \geq k$, and $|T_2^{(k)}| \geq k$.

The only injective orderings of a $[2] \times [2]$ subgrid preserving $<$ that can be

Figure 4.7: Four injective orderings of $[2] \times [2]$ grid preserving $<$ that can be present on all $[2] \times [2]$ subgrids of a large grid.

present on all $[2] \times [2]$ subgrids of a large grid are depicted in Figure 4.7. It is straightforward to check that the resulting ordering in the grid corresponds to one of the operations $\mathrm{lex}(x, y)$, $\mathrm{lex}(y, x)$, $\mathrm{lex}(x, \mathrm{neg}(y))$, and $\mathrm{lex}(y, \mathrm{neg}(x))$. Therefore f behaves as one of these operations on the grids $S_1^{(k)} \times T_1^{(k)}$, $S_1^{(k)} \times T_2^{(k)}$, and $S_2^{(k)} \times T_1^{(k)}$. If f behaves as (weakly negated) lex with major the first argument on one of the grids and as (weakly negated) lex with major the second argument on another grid for infinitely many k's, then clearly f satisfies conditions of Lemma 4.38 (possibly after swapping arguments) and we see that f generates ll or dual-ll. Otherwise the (weakly negated) lex has the same major argument in all three grids for infinitely many k's and it satisfies conditions of Lemma 4.39 (possibly after swapping arguments). We again conclude that f generates ll or dual-ll. □

4.8 A Complete Classification of Complexity of TCSPs

In this section, we connect results from Section 4.6 and Section 4.7 and prove that every temporal constraint satisfaction problem is either tractable or NP-complete. We start with three general lemmas about polymorphisms of temporal constraint languages:

Lemma 4.42. *Let f be k-ary polymorphism such that there are $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ such that $x_i < y_i$ for all $i \in [k]$ and $f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)$. Then f generates a constant operation.*

Proof. The proof directly follows from Lemma 2.13. The group of automorphisms of $(\mathbb{Q}, <)$ has just one orbit of 2-sets and so it is enough to show that f generates a non-injective endomorphism and apply Lemma 2.13. Because $x_i < y_i$ for all $i \in [k]$, there are automorphisms $\alpha_2, \ldots, \alpha_k$ such that the automorphism α_i maps x_1 to x_i and y_1 to y_i for all $i \in \{2, \ldots, k\}$. The operation $g(x) = f(x, \alpha_2(x), \ldots, \alpha_k(x))$ is then a non-injective endomorphism as $g(x_1) = f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k) = g(y_1)$. □

Lemma 4.43. *Let f be a binary operation not generating* cyc *and not preserving $<$. Then either f generates a constant operation or* $\mathrm{neg}(f(x, y))$ *preserves $<$.*

Proof. Consider a temporal constraint language $\Gamma := \mathrm{Inv}(f)$. As f does not preserve $<$ and $<$ has only one orbit of pairs in the automorphism group of $(\mathbb{Q}, <)$, Lemma 2.12 asserts that there is a unary polymorphism g of Γ violating $<$. Either it is a non-injective endomorphism and Lemma 4.42 asserts that Γ has a constant operation or Proposition 4.34 implies that g generates neg. As Theorem 2.9 implies that the clone generated by f is equal to $\mathrm{Pol}(\Gamma)$, we know that either f generates a constant operation or f generates neg. In the first case, we are done. So assume next that we are in the second case.

We show that $\mathrm{neg}(f(x, y))$ preserves $<$. Because f does not generate a non-injective unary polymorphism, we have that $f(x_1, y_1) \neq f(x_2, y_2)$ for all $x_1, x_2, y_1, y_2 \in \mathbb{Q}$, $x_1 < x_2$, $y_1 < y_2$. Because f also does not generate a unary polymorphism violating *Betweenness* (all unary injective polymorphisms violating *Betweenness* generate cyc by Proposition 4.34), it satisfies for every $x_1, y_1, x_2, y_2, x_3, y_3 \in \mathbb{Q}$, $x_1 < x_2 < x_3$, $y_1 < y_2 < y_3$, that $f(x_1, y_1) < f(x_2, y_2) < f(x_3, y_3)$ or $f(x_1, y_1) > f(x_2, y_2) > f(x_3, y_3)$. As f does not preserve $<$, there are $a_1, b_1, a_2, b_2 \in \mathbb{Q}$, $a_1 < a_2$, $b_1 < b_2$ such that $f(a_1, b_1) > f(a_2, b_2)$. Consider arbitrary $x_3, y_3 \in \mathbb{Q}$ such that $x_3 > a_2$, $y_3 > b_2$. Because f does not generate cyc, it holds that $f(x_3, y_3) < f(a_2, b_2)$. By the same argument applied to a_2, b_2, x_3, y_3, we get that $f(x_3, y_3) > f(x_4, y_4)$ for all $x_4 > x_3$ and $y_4 > y_3$. Now, consider arbitrary $x_1, y_1, x_2, y_2 \in \mathbb{Q}$, $x_1 < x_2$, $y_1 < y_2$. We show that $f(x_1, y_1) > f(x_2, y_2)$. Clearly, there are x_3, y_3, x_4, y_4 such that $\max(x_2, a_2) < x_3 < x_4$, $\max(y_2, b_2) < y_3 < y_4$. Because $f(x_3, y_3) > f(x_4, y_4)$ and f does not generate cyc, we have that $f(x_1, y_1) > f(x_3, y_3)$ and $f(x_2, y_2) > f(x_3, y_3)$. Thus we can even infer that $f(x_1, y_1) > f(x_2, y_2)$ as otherwise f would generate cyc. We conclude that operation $\mathrm{neg}(f(x, y))$ preserves $<$. □

Lemma 4.44. *Let f be a binary operation generating* cyc. *Then either f generates a constant operation, or f generates all permutations of $\mathbb{Q}$ or there is an automorphism α of $(\mathbb{Q}, <)$ such that* $\mathrm{cyc}(\alpha(f(x, y)))$ *or* $\mathrm{cyc}(\alpha(\mathrm{neg}(f(x, y))))$ *preserves $<$.*

Proof. If there are $a_1, b_1, a_2, b_2 \in \mathbb{Q}$, $a_1 < a_2$, $b_1 < b_2$, such that $f(a_1, b_1) = f(a_2, b_2)$, then by Lemma 4.42 f generates a constant operation. So we further assume that $f(a_1, b_1) \neq f(a_2, b_2)$ for all $a_1, b_1, a_2, b_2 \in \mathbb{Q}$, $a_1 < a_2$, $b_1 < b_2$. Proposition 4.34 asserts, that any unary polymorphism violating *Sep*, generates all permutations of $\mathbb{Q}$. Therefore, we also further assume that any unary operation generated by f preserves *Sep*.

As f generates cyc, there are $a_1, b_1, a_2, b_2, a_3, b_3 \in \mathbb{Q}$, $a_1 < a_2 < a_3$, $b_1 < b_2 < b_3$, such that $f(a_1, b_1) < f(a_2, b_2) > f(a_3, b_3)$ or $f(a_1, b_1) > f(a_2, b_2) < f(a_3, b_3)$. Without loss of generality we assume that the first case is true as otherwise we can consider $\mathrm{neg}(f(x, y))$ instead. First, assume that $f(a_3, b_3) > f(a_1, b_1)$ and consider arbitrary $x_1, y_1 \in \mathbb{Q}$ such that $x_1 < a_1$, $y_1 < b_1$. It cannot be that $f(x_1, y_1) < f(a_1, b_1)$ as then $f(x, \alpha(x))$, where α is an automorphism of $(\mathbb{Q}, <)$

mapping x_1 to y_1, a_1 to b_1, a_2 to b_2, and a_3 to b_3, would be a unary operation violating *Sep*. For the same reason, it also cannot be that $f(x_1, y_1) > f(a_3, b_3)$. We see that $x_1, y_1, a_1, b_1, a_2, b_2$ are such that $f(a_2, b_2) > f(x_1, y_1) > f(a_1, b_1)$. We can therefore without loss of generality assume that we have chosen $a_1, b_1, a_2, b_2, a_3, b_3$ so that $f(a_3, b_3) < f(a_1, b_1) < f(a_2, b_2)$ as otherwise we can consider $\mathrm{neg}(f(x, y))$ instead of f and choose $x_1, y_1, a_1, b_1, a_2, b_2$ for $a_1, b_1, a_2, b_2, a_3, b_3$. Note that $\mathrm{neg}(\mathrm{neg}(f(x, y))) = f(x, y)$ and so we are still considering either $f(x, y)$ or $\mathrm{neg}(f(x, y))$.

Let $x_1, y_1 \in \mathbb{Q}$ be such that $x_1 < a_1$, $y_1 < b_1$. Because f does not generate a unary operation violating *Sep*, it is straightforward to check that $f(a_3, b_3) < f(x_1, y_1) < f(a_1, b_1)$. Similarly, we obtain that for any $x_2, y_2 \in \mathbb{Q}$ such that $x_2 > a_3$, $y_2 > b_3$ it holds that $f(x_1, y_1) > f(x_2, y_2) > f(a_3, b_3)$. To finish our observations about f, consider $x_2, y_2, x_2', y_2' \in \mathbb{Q}$, $a_3 < x_2 < x_2'$, $b_3 < y_2 < y_2'$. By taking $a_1, b_1, a_2, b_2, x_2, y_2$ instead of $a_1, b_1, a_2, b_2, a_3, b_3$, we obtain by the above arguments that $f(x_2, y_2) < f(x_2', y_2')$. Similarly for $x_1, y_1, x_1', y_1' \in \mathbb{Q}$, $x_1 < x_1' < a_1$, $y_1 < y_1' < b_1$, we obtain that $f(x_1, y_1) < f(x_1', y_1')$.

Set $d := \inf\{f(x, y) \mid x, y \in \mathbb{Q}, x < a_1, y < b_1\}$. As all such values $f(x, y)$ are bounded below by any $f(x', y')$ for $x', y' \in \mathbb{Q}$, $x' > a_3$, $y' > b_3$, we have that also $d > f(x', y')$. Let α be an automorphism of $(\mathbb{Q}, <)$ such that it maps d to 0 and set $g(x, y) := \mathrm{cyc}(\alpha(f(x, y)))$. We show that g preserves $<$.

Let $u_1, v_1, u_2, v_2 \in \mathbb{Q}$ be such that $u_1 < u_2$, $v_1 < v_2$. We have to verify that $\mathrm{cyc}(\alpha(f(u_1, v_1))) < \mathrm{cyc}(\alpha(f(u_2, v_2)))$. First, assume that $f(u_1, v_1) < f(u_2, v_2)$. If $f(u_1, v_1) > d$, we are done as $\alpha(f(u_1, v_1)) > 0$ and cyc thus preserves the ordering. Otherwise consider arbitrary $x_1, y_1, x_2, y_2 \in \mathbb{Q}$, $x_1 < x_2 < \min(u_1, a_1)$, $y_1 < y_2 < \min(v_1, b_1)$. As $f(x_1, y_1) > d$, we see that $f(u_1, v_1) < f(x_1, y_1) < f(x_2, y_2)$. But then, because f does not generate a unary polymorphism violating *Sep*, we have that $f(u_2, v_2) < f(x_1, y_1)$. As f preserves $<$ in the set $\{x \mid x < a_1\} \times \{y \mid y < b_1\}$ and we can choose x_1, y_1 arbitrarily if they are small enough, we conclude that $f(u_2, v_2) \leq d$ and therefore $\mathrm{cyc}(\alpha(f(u_1, v_1))) < \mathrm{cyc}(\alpha(f(u_2, v_2)))$.

Now, assume that $f(u_1, v_1) > f(u_2, v_2)$. We show that $f(u_1, v_1) > d$ and $f(u_2, v_2) < d$, which directly implies that $\mathrm{cyc}(\alpha(f(u_1, v_1))) < \mathrm{cyc}(\alpha(f(u_2, v_2)))$. If $f(u_1, v_1) < f(x_1, y_1)$ for all $x_1, y_1 \in \mathbb{Q}$, $x_1 < \min(u_1, a_1)$, $y_1 < \min(v_1, b_1)$, we can choose $x_1, y_1, x_1', y_1' \in \mathbb{Q}$, so that $x_1 < x_1' < u_1$, $y_1 < y_1' < v_1$, and $f(u_2, v_2) < f(u_1, v_1) < f(x_1, y_1) < f(x_1', y_1')$. A contradiction with the fact that f does not generate a unary polymorphism violating *Sep*. Similarly, if $f(u_2, v_2) > f(x_3, y_3)$ for all $x_3, y_3 \in \mathbb{Q}$, $x_3 > \max(u_2, a_3)$, $y_3 > \max(v_2, b_3)$, we can choose $x_3, y_3, x_3', y_3' \in \mathbb{Q}$, so that $u_2 < x_3 < x_3'$, $v_2 < y_3 < y_3'$, and $f(x_3, y_3) < f(x_3', y_3') < f(u_2, v_2) < f(u_1, v_1)$. Again, a contradiction with the fact that f does not generate a unary polymorphism violating *Sep*. So there are $x_1, y_1, x_3, y_3 \in \mathbb{Q}$ such that $x_1 < \min(u_1, a_1)$, $y_1 < \min(v_1, b_1)$, $x_3 > \max(u_2, a_3)$, $y_3 > \max(v_2, b_3)$, and $f(x_1, y_1) < f(u_1, v_1)$ and $f(x_3, y_3) > f(u_2, v_2)$. We see that $f(u_1, v_1) > d$ and $f(u_2, v_2) < d$ (as $f(x_3, y_3) < f(x_2, y_2)$ for all $x_2, y_2 \in \mathbb{Q}$, $x_2 < a_1$, $y_2 < b_1$). □

Putting the above two lemmas together yields the following result fundamental for our further progress:

Lemma 4.45. *Let f be a binary operation. Then f satisfies at least one of the following:*

- *Either f generates a constant operation, or*
- *f generates all permutations of $\mathbb{Q}$, or*
- *f preserves $<$, or*
- $\mathrm{neg}(f(x,y))$ *preserves $<$, or*
- *there is an automorphism α of $(\mathbb{Q},<)$ such that* $\mathrm{cyc}(\alpha(f(x,y)))$ *preserves $<$, or*
- *there is an automorphism α of $(\mathbb{Q},<)$ such that* $\mathrm{cyc}(\alpha(\mathrm{neg}(f(x,y))))$ *preserves $<$.*

Proof. Suppose f does not preserve $<$. If f does not generate cyc, we use Lemma 4.43 and obtain that either f generates a constant operation, or operation $\mathrm{neg}(f(x,y))$ preserves $<$. If f does generate cyc, we apply Lemma 4.44 and obtain that, indeed, we are in one of the cases described in the statement of the lemma. □

Intuitively, this lemma tells us that if we have a binary operation f generating a non-increasing automorphism, then either we are in one of the simple cases or f also generates a binary operation preserving $<$ (and thus not generating any non-increasing automorphism), which has otherwise similar properties as f. We use this result in the following characterization of temporal constraint languages:

Lemma 4.46. *Let Γ be a temporal constraint language. Then Γ satisfies at least one of the following conditions:*

- *Either* $\mathrm{Pol}(\Gamma)$ *contains a constant operation, or*
- $\mathrm{CSP}(\Gamma)$ *is NP-complete, or*
- $\mathrm{cyc} \in \mathrm{Pol}(\Gamma)$ *and* $\mathrm{neg} \in \mathrm{Pol}(\Gamma)$*, or*
- *there is binary $f \in \mathrm{Pol}(\Gamma)$ such that f violates Betweenness and preserves $<$.*

Proof. If *Betweenness* $\in \langle\Gamma\rangle$, then $\mathrm{CSP}(\Gamma)$ is clearly NP-complete as the problem $\mathrm{CSP}(\textit{Betweenness})$ is well-known to be NP-complete [45]. So suppose that *Betweenness* $\notin \langle\Gamma\rangle$. Lemma 2.12 asserts that there is an at most binary operation $f \in \mathrm{Pol}(\Gamma)$ that violates *Betweenness*. If f preserves $<$, we are done (note that in

this case f has to be binary). Otherwise $<$ has not primitive positive definition in Γ and by Proposition 4.34 either neg, or cyc, or a non-injective unary operation are polymorphisms of Γ. In the third case, Lemma 4.42 asserts that a constant operation is a polymorphism of Γ and we are done.

If cyc is a polymorphism of Γ, then consider relation $\mathit{Cyclic}(x, y, z) := (x < y < z) \vee (y < z < x) \vee (z < x < y)$. If this relation has a primitive positive definition in Γ, then CSP(Γ) is NP-complete as CSP(*Cyclic*) is known to be NP-complete [45]. So assume in the following that *Cyclic* does not have a primitive positive definition in Γ and by Lemma 2.12 there is a unary polymorphism of Γ (note that *Cyclic* has just one orbit of triples in Aut(Γ) when cyc $\in$ Pol(Γ)) violating *Cyclic*. Proposition 4.34 then asserts that neg $\in$ Pol(Γ) or a non-injective unary operation is a polymorphism of Γ. In the first case, we are done because $\{\mathrm{cyc}, \mathrm{neg}\} \subseteq \mathrm{Pol}(\Gamma)$. In the second case, we are done by Lemma 4.42.

The remaining case is that cyc $\notin$ Pol(Γ), Γ does not have a non-injective endomorphism and has a binary polymorphism f generating neg (not generating cyc) and not preserving *Betweenness* and $<$. We apply Lemma 4.43 to f and obtain that $\mathrm{neg}(f(x, y))$ preserves $<$. Clearly, $\mathrm{neg}(f(x, y))$ is a polymorphism of Γ and it still violates *Betweenness*, because $(x, y, z) \notin \mathit{Betweenness}$ if and only if $(\mathrm{neg}(x), \mathrm{neg}(y), \mathrm{neg}(z)) \notin \mathit{Betweenness}$. So $\mathrm{neg}(f(x, y))$ is the operation required in the statement of the lemma. □

For a while, we postpone our classification of temporal constraint languages and show that the constraint satisfaction problem for relation *Sep* defined in Theorem 4.32 is NP-complete.

Theorem 4.47. *The problem* CSP(*Sep*) *is NP-complete.*

Proof. We prove NP-hardness of CSP(*Sep*) by a reduction from the problem CSP(*Betweenness*), whose NP-completeness has been established in [45]. So assume we are given an instance $I = (X, \mathbb{Q}, \mathcal{C})$ where each constraint in $\mathcal{C}$ has a constraint relation *Betweenness*. The instance I' of CSP(*Sep*) is created as follows. The set of variables is $X \cup \{a\}$, where a is a new variable. For each constraint $C \in \mathcal{C}$ on variables x_1, x_2, x_3 we add a constraint $\mathit{Sep}(a, x_2, x_1, x_3)$ to $\mathcal{C}'$. It is obvious, that I' is in CSP(*Sep*) and that the transformation can be performed in polynomial time. We only have to check that I' has a solution if and only if I has a solution. If I has a solution $\overline{t}$, then for any constraint $C \in \mathcal{C}$ on variables x_1, x_2, x_3, their values are either $\overline{t}[x_1] < \overline{t}[x_2] < \overline{t}[x_3]$ or $\overline{t}[x_3] < \overline{t}[x_2] < \overline{t}[x_1]$. Therefore, if we assign to a a value smaller than all the values of $\overline{t}$, all constraints in $\mathcal{C}'$ are satisfied and we have a solution of I'. For the other implication, suppose I' has a solution. Because *Sep* is preserved by cyc, it also has a solution $\overline{t}$ in which a gets the minimal value among all variables of I'. Now, consider a constraint $C' \in \mathcal{C}'$ on variables a, x_2, x_1, x_3. As a has the minimal value in $\overline{t}$, it holds that either $\overline{t}[x_1] < \overline{t}[x_2] < \overline{t}[x_3]$ or $\overline{t}[x_3] < \overline{t}[x_2] < \overline{t}[x_1]$

and so the corresponding constraint $C \in \mathcal{C}$ is satisfied. We see that $\overline{t}$ restricted to X is a solution of I. □

Now, we are ready to further extend our classification:

Lemma 4.48. *Let* Γ *be temporal constraint language such that* $\{\mathrm{neg}, \mathrm{cyc}\} \subseteq \mathrm{Pol}(\Gamma)$. *Then* Γ *satisfies at least one of the following:*

- *Either* $\mathrm{Pol}(\Gamma)$ *contains a constant operation, or*
- $\mathrm{CSP}(\Gamma)$ *is NP-complete, or*
- $\mathrm{Aut}(\Gamma)$ *contains all permutations of* $\mathbb{Q}$*, or*
- *there is binary* $f \in \mathrm{Pol}(\Gamma)$ *such that* f *violates Sep and preserves* $<$.

Proof. If $Sep \in \langle\Gamma\rangle$, then Theorem 4.47 asserts that $\mathrm{CSP}(\Gamma)$ is NP-complete. So we can further assume that $Sep \notin \langle\Gamma\rangle$. Because *Sep* has just two orbits of 4-tuples in $\mathrm{Aut}(\Gamma)$ when $\mathrm{Pol}(\Gamma)$ contains cyc, Lemma 2.12 asserts that there is an at most binary polymorphism f violating *Sep*. If f is unary or generates a unary polymorphism violating *Sep*, Proposition 4.34 gives us that either Γ has a non-injective endomorphism and therefore by Lemma 4.42 a constant operation, or all permutations of $\mathbb{Q}$ are automorphisms of Γ. So we are done in this case.

We further assume that f is binary and does not generate a unary operation violating *Sep*. By Lemma 4.45, either $f(x, y)$, or $\mathrm{neg}(f(x, y))$, or $\mathrm{cyc}(\alpha(f(x, y)))$, or $\mathrm{cyc}(\alpha(\mathrm{neg}(f(x, y))))$ preserve $<$ for some polymorphism α of $(\mathbb{Q}, <)$. Let g denote such operation. In all these cases, it is straightforward to verify that $(u, v, x, y) \in Sep$ if and only if $(g(u), g(v), g(x), g(y)) \in Sep$ and we conclude that $g(x, y)$ still violates *Sep*. □

Results obtained in this section so far can be summarized (and slightly extended) in the following theorem:

Theorem 4.49. *Let* Γ *be a temporal constraint language. Then it satisfies at least one of the following:*

- $\mathrm{CSP}(\Gamma)$ *is NP-complete, or*
- $\mathrm{Pol}(\Gamma)$ *contains a constant operation, or*
- $\mathrm{Aut}(\Gamma)$ *contains all permutations of* $\mathbb{Q}$*, or*
- *there is binary* $f \in \mathrm{Pol}(\Gamma)$ *preserving* $<$ *and violating Betweenness.*

Proof. Lemma 4.46 gives us that either Γ has a constant polymorphism, or $\mathrm{CSP}(\Gamma)$ is NP-complete, or $\{\mathrm{cyc}, \mathrm{neg}\} \subseteq \mathrm{Pol}(\Gamma)$, or there is binary $f \in \mathrm{Pol}(\Gamma)$ such that f violates *Betweenness* and preserves $<$. In the first two cases and in the fourth case, we are immediately done. In the third case, Lemma 4.48 applied to Γ asserts that either we are in one of the first two cases, or $\mathrm{Aut}(\Gamma)$ contains all permutations of $\mathbb{Q}$, or there is binary $f \in \mathrm{Pol}(\Gamma)$ such that f violates *Sep* and preserves $<$. So it is enough to show that f also violates *Betweenness*.

If f does not preserve $\neq$ (i. e., there are $a_1, b_1, a_2, b_2 \in \mathbb{Q}$, $a_1 < a_2$, $b_1 > b_2$, such that $f(a_1, b_1) = f(a_2, b_2)$), it immediately follows that it does not preserve *Betweenness*. So assume in the following that f preserves $\neq$.

If there are $a_1, b_1, a_2, b_2, a_3, b_3 \in \mathbb{Q}$ such that $a_1 < a_2 < a_3$, $b_1 > b_2 > b_3$, and $f(a_1, b_1) < f(a_2, b_2) > f(a_3, b_3)$ or $f(a_1, b_1) > f(a_2, b_2) < f(a_3, b_3)$, then f clearly violates *Betweenness* and we are done. Assume for contradiction that such $a_1, b_1, a_2, b_2, a_3, b_3$ do not exist — i. e., for all such $a_1, b_1, a_2, b_2, a_3, b_3$, it holds that either $f(a_1, b_1) < f(a_2, b_2) < f(a_3, b_3)$ or $f(a_1, b_1) > f(a_2, b_2) > f(a_3, b_3)$. As f violates *Sep*, there are $\overline{t_1}, \overline{t_2} \in$ *Sep* such that $f(\overline{t_1}, \overline{t_2}) \notin$ *Sep*. We can without loss of generality assume that $\overline{t_1}[1] < \overline{t_1}[3] < \overline{t_1}[2] < \overline{t_1}[4]$ as otherwise we can permute entries of the tuples. All the possible combinations of two tuples are depicted in Figure 4.8. Tuples $\overline{t_1}$, $\overline{t_2}$ cannot be in combinations depicted in the first line as those configurations result in a tuple from *Sep*(because of the assumption from the beginning of the paragraph). For combinations in the second line, we show there are $u_1, u_2, u_1', u_2', v_1, v_2, v_1', v_2' \in \mathbb{Q}$, $u_1 < u_1'$, $v_1 > v_1'$, $u_2 < u_2'$, $v_2 > v_2'$, such that $f(u_1, v_1) < f(u_1', v_1')$ if and only if $f(u_2, v_2) > f(u_2', v_2')$. In the first combination in the second line, $f(\overline{t_1}[1], \overline{t_2}[1]) < f(\overline{t_1}[3], \overline{t_2}[3])$ if and only if $f(\overline{t_1}[4], \overline{t_2}[4]) > f(\overline{t_1}[2], \overline{t_2}[2])$, provided f violates *Sep*. Therefore we can choose $(u_1, v_1) := (\overline{t_1}[1], \overline{t_2}[1])$, $(u_1', v_1') := (\overline{t_1}[3], \overline{t_2}[3])$, $(u_2, v_2) := (\overline{t_1}[2], \overline{t_2}[2])$, $(u_2', v_2') := (\overline{t_1}[4], \overline{t_2}[4])$. In the second combination in the second line, we can choose $(u_1, v_1) := (\overline{t_1}[1], \overline{t_2}[1])$, $(u_1', v_1') := (\overline{t_1}[2], \overline{t_2}[2])$ and depending on the ordering of $f(u_1, v_1)$ and $f(u_1', v_1')$, either setting $(u_2, v_2) := (\overline{t_1}[3], \overline{t_2}[3])$, $(u_2', v_2') := (\overline{t_1}[2], \overline{t_2}[2])$ or setting $(u_2, v_2) := (\overline{t_1}[1], \overline{t_2}[1])$, $(u_2', v_2') := (\overline{t_1}[4], \overline{t_2}[4])$ gives us pairs with the desired properties. In the third and the fourth combination in the second line, the choice of $u_1, v_1, u_1', v_1', u_2, v_2, u_2', v_2'$ is obvious. Moreover, we can choose the values so that $(u_1, v_1) \neq (u_2', v_2')$ and $(u_1', v_1') \neq (u_2, v_2)$ because of the assumption on f from the beginning of the paragraph.

Having chosen $u_1, v_1, u_1', v_1', u_2, v_2, u_2', v_2'$ consider $u_3', v_3' \in \mathbb{Q}$ such that $u_3' > \max(u_1', u_2')$, $v_3' < \min(v_1', v_2')$. Because of the assumption on f from the beginning of the previous paragraph, we have that either $f(u_1, v_1) > f(u_1', v_1') > f(u_3', v_3') > f(u_2', v_2') > f(u_2, v_2)$, or $f(u_1, v_1) < f(u_1', v_1') < f(u_3', v_3') < f(u_2', v_2') < f(u_2, v_2)$. But then for $u_3, v_3 \in \mathbb{Q}$, $u_3 > u_3'$, $v_3 < v_3'$, there is no possible value of $f(u_3, v_3)$ — in all cases, we obtain a configuration violating *Betweenness* either on (u_1', v_1'), (u_3', v_3'), (u_3, v_3) or on (u_2', v_2'), (u_3', v_3'), (u_3, v_3). We conclude that f violates *Betweenness*.

□

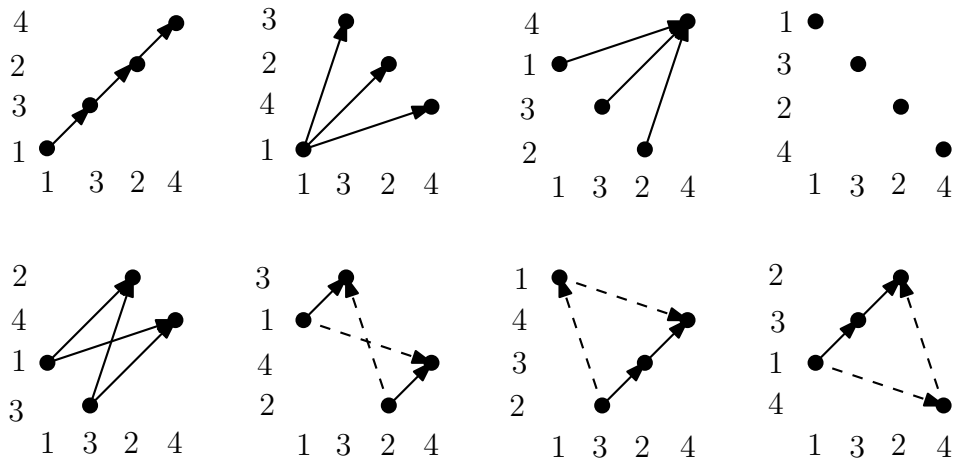


Figure 4.8: Possible combinations of two 4-tuples $\overline{t_1}$, $\overline{t_2}$ from *Sep*. Solid lines denote ordering resulting from f preserving $<$, dashed lines denote ordering required for $f(\overline{t_1}, \overline{t_2}) \notin Sep$. Numbers are indices of corresponding entries in tuples $\overline{t_1}$, $\overline{t_2}$.

The first three cases in the previous theorem are easy to deal with. Therefore, we concentrate on the fourth case. First, we prove two technical lemmas:

Lemma 4.50. *Let f be a binary operation such that there are $c, d \in \mathbb{Q}$ and sets $S^{(i)}, T_1^{(i)}, T_2^{(i)} \subseteq \mathbb{Q}$ for each $i > 0$ such that $|S^{(i)}| = |T_1^{(i)}| = |T_2^{(i)}| = i$, $t_1 < d < t_2$ for all $t_1 \in T_1^{(i)}$, $t_2 \in T_2^{(i)}$, f behaves as a projection to the first argument on $S^{(i)} \times T_1^{(i)}$ and on $S^{(i)} \times T_2^{(i)}$ and $f(s, t_1) < c < f(s, t_2)$ for all $s \in S^{(i)}, t_1 \in T_1^{(i)}, t_2 \in T_2^{(i)}$ (see Figure 4.9 for an illustration of the operation f). Then f generates* lex.

Proof. We show that for all positive integers k, f generates a binary operation g_k such that there are sets S', T_1', $|S'| \geq k$, $|T_1'| \geq k$ such that $g_k(x, y)$ behaves as $\mathrm{lex}(y, x)$ on $S' \times T_1'$. This already implies that f generates ll.

So fix k a positive integer, choose $l := (3k)^k$, and pick arbitrary $T_1' \subset T_1^{(l)}$, $T_2'' \subset T_2^{(l)}$, and $S' \subset S^{(l)}$ such that $T_1' = \{b_1, \ldots, b_k\}$, $b_1 < b_2 < \ldots < b_k$, $T_2'' = \{c_1, \ldots, c_{2k}\}$, $c_1 < c_2 < \ldots < c_{2k}$, and S' is formed by the k smallest elements of $S^{(l)}$. We also define $T_2' = \{c_1, \ldots, c_k\}$. There is an automorphism β of $(\mathbb{Q}, <)$ that maps the set $\{f(s, t) \mid s \in S'', t \in T_1' \cup T_2''\}$ to $S^{(l)}$ where S'' contains the $l/3k$ smallest elements of $S^{(l)}$. Let $\alpha_1, \ldots, \alpha_k$ be automorphisms of $(\mathbb{Q}, <)$ such that α_i maps $b_1, \ldots, b_i$ to $b_1, \ldots, b_i$, and $\{b_{i+1}, \ldots, b_k\} \cup T_2'$ to T_2''. Now, we are ready to define g_k:

$$g_k(x, y) := f(\beta(f(\beta(f(\ldots \beta(f(\beta(f(x, \alpha_1(y))), \alpha_2(y))) \ldots)), \alpha_{k-1}(y))), \alpha_k(y))$$

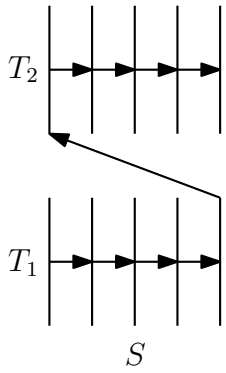


Figure 4.9: The operation from Lemma 4.50.

We verify properties of g_k. For that let $u_1, u_2 \in S'$, $v_1, v_2 \in T_1'$. If $v_1 \neq v_2$, suppose without loss of generality that $v_1 < v_2 = b_i$, $i \in [k]$. It follows that $\alpha_{i-1}(v_1) \in T_1'$ and $\alpha_{i-1}(v_2) \in T_2''$ and so $\beta(f(u_1', \alpha_{i-1}(v_1))) < \beta(f(u_2', \alpha_{i-1}(v_2)))$ for all $u_1', u_2' \in S^{(l)}$. For all $j \in [k]$, $j \geq i$, it holds that $\alpha_j(v_1), \alpha_j(v_2) \in T_1'$. Thus starting at the function on level $k - i$ from outside, the second argument of f is both for (u_1, v_1) and (u_2, v_2) from T_1' and the first argument is smaller for (u_1, v_1) than for (u_2, v_2) and remains in $S^{(l)}$. We conclude that $g_k(u_1, v_1) < g_k(u_2, v_2)$.

If $v_1 = v_2$, suppose without loss of generality that $u_1 < u_2$. As $v_1 = v_2$, second arguments of f at all levels are always the same. Operation f satisfies that $f(x, y) < f(x', y)$ for all $x, x' \in S^{(l)}$, $x < x'$ and $y \in T_1' \cup T_2''$. Thus the ordering of the first argument is preserved on all levels and as $u_1, u_2 \in S'$ and $v_1, v_2 \in T_1'$, it remains all the time in $S^{(l)}$. We conclude that $g_k(u_1, v_1) < g_k(u_2, v_2)$. □

Lemma 4.51. *Let f be a binary operation violating Betweenness and preserving $<$. Then there are $\overline{t_1}, \overline{t_2} \in$ Betweenness such that $f(\overline{t_1}, \overline{t_2})$ is injective and $f(\overline{t_1}, \overline{t_2}) \notin$ Betweenness.*

Proof. As f violates *Betweenness*, there are two triples $\overline{t_1}, \overline{t_2} \in$ *Betweenness* such that $\overline{t} := f(\overline{t_1}, \overline{t_2}) \notin$ *Betweenness*. Because f preserves $<$, we can without loss of generality assume that $\overline{t_1}[1] < \overline{t_1}[2] < \overline{t_1}[3]$ and $\overline{t_2}[1] > \overline{t_2}[2] > \overline{t_2}[3]$. If $\overline{t}$ is injective, we are done. Otherwise we distinguish two cases:

1. $\overline{t}[1] = \overline{t}[2] = \overline{t}[3]$: In that case, take a triple $\overline{s_1}$ such that $\overline{s_1}[1] < \overline{t_1}[1]$, $\overline{s_1}[2] = \overline{t_1}[2]$, and $\overline{s_1}[3] = \overline{t_1}[3]$. We also choose a triple $\overline{s_2}$ such that $\overline{t_2}[2] < \overline{s_2}[1] < \overline{t_2}[1]$, $\overline{s_2}[2] = \overline{t_2}[2]$, and $\overline{s_2}[3] = \overline{t_2}[3]$. It is straightforward to check that $\overline{s_1}[1] < \overline{s_1}[2] < \overline{s_1}[3]$ and $\overline{s_2}[1] > \overline{s_2}[2] > \overline{s_2}[3]$ and thus both triples belong to *Betweenness*. Now, consider $\overline{s} := f(\overline{s_1}, \overline{s_2})$. Because f preserves $<$, we have that $\overline{s}[2] = \overline{t}[2]$, $\overline{s}[3] = \overline{t}[3]$, and $\overline{s}[1] < \overline{t}[1] = \overline{s}[2] = \overline{s}[3]$. Therefore $\overline{s} \notin$ *Betweenness*. Take $\overline{s_1}$ instead of $\overline{t_1}$, $\overline{s_2}$ instead of $\overline{t_2}$ and proceed with case 2.

2. If exactly two entries in $\overline{t}$ have the same value, let i, j be their indices and let k be the index of the entry with the unique value. We assume that $\overline{t}[k] > \overline{t}[i]$ (the other case is symmetric). It is straightforward to verify that there is an entry in $\overline{t}$ such that making the value of this entry smaller would make $\overline{t}$ injective and it would still not be in *Betweenness*. We can without loss of generality assume that i is an index of such entry. We choose $\overline{s_1}$ so that $\overline{s_1}[i] < \overline{t_1}[i]$, $\overline{s_1}[j] = \overline{t_1}[j]$, $\overline{s_1}[k] = \overline{t_1}[k]$, and $\overline{s_1}[1] < \overline{s_1}[2] < \overline{s_1}[3]$. We choose $\overline{s_2}$ such that $\overline{s_2}[i] < \overline{t_2}[i]$, $\overline{s_2}[j] = \overline{t_2}[j]$, $\overline{s_2}[k] = \overline{t_2}[k]$, and $\overline{s_2}[1] > \overline{s_2}[2] > \overline{s_2}[3]$. As in case 1, we see that $\overline{s_1}, \overline{s_2} \in$ *Betweenness*. For tuple $\overline{s} := f(\overline{s_1}, \overline{s_2})$, we have that $\overline{s}[i] < \overline{t}[i]$, $\overline{s}[j] = \overline{t}[j]$, and $\overline{s}[k] = \overline{t}[k]$. By the choice of i we conclude that $\overline{s}$ is injective, $\overline{s} \notin$ *Betweenness* and we are done.

□

Now, we are ready to prove the crucial result of this section. We again use the product Ramsey theorem (Theorem 4.40) in a very similar way as in the proof of Lemma 4.41.

Theorem 4.52. *Let Γ be a temporal constraint language and f its binary polymorphism preserving $<$ and violating Betweenness. Then f generates* $\mathrm{lex}(x, y)$, $\mathrm{lex}(x, \mathrm{neg}(y))$, pp, *or* dual-pp.

Proof. As f violates *Betweenness* and preserves $<$, Lemma 4.51 asserts there are $\overline{t_1}, \overline{t_2} \in$ *Betweenness* such that $\overline{t} := f(\overline{t_1}, \overline{t_2}) \notin$ *Betweenness* and $\overline{t}$ is injective. As f preserves $<$, we can without loss of generality assume that $\overline{t_1}[1] < \overline{t_1}[2] < \overline{t_1}[3]$ and $\overline{t_2}[1] > \overline{t_2}[2] > \overline{t_2}[3]$ (otherwise, we can consider f with swapped arguments).

There are two possibilities for triple $\overline{t}$. Either $\overline{t}[1] > \overline{t}[2] < \overline{t}[3]$ or $\overline{t}[1] < \overline{t}[2] > \overline{t}[3]$. Depending on this condition we choose grids we apply product Ramsey theorem to. In the first case, let $S_1 := \{x \in \mathbb{Q} \mid \overline{t_1}[1] < x < \overline{t_1}[2]\}$, $S_2 := \{x \in \mathbb{Q} \mid \overline{t_1}[3] < x\}$, $T_1 := \{y \in \mathbb{Q} \mid \overline{t_2}[3] < y < \overline{t_2}[2]\}$, and $T_2 := \{y \in \mathbb{Q} \mid \overline{t_2}[1] < y\}$. Similarly, in the second case let $S_1 := \{x \in \mathbb{Q} \mid \overline{t_1}[2] < x < \overline{t_1}[3]\}$, $S_2 := \{x \in \mathbb{Q} \mid x < \overline{t_1}[1]\}$, $T_1 := \{y \in \mathbb{Q} \mid \overline{t_2}[2] < y < \overline{t_2}[1]\}$, and $T_2 := \{y \in \mathbb{Q} \mid y < \overline{t_2}[3]\}$. See Figure 4.6 for an illustration of chosen grids.

We apply the product Ramsey theorem to the grid $S_1 \times T_1$. We use the theorem for $d = m = 2$ and arbitrarily large integer k. We color each $[2]\times[2]$ subgrid of this grid according to the non-strict linear order of the four elements (see Figure 4.10). As S_1 and T_1 are infinite and, in particular, larger than $\mathcal{R}(\mathcal{R}(k))$, the theorem asserts that there are subsets $U^{(k)} \subseteq S_1$ and $V^{(k)} \subseteq T_1$ such that $|U^{(k)}| \geq \mathcal{R}(k)$, $|V^{(k)}| \geq \mathcal{R}(k)$, and $[2] \times [2]$ subgrids of $U^{(k)} \times V^{(k)}$ are monochromatic. Next, we apply the Product Ramsey theorem to the grid $U^{(k)} \times T_2$ with the same kind of coloring as above and obtain subsets $S_1^{(k)} \subseteq U^{(k)}$ and $T_2^{(k)} \subseteq T_2$. We end with applying the product Ramsey theorem to the grid $S_2 \times V^{(k)}$ with the same kind of coloring as above and obtain subsets $S_2^{(k)} \subseteq S_2$ and $T_1^{(k)} \subseteq V^{(k)}$. Again, both grids $S_1^{(k)} \times T_2^{(k)}$ and $S_2^{(k)} \times T_1^{(k)}$ have the same linear ordering induced on all

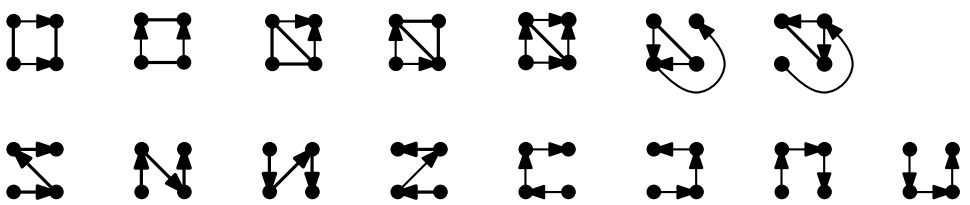


Figure 4.10: Possible non-strict linear orders on 2×2 grids. Strong lines denote equalities, oriented lines ordering. We omit edges implied by transitivity.

$[2] \times [2]$ subgrids, $|S_1^{(k)}| \geq k$, $|S_2^{(k)}| \geq k$, $|T_1^{(k)}| \geq k$, and $|T_2^{(k)}| \geq k$. The only linear orderings that can be induced on $[2] \times [2]$ subgrids of a large grid are the first two orderings depicted in the first line of Figure 4.10 and the first four orderings depicted in the second line of Figure 4.10. Therefore f on each of those three grids behaves either as a projection to one of arguments, as $\mathrm{lex}(x, y)$, as $\mathrm{lex}(y, x)$, as $\mathrm{lex}(x, \mathrm{neg}(y))$, or as $\mathrm{lex}(y, \mathrm{neg}(x))$. We see that f can behave on each of grids in 6 different ways and thus there are just 6^3 possibilities how f behaves on grids for given k. Hence, we there is an infinite set $K \subseteq \mathbb{N}$ such that f behaves the same way on grids $S_1^{(k)} \times T_1^{(k)}$ for all $k \in K$, the same way on grids $S_1^{(k)} \times T_2^{(k)}$ for all $k \in K$, and the same way on grids $S_2^{(k)} \times T_1^{(k)}$ for all $k \in K$.

If f behaves as $\mathrm{lex}(x, y)$, $\mathrm{lex}(y, x)$, $\mathrm{lex}(x, \mathrm{neg}(y))$, or $\mathrm{lex}(y, \mathrm{neg}(x))$ on some of the grids for all $k \in K$, it clearly generates either $\mathrm{lex}(x, y)$ or $\mathrm{lex}(x, \mathrm{neg}(y))$ and we are done.

So assume that f behaves as a projection on all three grids for all $k \in K$. Observe that by the choice of $S_1^{(k)}$, $S_2^{(k)}$, $T_1^{(k)}$, and $T_2^{(k)}$, and because f preserves $<$, we have that either $f(x, y) < f(\overline{t_1}[2], \overline{t_2}[2]) < f(x', y')$ for all $(x, y) \in S_1^{(k)} \times T_1^{(k)}$, $(x', y') \in (S_1^{(k)} \times T_2^{(k)}) \cup (S_2^{(k)} \times T_1^{(k)})$ (this is in case $S_1^{(k)}$ is below $S_2^{(k)}$ and $T_1^{(k)}$ is below $T_2^{(k)}$), or $f(x, y) > f(\overline{t_1}[2], \overline{t_2}[2]) > f(x', y')$ for all $(x, y) \in S_1^{(k)} \times T_1^{(k)}$, $(x', y') \in (S_1^{(k)} \times T_2^{(k)}) \cup (S_2^{(k)} \times T_1^{(k)})$ (this is in case S_1'' is above $S_2^{(k)}$ and $T_1^{(k)}$ is above $T_2^{(k)}$). Now, we distinguish two possibilities. Either f behaves as a projection to the same argument on each of three grids for all $k \in K$ or f behaves as a projection to the first argument on one grid and as a projection to the second argument on some other grid for all $k \in K$. In the first case, suppose without loss of generality that f behaves on each grid as a projection to the first argument (otherwise we can swap arguments). Due to the observation from the beginning of the paragraph, we can apply Lemma 4.50 and conclude that f generates lex.

Now, we turn our attention to the second case and show that f generates pp or dual-pp. Again, we assume without loss of generality that f behaves as a projection to the first argument on $S_1^{(k)} \times T_1^{(k)}$ for all $k \in K$. Let $S_1^{(k)} \times T_2^{(k)}$ be the grid where f behaves as a projection to the second argument (the other case is symmetric). Due to the observation from the beginning of the previous paragraph

we immediately see that f behaves either as pp (this is in case $T_1^{(k)}$ is above $T_2^{(k)}$) or as dual-pp (this is in case $T_1^{(k)}$ is below $T_2^{(k)}$) on the grid $S_1^{(k)} \times (T_1^{(k)} \cup T_2^{(k)})$ after swapping arguments for all $k \in K$. Thus f generates pp or dual-pp. □

Before we get to our final theorem, we show that if we have $\mathrm{lex}(x, y)$, or $\mathrm{lex}(x, \mathrm{neg}(y))$ operations, we can assume that the operation violating *Betweenness* is injective.

Lemma 4.53. *Let f be a binary operation violating Betweenness and preserving $<$ and g a binary injective operation preserving $<$ with major the first argument. Then $\{f, g\}$ generates a binary injective operation preserving $<$ and violating Betweenness.*

Proof. As f violates *Betweenness* and preserves $<$, we can by Lemma 4.51 choose two triples $\overline{t_1}, \overline{t_2} \in$ *Betweenness* such that $\overline{t} := f(\overline{t_1}, \overline{t_2}) \notin$ *Betweenness* and $\overline{t}$ is injective. We define operation $h(x, y) := g(g(f(x, y), x), y)$. It is easy to see that h is injective: If $h(x, y) = h(x', y')$ for some $x, x', y, y' \in \mathbb{Q}$, we infer that $y = y'$ because of the outer application of g, which is injective, and similarly that $x = x'$ because of the inner application of g. As both f and g preserve $<$, it follows that h also preserves $<$. Operation h also violates *Betweenness*, as $h(\overline{t_1}, \overline{t_2}) = g(g(\overline{t}, \overline{t_1}), \overline{t_2})$ and $\overline{t}$ is injective, g preserves $<$ and has major the first argument. □

We conclude the section by putting all the results obtained so far together:

Theorem 4.54. *Let Γ be a temporal constraint language. Then* $\mathrm{CSP}(\Gamma)$ *is tractable if Γ satisfies one of the following conditions:*

1. *Γ has a constant polymorphism, or*
2. $\mathrm{ll} \in \mathrm{Pol}(\Gamma)$*, or*
3. $\text{dual-ll} \in \mathrm{Pol}(\Gamma)$*, or*
4. $\mathrm{pp} \in \mathrm{Pol}(\Gamma)$ *and Γ also has a polymorphism providing a min-intersection closure, or*
5. $\mathrm{pp} \in \mathrm{Pol}(\Gamma)$ *and Γ also has a polymorphism providing a min-union closure, or*
6. $\mathrm{pp} \in \mathrm{Pol}(\Gamma)$ *and Γ also has a polymorphism providing a min-xor closure, or*
7. $\text{dual-pp} \in \mathrm{Pol}(\Gamma)$ *and Γ also has a polymorphism providing a max-intersection closure, or*
8. $\text{dual-pp} \in \mathrm{Pol}(\Gamma)$ *and Γ also has a polymorphism providing a max-union closure, or*

9. dual-pp $\in$ Pol(Γ) *and* Γ *also has a polymorphism providing a max-xor closure.*

Otherwise CSP(Γ) *is NP-complete.*

Proof. If Γ has a constant endomorphism, then assigning the same value to every variable of an instance I of CSP(Γ) is a solution of I. So CSP(Γ) is tractable. We have seen in Section 4.4 (Theorem 4.20) that if ll is a polymorphism of Γ, then CSP(Γ) is tractable. If Γ has the pp polymorphism and a polymorphism providing a min-intersection, a min-union, or a min-xor closure, we have shown in Section 4.5 (Theorem 4.23, Theorem 4.25, and Theorem 4.26) that CSP(Γ) is tractable. All the algorithms for dual polymorphisms are straightforward modifications of the corresponding algorithms for normal cases. Therefore all the temporal constraint satisfaction problems mentioned in the statement are really tractable.

Now, we concentrate on the NP-completeness part. Theorem 4.49 asserts that either CSP(Γ) is NP-complete (as it contains some relation whose constraint satisfaction problem is NP-complete), or Pol(Γ) contains a constant operation (in which case CSP(Γ) is tractable as we have argued above), or Pol(Γ) contains all permutations of $\mathbb{Q}$, or there is binary $f \in$ Pol(Γ) that preserves $<$ and violates *Betweenness*.

In the third case, Γ is an equality constraint language and results in Section 3 give us that CSP(Γ) is either tractable or NP-complete. CSP(Γ) is tractable only if Γ has a binary injective polymorphism g (Theorem 3.6) and it is easy to see that there exists some permutation π of $\mathbb{Q}$ such that $\pi(g(x, y)) = \mathrm{ll}(x, y)$. Therefore Γ is ll-closed or NP-complete.

So it remains to deal with the case of the binary operation f. By Theorem 4.52, the operation f either generates pp, dual-pp, lex, or weakly negated lex. In the first two cases, we are either in one of the described tractable cases or NP-complete by Theorem 4.36 and its dual counterpart. In the third and the fourth case, f generates a binary **injective** operation g preserving $<$ and violating *Betweenness* by Lemma 4.53. Finally, using Lemma 4.41 we get that g generates ll or dual-ll and thus we are in one of the described tractable cases. □

4.9 Conclusion

In this chapter, we have completely characterized complexity of constraint satisfaction problems for temporal constraint languages. We have identified several interesting tractable classes (in particular, two classes strictly containing class Ord-Horn) and shown that algorithms for some of these constraint satisfaction problems cannot follow from standard resolution techniques.

Several of the techniques introduced in this chapter can be applied not only to prove the presented results; we rather believe that they are useful in many

other contexts of constraint satisfaction complexity classification. For instance, the application of the product Ramsey theorem in Section 4.8 on *polymorphisms* of a constraint language is an example with great potential for future use; note that for a temporal constraint language, we only used the special case $d = m = 2$. This technique is probably useful for any constraint language that can be defined within an ω-categorical structure.

We also expect that our algorithmic results will inspire future algorithms for constraint satisfaction. While most of the polynomial algorithms that are known and used to solve infinite-domain constraint satisfaction problems are based on local consistency techniques, we used graph algorithms on an appropriately defined notion of constraint graph in Section 4.4. It remains an interesting open question whether this algorithm can be sped up for example using structural results from Corollary 4.14.

Another possible direction of future research is to combine the *qualitative* temporal constraints that were studied here with *quantitative* constraints, for example linear constraints, i. e., constraints of the form $15x \leq 7y + z - 3$. This moves the subject towards the area of linear programming. Surprisingly, even though Ord-Horn constraints define non-convex solution spaces, the combination of Ord-Horn constraints and linear constraints can still be solved in polynomial time [4, 26, 69]. The algorithms known for such problems solve a polynomial number of linear programs. We believe that a similar approach can be used to extend these results to a combination of ll-closed and linear constraints. Because of our structural characterization of ll-closed constraints in Corollary 4.14, it is enough to extend the result to constraints of the type $x > y \vee x > z \vee x = y = z$. It seems that such constraints can be modeled by a combination of Ord-Horn clauses and linear programming, which would give the desired result. Another possible question is, whether it is possible to efficiently solve a combination linear constraints and other tractable classes of temporal constraints we have identified.

Part II

Geometric Representations of Graphs and Graph Drawing

Chapter 5

Introduction to Geometric Representations of Graphs and Graph Drawing

In this part, we present three results on geometric representations of graphs and graph drawing. This area turns out to be an interesting interconnection of graph theory and geometry, which has developed to a broad research field over the years. We introduce geometric representations of graphs in Section 5.1 and speak about graph drawing in Section 5.2.

5.1 Geometric Intersection Graphs

One common way, how to represent a graph in a geometric fashion, is to assign to each vertex of the graph a geometric object (for example a segment in the plane) such that two objects intersect if and only if the corresponding vertices are joined by an edge. We use the following definitions to formally speak about representations:

Definition 5.1. *Let $\mathcal{F} = \{S_1, \ldots, S_n\}$ be a family of sets. The* intersection graph *of $\mathcal{F}$, denoted $\Omega(\mathcal{F})$, is the graph having $\mathcal{F}$ as a vertex set and S_i is adjacent to S_j, $i, j \in [n]$, if and only if $i \neq j$ and $S_i \cap S_j \neq \emptyset$. A graph G is an* intersection graph *if there exists a family $\mathcal{F}$ such that G is isomorphic to $\Omega(\mathcal{F})$ and in that case, $\mathcal{F}$ is called a* representation *of G. For a vertex v of an intersection graph, $\overline{v}$ denotes the set in $\mathcal{F}$ corresponding (in an isomorphism) to v.*

It is easy to observe that every graph is the intersection graph and hence the previous definition seems to be useless. But we can restrict the sets in the family $\mathcal{F}$ in some way and then the fact that the graph is the intersection graph is non-trivial again. For example, we can consider intersection graphs of segments in the plane. See Figure 5.1 for an example of the intersection graph of segments.

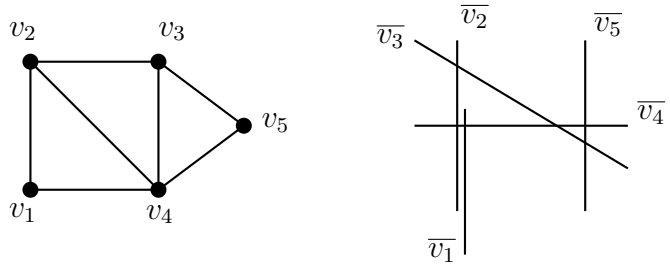


Figure 5.1: An example of the intersection graph of segments and its representation. Segments $\overline{v_1}$ and $\overline{v_2}$ should lie on the same line. They are drawn slightly apart just for clarity.

It is easy to see that not all graphs are in this class. In fact, it is NP-hard to decide whether a graph is the intersection graph of segments in the plane [72] so these graphs form a non-trivial subset of all graphs.

Now, we present a few other examples of intersection graphs:

Example 5.1. *(Classes of geometric intersection graphs)*

- Interval graphs *(Figure 5.2): These are intersection graphs of closed intervals of the real line. This class of intersection graphs is one of the oldest ones and also one of the most popular ones. It was introduced by Hajós already in 1957 [51] and lots of results about this class have been proved since. A subclass of interval graphs are* proper interval graphs *which are intersection graphs of closed intervals such that no interval contains another one (note that this is also equivalent to the requirement that all the segments have the same length [88]).*

- Circular-arc graphs *(Figure 5.3): These are intersection graphs of closed arcs of a circle. Although these graphs resemble interval graphs at the first sight, they are quite different in their nature. For example, a cycle of any length is the circular-arc graph but it is not the interval graph.*

- Disk graphs *(Figure 5.4): These are intersection graphs of closed disks in the plane. Although these graphs are still simple in their geometric nature, they already form a rather rich class of graphs. For example all planar graphs are disk graphs [96].*

- String graphs *(Figure 5.5): These are intersection graphs of simple Jordan curves (a curve is simple if it does not intersect itself) in the plane. It is easy to verify that this class of graphs contains all the other intersection classes presented so far. Still, recognizing whether a graph is a string graph is NP-complete [71, 92]. Interestingly, this was one of few recognition problems,*

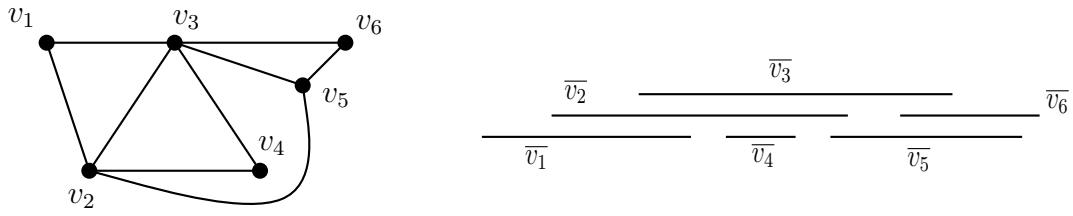


Figure 5.2: An example of the interval graph with its representation. Intervals are depicted above each other to make the length of each interval clear, although they should lie on the same line.

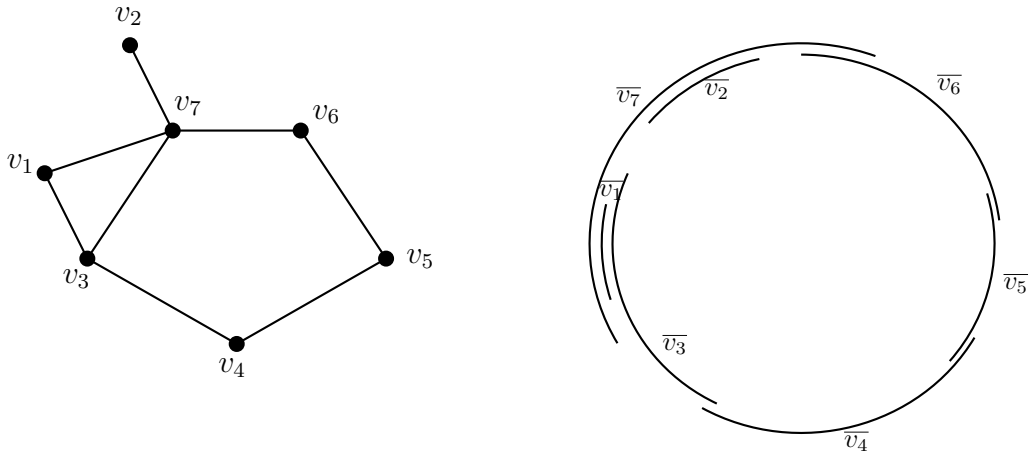


Figure 5.3: An example of the circular-arc graph with its representation. Again, arcs are drawn slightly apart for the sake of clarity.

where proving that the problem is in NP was harder than showing NP-hardness and finding the proof took a long time.

- d-box graphs*: A d-dimensional box is the Cartesian product of intervals $[a_i, b_i]$ for $1 \leq i \leq d$. A graph is a* d-box graph *if it is the intersection graph of d-dimensional boxes in $\mathcal{R}^d$. Hence interval graphs are precisely 1-box graphs. This class of intersection graphs is an example of the class where geometric objects are not in the plane but a space of arbitrary dimension.*

There are two basic directions in the study of intersection graphs. One direction of research studies which graphs are in a particular class of intersection graphs. Our result in Chapter 6 is an example of a result of this kind. The other direction of research focuses more on computational problems for graphs from some class $\mathcal{C}$ of intersection graphs. Either special properties of graphs from $\mathcal{C}$ are used to design more efficient algorithms, or it is shown that a problem is still

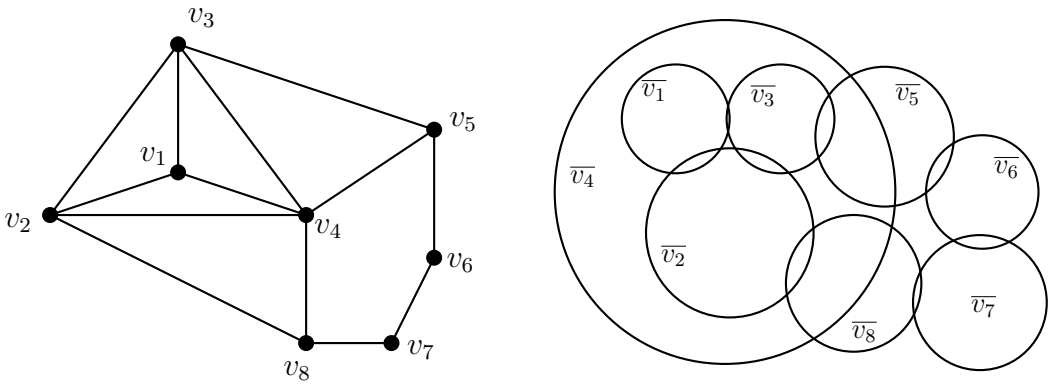


Figure 5.4: An example of the disk graph with its representation.

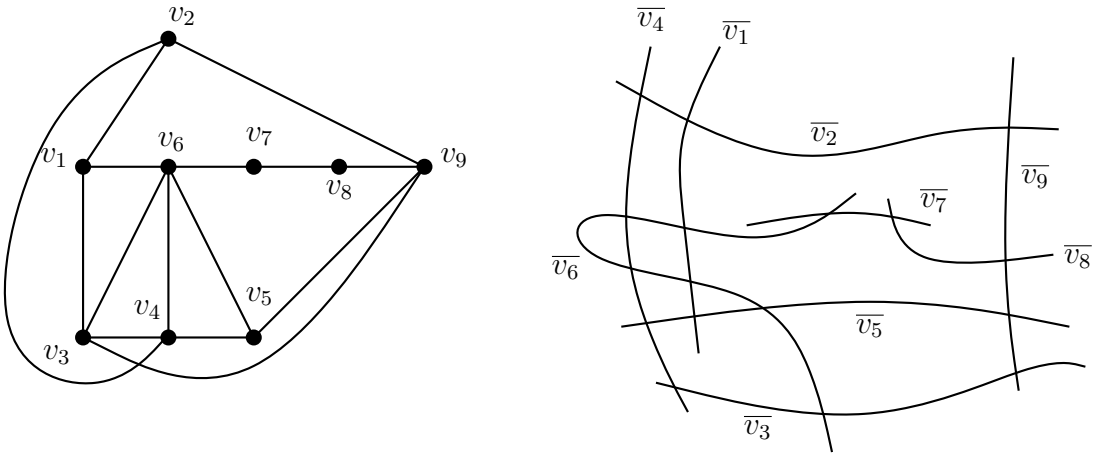


Figure 5.5: An example of the string graph with its representation.

hard even when restricted to the graphs from $\mathcal{C}$. Our result in Chapter 7 is an example of such result.

As we work with segment intersection graphs in the following two chapters, we now present a few results for them. As we already mentioned above, these are intersection graphs of straight line segments in the plane (SEG denotes this class of graphs). If a graph G is the intersection graph of straight line segments in the plane such that the segments are only in d distinct directions, we say that G is the *segment intersection graph in d directions* (d-DIR denotes this class of graphs). Note that the graph in Figure 5.1 is the 3-DIR graph. Obviously, 1-DIR graphs are exactly interval graphs. For simplicity, we often identify a vertex of a graph with the segment representing it if no ambiguity arises.

The following question of Scheinerman[93] has motivated a lot of research in the area of intersection graphs: *Is every planar graph the intersection graph of a set of segments in the plane?* Although there have been several partial answers to the question, the original question still remains unanswered. H. de Fraysseix, P. Ossona de Mendez, and J. Pach[43] and independently I. Ben-Arroyo Hartman, I. Newman and R. Ziv[52] have shown that every planar bipartite graph is the contact graph of a set of segments in two directions (the *contact graph* of segments is the intersection graph of segments such that no two segments cross). N. de Castro *et al.*[24] have extended the result and showed that planar triangle-free graphs are contact graphs of segments in three directions. Probably the best result in this direction is the paper by H. de Fraysseix and P. Ossona de Mendez[42] showing that every 4-connected 3-colorable plane graph is the contact graph of segments and that any 4-colored planar graph without an induced C_4 using 4 colors is the intersection graph of segments.

A weaker conjecture that every planar graph is the intersection graph of a set of pseudosegments (i. e., simple Jordan curves such that every two curves intersect at most once) in the plane has been proved very recently by J. Chalopin, D. Gonçalves, and P. Ochem [25].

We refer the reader to [78, 19] for a more comprehensive introduction to intersection graphs and related topics.

5.2 Graph Drawing

When we work with graphs, we usually draw them on a sheet of paper to get the idea how the graph "looks like". Also in other areas of science and engineering, visualization of graphs is a key component of support tools — for example for drawing data flow diagrams, object-oriented class hierarchies, organization charts, circuit schematics, and many others. Hence, we need algorithms which draw graphs so that they are easy to read and understand and which require no, or only small, human assistance.

Because of the combinatorial and geometric nature of the problem, and the

wide range of application domains, research in graph drawing has been conducted within several diverse areas of graph theory, algorithmics, and human-computer interaction. Books such as [6, 60] give an excellent overview of the topic.

There are several measures of the quality of drawing. One aim is to minimize the number of crossings, another to display graph symmetries and yet another to keep the area of the drawing small. These goals can be formalized as multi-objective optimization problems (e. g., construct a drawing with minimum area and minimum number of bends and crossings). Naturally, trade-offs are often necessary in order to solve these problems.

Probably the oldest problem from the area of graph drawing is: Can we draw a graph into the plane so that curves corresponding to edges do not cross? Here, *drawing of the graph* is a function Γ which maps each vertex v to a distinct point $\Gamma(v)$ and each edge $\{u, v\}$ to a simple open Jordan curve $\Gamma(u, v)$, with endpoints $\Gamma(u)$ and $\Gamma(v)$. Graphs which admit such drawings are called *planar* and are studied for a long time. A classic result independently established in [39, 97, 100] shows, that every planar graph admits a planar straight-line drawing. Drawings constructed in those works contained vertices exponentially close together and so such drawings are impractical for our purposes. More practical straight-line drawings are due to a result of de Fraysseix, Pach, and Pollack [44] and independently Schnyder [94]. They show that every planar graph with n vertices has a planar straight-line drawing with vertices drawn into a grid with $O(n^2)$ area.

Sometimes, other than straight-line drawings are desirable. For example in VLSI design, we need a drawing of a graph into the plane or 3D space, such that each edge is drawn as a polygonal chain of axis-parallel segments. These drawings are called *orthogonal*. Obviously, a graph has to have the maximum degree 4 (or 6, respectively) to allow for the orthogonal drawing. This obstacle can be avoided by drawing vertices as boxes rather than only points. In orthogonal drawing an important aesthetic criterion is to minimize the number of bends in each edge. For a given embedding of a planar graph, it is possible to find the drawing of edges which minimizes the number of bends using minimum cost flows [99]. However, it is generally NP-hard to minimize bends over all possible embeddings of a planar graph [46].

A special case of orthogonal graph drawing arises when vertices of a graph are placed on the main diagonal of the three-dimensional grid — such drawings are called *diagonal drawings*. Diagonal drawings are examples of the wider class of general-position orthogonal drawings, in which no two vertices are in a common grid plane. This model, introduced by Papakostas and Tollis [85] and Biedl [10] in the context of 3-dimensional orthogonal box-drawings, has also been used in the quadratic-time algorithm [102], which produces general position 4-bend drawings of simple graphs with maximum degree 6. This algorithm moves the vertices from an initial diagonal layout with the aim of reducing bends. In diagonal drawings, minimizing the number of bends requires finding a balanced ordering of vertices. Here balanced means that the neighbors of each vertex v are evenly distributed

to the left and right of v. Such orderings are also a starting point for several other graph drawing algorithms [61, 62, 85, 101, 102]. The problem of determining such an ordering was recently studied by Biedl *et al.* [8] and we further extend their results in Chapter 8.

Chapter 6

Representing Series-parallel Graphs as Intersection Graphs of Line Segments in Three Directions

In this chapter, we show that series-parallel graphs (i. e., K_4-minor free graphs) can be represented as contact intersection graphs of straight-line segments in three directions. Moreover, in our representations no two segments of the same direction intersect. The paper of Fraysseix *et al.*[42] covers all series-parallel graphs, but cannot be adapted to give contact representations using only three directions. Three directions are the best possible, because a series-parallel graph might contain a triangle. For the same reason, our work is not covered by the work of N. de Castro *et al.*[24]. Furthermore, our construction appears to be much simpler. This chapter is based on the paper [14] of the author.

6.1 Series-parallel Graphs

A *series-parallel network* (*SP network*) is a graph with two distinguished vertices s and t, called a *source* and a *sink*, inductively defined as follows: let G_1 and G_2 be two SP networks, where s_1 is the source of G_1, t_1 is the sink of G_1, and similarly s_2, t_2 are the source and the sink of G_2. Then the following graphs are also SP networks:

- The graph G_s that is created from G_1 and G_2 by identifying t_1 with s_2. The source of G_s is s_1 and the sink of G_s is t_2. This operation is called a *serial composition*.

- The graph G_p that is created from G_1 and G_2 by identifying s_1 with s_2 and t_1 with t_2. The source of G_p is the vertex created by identification of

the sources and the sink of G_p is the vertex created by identification of the sinks. This operation is called a *parallel composition*.

A *series-parallel graph* (*SP graph*) is a graph G such that the 2-connected components of G are SP networks. It is a well-know fact [19] that SP graphs can be equivalently characterized as K_4-minor free graphs. An SP network G is called *maximal* if each serial composition during its creation was followed by a parallel composition with an edge.

6.2 Intersection Graphs of Line Segments

We study intersection graphs of segments in three directions in which no two segments with the same direction can intersect, no three segments share a common point, and a common point of two segments is an endpoint of exactly one of them. We denote this class of graphs by PURE-3-SEG-CONTACT. The three directions we consider in our paper are *horizontal*, *vertical* and *diagonal* (defined by the equations $x = 0$, $y = 0$, and $x = y$, respectively). In the rest of the paper we consider only segments and lines in one of these three directions.

Recall that for a vertex v, $\overline{v}$ denotes the segment representing the vertex v. A *cone* is an area between two different halflines l_1, l_2 ending in a single common point. We always consider only cones with an angle smaller than or equal to $\pi/2$. We say that a cone is *bounded* by two segments σ_1, σ_2 if they both contain the point $l_1 \cap l_2$, one of the segments is contained in one of the halflines (say $\sigma_1 \subset l_1$) and the other segment shares more than one point with the other half line (say $\sigma_2 \cap l_2$ contains more than one point). See Figure 6.1 for an example of a cone.

Let C be a cone bounded by two segments σ_1 and σ_2. We say that C is *admissible* if there exists a segment $\tau \subset C$ such that τ is in the (unique) direction different from the directions of σ_1 and σ_2 and τ shares a common point with σ_1 and a common point with σ_2. Moreover, these common points are not endpoints of σ_1 or σ_2. We say that τ is *assuring admissibility* of C.

We begin with an easy lemma about touching representations.

Lemma 6.1. *Let $G = (V, E)$ be a graph such that is has a representation I_G in* PURE-3-SEG-CONTACT. *Then for each subgraph $G' = (V', E')$ of G there is a representation $I_{G'}$ of the graph G' in* PURE-3-SEG-CONTACT. *Furthermore, $I_{G'}$ is obtained from I_G by shortening some of the segments and deleting some of the segments.*

Proof. We first remove from I_G those segments that correspond to vertices not present in G'. Thus we obtain a representation of $G[V']$. Now, let $F := E(G[V'])\setminus E'$. If $F = \emptyset$, we are done. Otherwise consider $e = \{u, v\} \in F$. Segments $\overline{u}$ and $\overline{v}$ meet in a common point in $I_{G[V']}$ and without loss of generality $\overline{u}$ ends there. We can shorten $\overline{u}$ so that it does not intersect $\overline{v}$, but all other intersection points are

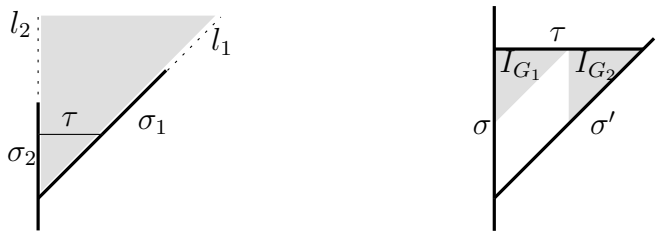


Figure 6.1: In the left: a cone bounded by segments σ_1 and σ_2. The cone is admissible and τ assures its admissibility. In the right: an example of a representation of a serial composition of G_1 and G_2.

preserved. Obtained representation will clearly represent graph $G[V'] - e$. If we repeat the above procedure for each edge $e \in F$, we clearly obtain a representation I'_G of G'. □

6.3 Representations of SP Graphs

Now, we present a lemma about representations of SP networks.

Lemma 6.2. *For every maximal SP network G with the source s and the sink t, every admissible cone C, bounded by two segments σ and σ', and for every segment τ assuring admissibility of C there exists a representation I_G of G in* PURE-3-SEG-CONTACT *such that $\sigma = \overline{s}$, $\sigma' = \overline{t}$, and I_G is inside the triangle defined by σ, σ', and τ.*

Proof. We prove the lemma by induction on the size of G. If G contains just s and t connected by an edge, we are immediately done (σ and σ' already represent the graph).

If G is a serial composition of graphs G_1 and G_2, let u denote the common node of the two composed graphs. We represent u as a segment $\overline{u} = \tau$ given in the statement of the lemma. It is now easy to check (see Figure 6.1) that there are admissible cones bounded by σ and τ and bounded by σ' and τ. Hence, G_1 can be represented in the admissible cone bounded by σ and τ limited by some segment as shown in Figure 6.1 by induction. Similarly, we can represent G_2 between σ' and τ.

Otherwise G is a parallel composition of G_1 and G_2. We can first by induction represent G_1 in C limited by τ and then choose a segment τ' in C parallel to τ and close enough to the intersection of σ and σ' so that it does not intersect any segment of the representation of G_1. Then we can represent G_2 in C limited by τ' by induction, and we are done. □

Now, we introduce a lemma allowing us to make a graph 2-connected. First, we state a well-known result:

Proposition 6.3. *A graph G contains K_4 as a minor if and only if G contains a subdivision of K_4 as a subgraph.*

Lemma 6.4. *Let G be a graph without K_4 as a minor. Then there is a graph H such that G is a subgraph of H, H is 2-connected, and H does not contain K_4 as a minor.*

Proof. If G contains several connected components, we can connect the components by new edges so that G becomes connected and we do not introduce a K_4 minor. So assume that G is connected. We proceed by induction on the number of blocks (a block is an edge-maximal 2-connected subgraph or an edge) of G. If G contains only one block, we are done since G is 2-connected. Let C_1 and C_2 be two different blocks of G sharing an articulation v. Let u_1 and u_2 be neighbors of v in C_1 and C_2, respectively. Let G' be the graph G where we add the edge $\{u_1, u_2\}$. Clearly, the number of blocks in G' is one less than the number of blocks in G. Hence, if we did not introduce K_4 minor, we can proceed by induction. Suppose G' contains a K_4 minor. By Proposition 6.3 G' contains a subdivision S of K_4. Let $W := \{w_1, \ldots, w_4\}$ be the vertices of degree three in S. As there are three vertex disjoint paths from each vertex of W to each other vertex of W, vertices in W must be contained in one 2-connected component of G'. For the same reason W must be contained even in a 2-connected component of G, as for any vertex $x \in C_1 - v$ there are at most two vertex disjoint paths from x to any vertex in $C_2 - v$. Without loss of generality let S be contained in C_1. It follows that S must contain a path that goes from C_1 through v to C_2 and via the edge $\{u_1, u_2\}$ back to C_1. But if we replace the path between v and u_2 by an edge $\{v, u_1\}$, we obtain a subdivision of K_4 that is a subgraph of G. Proposition 6.3 immediately yields a contradiction. □

Theorem 6.5. *Let $G = (V, E)$ be an SP graph. Then G has a representation $I_G \in$* PURE-3-SEG-CONTACT.

Proof. First, we use Lemma 6.4 to obtain a graph H that is 2-connected, does not contain a K_4 minor, and contains G as a subgraph. Then we add edges to H and obtain a maximal SP network H'. The application of Lemma 6.2 to H' yields a representation that lies between two arbitrary touching segments σ and σ' in different directions. Finally, we apply Lemma 6.1 and obtain a representation of G. □

6.4 Conclusion

We have presented a simple construction showing that each series-parallel graph has a representation as a contact graph of segments on three directions. Although it is unlikely that one would succeed in answering the question of Scheinerman using such simple techniques as in this chapter, it may be still possible to answer the question for some restricted classes of planar graphs.

Chapter 7

Fixed Parameter Tractability of Independent Set in Segment Intersection Graphs

In this chapter, we present a fixed parameter tractable algorithm for the Independent Set problem in 2-DIR graphs and also its generalization to d-DIR graphs. Moreover, our algorithms are robust in the sense that they do not need the actual representation of the input graph and they answer correctly even if they are given a graph from outside the promised class. The chapter is based on the paper [63] of the author.

7.1 Introduction

Let $G = (V, E)$ be a simple undirected graph. In the following we use n for the number of vertices $|V|$ and m for the number of edges $|E|$. Let deg v denote the number of edges incident with v, $N(v)$ the open neighborhood of v (i. e. $N(v) := \{u \in V \mid \{u, v\} \in E\}$) and $N[v]$ the closed neighborhood of v (i. e. $N[v] := N(v) \cup \{v\}$). If $U \subseteq V$, then we write $G[U]$ for the subgraph induced on the set U (i. e., the graph $G' = (U, E')$ where $E' := \{\{u, v\} \in E \mid u \in U, v \in U\}$). We write $G - v$ instead of $G[V \setminus \{v\}]$ for the sake of brevity. A set of vertices U is called *independent* if $G[U]$ contains no edges, and it is called a *clique* if $G[U]$ is a complete graph. The symbols $\alpha(G)$ and $\omega(G)$ denote the largest size of an independent set and of a clique, respectively.

The problem of finding an independent set of maximum size is a well known NP-hard problem, which remains NP-hard for many restricted graph classes. For instance for planar cubic graphs [45]. Though it is solvable in polynomial time in interval graphs (1-DIR graphs), it is NP-hard already for 2-DIR graphs [73]. Also from the fixed parameter complexity point of view, the problem is hard. It is W[1]-complete when parameterized by the size of the independent

set k [34]. Therefore it is reasonable to ask for its parameterized complexity for restricted graph classes. As observed many times (and noted e. g., in the recent monograph of Niedermeier [83]), the problem is trivial for planar graphs (when parameterized by k) since every planar graph is guaranteed to have $\alpha(G) \geq \frac{n}{4}$. Fellows asked [private communication, 2005] what the fixed parameter complexity of this problem is when restricted to 2-DIR graphs. Note that the answer is not as obvious as for planar graphs, since 2-DIR do not guarantee independent sets of any nontrivial size — all cliques are 2-DIR graphs (in fact even 1-DIR graphs).

In this chapter, we answer this question, even in a more general setting for d-DIR graphs. This complements independent result of Marx [77] showing that finding an independent set in segment intersection graphs (i. e., with unlimited number of directions) is W[1]-complete. The author also presents a fixed parameter tractable algorithm for the case when the number of directions is bounded but his algorithm requires a segment representation of an input graph.

In Section 7.3, we present a fixed parameter tractable algorithm (i. e., an algorithm running in time $O(f(k)n^{O(1)})$) for the Independent Set problem in 2-DIR graphs. In Section 7.4, we generalize this algorithm to an FPT algorithm for d-DIR graphs (where both d and k are parameters). Our algorithms are designed to be "robust" — i. e., they either output an independent set of size k or answer that there is no independent set of size k or detect that the given graph is not a 2-DIR (d-DIR) graph. Hence they answer correctly even if the input graph is not from the required class. This fact is important especially because of the fact that SEG, d-DIR, and even 2-DIR graphs are NP-hard to recognize [70, 72]. Thus it is not possible for an algorithm to first construct a segment representation of the graph and then use it in its computation.

7.2 Reduction Step

Observation 7.1. *Let $G = (V, E)$ be an arbitrary graph and $u, v \in V$ two of its vertices such that $N[u] \subseteq N[v]$. Then $\alpha(G) = \alpha(G - v)$.*

Proof. Clearly, $\alpha(G) \geq \alpha(G - v)$ and so we concentrate on the second inequality. Note that $N[u] \subseteq N[v]$ implies $\{u, v\} \in E$. Hence at most one of the vertices u, v can be in any independent set. If an independent set S contains v, the set $S \setminus \{v\} \cup \{u\}$ is also independent and of the same size. That proves the statement of the observation. □

We call a graph *reduced* if no closed neighborhoods are in inclusion. By consecutive application of the reduction step described in Observation 7.1, we reduce the input graph G to a reduced graph G' such that $\alpha(G) = \alpha(G')$. Such reduction can be performed in time $O(mn)$ (and independently of k). In pseudocode of algorithms presented below, we use procedure `Reduce` to perform this graph reduction.

7.3 Algorithm for 2-DIR Graphs

In this section, we present an FPT algorithm for Independent Set in 2-DIR graphs. We begin with an easy observation:

Lemma 7.2. *Let $G = (V, E)$ be a 2-DIR graph. Then for each $v \in V$, the graph $G[N(v)]$ is an interval graph.*

Proof. Consider a 2-DIR representation R_G of the graph G. The vertex v is represented as a segment $\overline{v}$ in one of the two directions. Consider the set of segments $I_v := \{\overline{v} \cap \overline{u} \mid \{u, v\} \in E\}$ This set clearly contains one segment for each neighbor of v and all the segments have the same direction (some of them can be a single point). Hence, I_v induces an interval graph. Moreover, it can be easily verified that I_v is an interval representation of $G[N(v)]$. □

Next, we derive an important general lemma about d-DIR graphs.

Lemma 7.3. *Let R_G be a d-DIR-representation of a reduced graph $G = (V, E)$. If $|V| > ((k-1)(d-1)+1)d(k-1)^2$, then either there are segments $s_1, \ldots, s_k \subseteq R_G$ such that all of them are parallel with the same direction and no two of them lie on the same line, or there is a line l which contains k pairwise non-intersecting segments.*

Proof. Suppose that R_G does not contain k parallel segments lying on k different lines. Let l be a line containing at least one segment. Denote by V_l the set of vertices corresponding to the segments lying on the line l and let G_l be the (interval) graph $G[V_l]$. Since G is reduced, G_l is a proper interval graph (no interval can be contained in another one).

Let $c := (k-1)(d-1)+1$. First, we prove that $\omega(G_l) \leq c$. Suppose for contradiction that G_l contains a clique of size $c+1$. Order the vertices from left to right by the left endpoints of their intervals (or by the the order of their right endpoints — since this is a proper interval representation, these two orderings are the same). Let $C_2, \ldots, C_{c+1}$ be the lexicographically minimal cliques of sizes $2, \ldots, c+1$, respectively, and denote their vertices $C_i = \{v_1^i, v_2^i, \ldots, v_i^i\}$ (in the ordering). Consider the last two vertices v_c^{c+1}, v_{c+1}^{c+1} of C_{c+1}. The vertex v_c^{c+1} must have a neighbor u_{c+1} (in the graph G) that is not a neighbor of v_{c+1}^{c+1} (otherwise v_c^{c+1} would be thrown out in the reduction step). The segment $\overline{u_{c+1}}$ cannot lie on the line l since v_c^{c+1} lies before v_{c+1}^{c+1} and so the vertex u_{c+1} would have to lie before v_{c+1}^{c+1} and we would get a lexicographically smaller clique $C_{c+1} \setminus \{v_{c+1}^{c+1}\} \cup \{u_{c+1}\}$ of the size $c+1$. The same argument for cliques $C_2, \ldots, C_c$ yields vertices $u_2, \ldots, u_c$ in the directions different from the direction of l. Let $r_i := \overline{v_{i-1}^i} \setminus \overline{v_i^i}$ be the part of the segment representing v_{i-1}^i that is not part of the segment representing v_i^i, for $2 \leq i \leq c+1$. Since the segments r_i do not overlap and each segment $\overline{u_i}$ intersects v_{i-1}^i in r_i, each of the vertices $u_2, \ldots, u_{c+1}$ lies on a different line. But by the choice of $c = (k-1)(d-1)+1$ (applying the pigeon-hole principle), there must be a

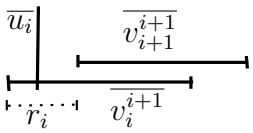


Figure 7.1: A situation of the last two vertices of the i-th clique.

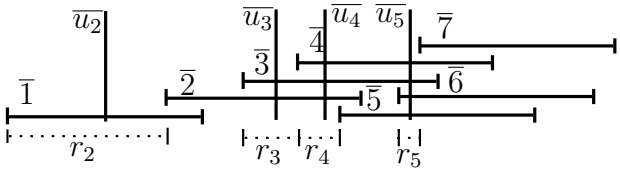


Figure 7.2: A situation in a graph G_l for $c = 4$. The lexicographically minimal cliques are: $C_2 = \{1, 2\}, C_3 = \{2, 3, 4\}, C_4 = \{2, 3, 4, 5\}, C_5 = \{3, 4, 5, 6, 7\}$.

direction with at least k segments from $u_2, \ldots, u_{c+1}$. This is a contradiction with the assumption that there are no such k segments. See Figure 7.1 for an example of a situation in one clique and Figure 7.2 for an example of the whole situation on the line l.

Since G_l is an interval graph, it is in particular a perfect graph and hence $\alpha(G_l) \cdot \omega(G_l) \geq |V_l|$. As we assumed that $|V| > cd(k-1)^2$ and because for every direction, there are at most $k-1$ lines containing at least one segment in this direction (otherwise we easily find k parallel segments), there must be a line l with at least $c(k-1)+1$ segments. Hence, for this line l, we conclude that $\alpha(G_l) \geq \lceil \frac{c(k-1)+1}{c} \rceil = k$ as claimed by the statement of the lemma. □

Corollary 7.4. *Let $G = (V, E)$ be a reduced* 2-DIR *graph. Then either $|V| \leq 2k(k-1)^2$ or $\alpha(G) \geq k$.*

Proof. The corollary trivially follows from Lemma 7.3. □

Algorithm 7.1.

Let $G = (V, E)$ be the input graph.
`Reduce(`G`)`
`if` $|V| \leq 2k^2(k^2-1)^2$ `then begin`
  Compute the adjacency matrix of G.
  `for each` $I \subseteq V$, $|I| = k$ `do`
    `if` I is independent `then`
      Output I and `exit.`
  Output that there is no independent set of size k in G.
`end`
`else begin`

```
    Find greedily an inclusion-wise maximal independent set I.
    if |I| ≥ k then
      Output I and exit.
    for each v ∈ I do begin
      Compute G' = G[N(v)].
      if G' is not an interval graph then
        Output that G is not a 2-DIR graph and exit.
      Find a maximum independent set I of G' (using the fact that
        G' is an interval graph.
      if |I| ≥ k then
        Output I and exit.
    end
    Output that G is not a 2-DIR graph.
  end
```

Theorem 7.5. *Algorithm 7.1 finds in time $O(k^{6k+2}+mn+n^2)$ for a given graph $G=(V,E)$ either an independent set of size at least k or answers that there is no independent set of size (at least) k or detects that G is not a* 2-DIR *graph.*

Proof. The algorithm first performs the reduction step as described in Section 7.2. This takes $O(mn)$ time. Let $G' = G[V']$ be the resulting graph. If $|V'| \leq 2k^2(k^2-1)^2 = O(k^6)$, we run a brute-force algorithm that tries all subsets of vertices of size k — there are $O(k^{6k})$ such sets — and for each of them, we check whether it is an independent set or not (this can be easily done in time $O(k^2)$ if we have precomputed adjacency matrix of the graph). Hence in this case we are done in the time $O(k^{6k+2}+mn+n^2)$.

If $|V'| > 2k^2(k^2-1)^2$, then Corollary 7.4 asserts that either G' is not a 2-DIR graph or it contains an independent set of size k^2. We find an (inclusion-wise) maximal independent set I in G' — this can be done in $O(m+n)$ time by a greedy algorithm. If it has size at least k, we are done. Otherwise we claim that there must be a vertex $v \in I$ such that there is an independent set of size k in $N(v)$ (if G was a 2-DIR graph). This follows from the fact that every vertex not in I is adjacent to at least one vertex in I. Hence if J is an independent set of size k^2, then some vertex in I must be adjacent to at least k vertices from $J \setminus I$. So it is enough to find a maximum independent set in $G'[N(v)]$ for each $v \in I$. From Lemma 7.2, we know that each of the graphs $G'[N(v)]$ is an interval graph, and thus its independence number can be computed in polynomial time. Hence, for every vertex v of I, we verify that $G'[N(v)]$ is an interval graph (and reject G as not being 2-DIR if it is not) in $O(m+n)$ time and we compute a maximum independent set in time $O(m+n)$. If none of these independent sets has size at least k, we again reject G as not being a 2-DIR graph. □

Note that if we get a 2-DIR representation of the graph G as part of the input, the algorithm from Theorem 7.5 would be much simpler. In the kernelization step,

it would suffice to have enough vertices guaranteeing an independent set of size k (and not k^2) and we could find it by simply checking the types of independent sets described in Lemma 7.3.

7.4 Algorithm for d-DIR Graphs

In this section, we present an algorithm for Independent Set in d-DIR graphs. The algorithm is in fact simpler than the one for 2-DIR graphs but its correctness is less obvious and its running time is worse. Due to Lemma 7.3, a large enough reduced d-DIR graph must contain a big independent set. Hence, we use a similar trick as for 2-DIR graphs and for d-DIR graphs whose number of vertices does not guarantee an independent set of size k^{3d}, we run a brute force algorithm to decide the existence of an independent set of size k. For larger graphs, we indirectly show that there is a sufficiently limited number of cliques in whose neighborhoods it suffices to search for independent sets of size k, and that those neighborhoods have a simple structure. Now, we formalize the above statements:

Corollary 7.6. *Let $G = (V, E)$ be a reduced d-DIR graph. Then either $|V| \leq ((k-1)(d-1)+1)d(k-1)^2$ or $\alpha(G) \geq k$.*

Proof. The statement trivially follows from Lemma 7.3. □

Lemma 7.7. *Let R_G be a d-DIR representation of a reduced graph $G = (V, E)$. Let d_v be the direction of the segment $\overline{v}$ for $v \in V$. Denote by G' the graph obtained from $G[N(v)]$ by the reduction procedure of Section 7.2. Then G' contains at most two vertices that were represented by segments in the direction d_v in R_G.*

Proof. Let l be the line containing $\overline{v}$. Note that G' contains only neighbors of v. Let V_l be the set of neighbors of v that are represented by segments parallel with the direction d_v, and let S_l be the set of segments representing them. Then all segments of S_l lie on the line l. Since G was reduced, no segment in S_l can contain another one (including $\overline{v}$ itself) and hence each segment in S_l contains exactly one endpoint of the segment representing v. Let $v_1, \ldots, v_t$ be the vertices of S_l whose segments contain the left endpoint of $\overline{v}$, ordered from left to right. Since apart from S_l, $G[N(v)]$ contains only vertices corresponding to the segments crossing $\overline{v}$, we see that $N[v_1] \subseteq N[v_2] \subseteq \ldots \subseteq N[v_t]$, and hence only the vertex v_1 survives the reduction step and is included in G'. Similarly, at most one of the vertices represented by segments containing the right endpoint of $\overline{v}$ remains in G'. See Figure 7.3 for an example of a reduction of a neighborhood. □

Theorem 7.8. *There is an algorithm running in time $O(d^{2k}k^{9dk+2} + n^2 + k^{3d}mn)$ that given a graph $G = (V, E)$ either finds an independent set of size k or answers that there is no independent set of size k or detects that G is not a d-DIR graph.*

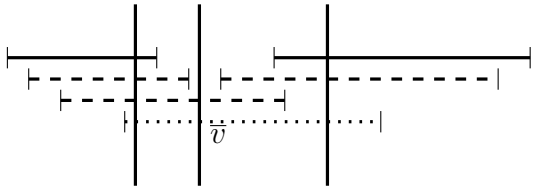


Figure 7.3: A vertex v (dotted interval) and its neighbors. Dashed intervals are the vertices that are removed in the reduction step on $G[N(v)]$. Vertical thick lines represent points of intersection with segments in other directions.

Proof. The algorithm first performs the reduction step as described in Section 7.2. This takes $O(mn)$ time. Let $G' := G[V']$ be the resulting graph. If $|V'| \leq ((k^{3d}-1)(d-1)+1)d(k^{3d}-1)^2 = O(d^2k^{9d})$, we run a brute-force algorithm that tries all subsets of vertices of size k — there are $O(d^{2k}k^{9kd})$ such sets — and for each set checks whether it is an independent set or not (that can be easily done in time $O(k^2)$ provided we have precomputed the adjacency matrix of G'). Hence, in this case we are done in time $O(d^{2k}k^{9dk+2} + n^2 + mn)$.

If $|V'| > ((k^{3d}-1)(d-1)+1)d(k^{3d}-1)^2$, we know by Corollary 7.6 that either G' is not a d-DIR graph or G' contains an independent set of size k^{3d}. Now, we start the following recursive procedure:

Algorithm 7.2.

```
procedure FindIndependentSet(G, depth)
begin
  Find a maximal independent set I in the given graph G.
  if |I| ≥ k then
    Output I and abort (at all levels of recursion).
  if depth ≥ 3d then
    Output that G is not a d-DIR graph and abort.
  for each v ∈ I do begin
    G' := G[N(v)]
    Reduce(G')
    if |V(G')| ≥ k then
      FindIndependentSet(G', depth + 1)
  end
end
```

If the recursive procedure did not find any independent set, we answer that G is not a d-DIR graph. Hence, the overall algorithm for d-DIR graphs looks as follows:

Algorithm 7.3.

Let $G = (V, E)$ be the input graph.

```
Reduce(G)
if |V| ≤ ((k^{3d} − 1)(d − 1) + 1)d(k^{3d} − 1)^2 then begin
  Compute the adjacency matrix of G.
  for each I ⊆ V, |I| = k do
    if I is independent then
      Output I and exit.
  Output that there is no independent set of size k in G.
end
else begin
  FindIndependentSet(G,0)
  Output that G is not a d-DIR graph.
end
```

The time complexity of the recursive call of `FindIndependentSet` is obviously $O(k^{3d}mn)$. What remains to be proved is that if G is a d-DIR graph, then the recursion always finds an independent set of size k (recall that if G is a d-DIR graph it must contain such independent set). Correctness of the other answers easily follows from this fact. If G is a d-DIR graph, it contains an independent set of size k^{3d}. Hence some vertex from the maximal independent set must have an independent set of size k^{3d-1} in its neighborhood (by a similar argument as in Theorem 7.5). By induction and because the reduction does not change the size of a maximum independent set, we argue that there is a branch of recursion whose graph considered in the depth i of the recursion contains an independent set of size k^{3d-i}. By Lemma 7.7, we know that during the recursion we can choose at most three vertices from each direction. Thus in the depth of recursion $3d-1$ every obtained graph can contain at most one vertex. Hence, we see that if G is a d-DIR graph, then the recursion must stop before the depth $3d-1$ as otherwise we get a contradiction. There are only two ways of stopping the recursion before the depth $3d$. Either we get a graph with less than k vertices or we find an independent set of size k. As there must be a branch of recursion whose graph at the depth i contains an independent set of size k^{3d-i}, we know that this branch cannot stop because of the lack of vertices and hence it must stop because it found an independent set of size k. □

7.5 Conclusion

We have presented efficient FPT algorithms for the Independent Set problem restricted to intersection graphs of segments with segments lying in a bounded number of direction (where both the size of the sought independent set k and the number of directions d are considered as parameters). Given the simplicity of the situation for two directions, it might be interesting to determine whether

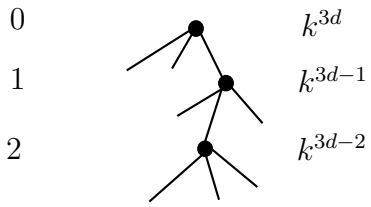


Figure 7.4: A recursion tree with a depth of the recursion written on the left and guaranteed independent set size written on the right.

there is a fixed parameter tractable algorithm running in time $2^{O(k)} \cdot p(m, n)$ in this case.

It is worth mentioning the parallel to the (classical) complexity of the Clique problem restricted to intersection graphs. The Clique problem is known to be NP-hard for string graphs and also for intersection graphs of convex sets, polynomial time solvable (but not FPT with regard to d as parameter) for d-DIR graphs and its complexity still remains open for SEG graphs. This question was asked in [73], for a recent survey on its development and related questions cf. [5].

Chapter 8

On the Complexity of the Balanced Vertex Ordering Problem

In this chapter, we consider the problem of finding a balanced ordering of the vertices of a graph. More precisely, we want to minimize the sum, taken over all vertices v, of the difference between the number of neighbors to the left and right of v. This problem, which has applications in graph drawing, was recently introduced by Biedl *et al.* [Discrete Applied Math. 148:27–48, 2005]. They proved that the problem is solvable in polynomial time for graphs with maximum degree three, but is NP-hard for graphs with maximum degree six. One of our main results is to close the gap in these results, by proving NP-hardness for graphs with maximum degree four. Furthermore, we prove that the problem remains NP-hard for planar graphs with maximum degree four and for 5-regular graphs. On the other hand, we introduce a polynomial time algorithm that determines whether there is a vertex ordering with total imbalance smaller than a fixed constant, and a polynomial time algorithm that determines whether a given multigraph with even degrees has an 'almost balanced' ordering. Results in this chapter are based on the paper [65] of the author.

8.1 Introduction

Let $G = (V, E)$ be a multigraph without loops. An *ordering* of G is a bijection $\sigma : V \to \{1, \ldots, |V|\}$. For $u, v \in V$ with $\sigma(u) < \sigma(v)$, we say that u is *to the left* of v and that v is *to the right* of u. The *imbalance* of $v \in V$ in σ, denoted by $B_\sigma(v)$, is

$$\big|\,|\{e \in E : e = \{u, v\}, \sigma(u) < \sigma(v)\}| - |\{e \in E : e = \{u, v\}, \sigma(u) > \sigma(v)\}|\,\big|.$$

When the ordering σ is clear from the context we simply write $B(v)$ instead

of $B_\sigma(v)$. The *imbalance* of ordering σ, denoted by $B_\sigma(G)$, is $\sum_{v \in V} B_\sigma(v)$. The minimum value of $B_\sigma(G)$, taken over all orderings σ of G, is denoted by $M(G)$. An ordering with imbalance $M(G)$ is called *minimum*. The following two facts hold for every ordering:

- Every vertex of odd degree has imbalance at least one.
- The two vertices at the beginning and at the end of the ordering have imbalance equal to their degrees.

These two facts imply the following lower bound on the imbalance of an ordering. Let $\text{odd}(A)$ denote the number of odd degree vertices among the vertices of $A \subseteq V$. Let $(d_1, \ldots, d_n)$ be the sequence of vertex degrees of G, where $d_i \leq d_{i+1}$ for all $i \in \{1, 2, \ldots, n-1\}$. Then

$$B_\sigma(G) \geq \text{odd}(V) - (d_1 \bmod 2) - (d_2 \bmod 2) + d_1 + d_2.$$

An ordering σ is *perfect* if the above inequality holds with equality. PERFECT ORDERING is the decision problem whether a given multigraph G has a perfect ordering. This problem is clearly in NP.

Biedl *et al.* [8] gave a polynomial time algorithm to compute a minimum ordering of graphs with maximum degree at most three. On the other hand, they proved that it is NP-hard to compute a minimum ordering of a (bipartite) graph with maximum degree six.

One of the main results of this paper is to close the above gap in the complexity of the balanced ordering problem with respect to the maximum degree of the graph. In particular, we prove that the PERFECT ORDERING problem is NP-complete for simple graphs with maximum degree four.

Whether the balanced ordering problem is efficiently solvable for planar graphs with maximum degree four is of particular interest since a number of algorithms for producing orthogonal drawings of planar graphs with maximum degree four start with a balanced ordering of the vertices [9, 85]. We answer this question in the negative by proving that the PERFECT ORDERING problem is NP-complete for planar simple graphs with maximum degree four.

Our third NP-hardness result states that finding an ordering with minimum imbalance is NP-hard for 5-regular simple graphs. All of these NP-hardness results for ordering problems are presented in Section 8.3. The proofs are based on reductions from various satisfiability problems. Section 8.2 contains several NP-completeness results for used satisfiability problems. While the complexity of most of these satisfiability problems follows from a general result by Schaefer [91], we believe that our proofs are simpler and the result for PLANAR 2–IN–4SAT is of independent interest.

In Section 8.4, we present our positive complexity results. In particular, we describe a polynomial time algorithm that determines whether a given graph has

an ordering with at most k imbalanced vertices for any constant k. This algorithm has several interesting corollaries. For example, the PERFECT ORDERING problem can be solved in polynomial time for a multigraph in which all the vertices have even degrees (in particular, for 4-regular multigraphs).

8.2 NP-Hardness of Satisfiability Problems

In this section, we prove several NP-hardness results about various satisfiability problems. Note that the results in this section could be also achieved by verifying conditions of a general theorem of Schaefer [91], but we feel that our proofs are simpler. First, we introduce several basic definitions about satisfiability. Throughout this chapter, formulae are considered to be in a *conjunctive normal form*. That is, each formula φ is a conjunction of some m clauses $c_1 \wedge c_2 \wedge \cdots \wedge c_m$, where each *clause* c_i is a disjunction of n_i literals $l_1^i \vee l_2^i \vee \cdots \vee l_{n_i}^i$ for all $i \in \{1, \ldots, m\}$. A *literal* is either a variable or its negation. The *size* of a clause is the number of literals in it. Suppose φ is a formula with variables $x_1, \ldots, x_n$. The *incidence graph* of φ is the bipartite graph with vertex set $\{c_1, \ldots, c_m, x_1, \ldots, x_n\}$, where $\{c_i, x_j\}$ is an edge if and only if the variable x_j occurs in the clause c_i. A *truth assignment* of a formula φ with variables $x_1, \ldots, x_n$ is an arbitrary function $t : \{1, \ldots, n\} \to \{0, 1\}$. The values 0 and 1 are also sometimes called *false* and *true*, respectively. A truth assignment t is *satisfying* if there is at least one true literal in every clause. The formula φ is *satisfiable* if it has at least one satisfying truth assignment.

The decision problem asking whether a given formula φ is satisfiable is called SAT. If we assume that every clause in the given formula φ has size exactly three, then the decision problem asking whether φ is satisfiable is called 3SAT. Two common variants of 3SAT are *Not–All–Equal 3-Satisfiability* (NAE–3SAT for short) and *1–in–3 Satisfiability* (1–IN–3SAT). Both these problems are defined on formulae in which each clause has size exactly three. Furthermore in NAE–3SAT the formulae are without negations. A truth assignment t is *NAE satisfying* if each clause has at least one true and at least one false literal. t is called *1–in–3 satisfying* if each clause has exactly one true literal. The notions of *NAE satisfiable* and *1–in–3 satisfiable* formulae, and the corresponding decision problems are defined in the obvious way. It is well known that SAT, NAE–3SAT, and 1–IN–3SAT are NP-complete (see [91]).

We say that a formula φ for which all clauses have five literals is *2–or–3–in–5 satisfiable* if there exists a truth assignment such that in each clause either two or three literals are true. Let 2–OR–3–IN–5SAT denote the decision problem asking whether a given formula without negations is 2–or–3–in–5 satisfiable.

Lemma 8.1. *The problem* 2–OR–3–IN-5SAT *is NP-complete.*

Proof. The problem clearly belongs to NP. We prove NP-completeness by a re-

duction from NAE–3SAT, which is NP-complete [91]. Suppose we are given a formula φ without negations. Create a formula φ' from φ by adding a new clause $c_0 := x \vee x \vee x \vee x' \vee x'$, and by substituting each clause c with clause $c \vee x \vee x'$ where x and x' are new variables. Given a NAE-satisfying truth assignment t for φ, a 2-or-3-in-5-satisfying truth assignment t' for φ' can be created by setting $t'(x) := 0$ and $t'(x') := 1$. Also, if t' is a 2–or–3–in–5 satisfying truth assignment for φ', then $t'(x) = \neg t'(x')$ by clause c_0. Thus, restricting t' to the variables of φ, we obtain a NAE-satisfying truth assignment for φ. □

The next lemma uses the following version of the satisfiability problem. Let φ be a formula in which all clauses have four literals. A truth assignment t is *2–in–4 satisfying* if each clause in φ has exactly two true literals. φ is *2–in–4 satisfiable* if there exists a 2–in–4 satisfying truth assignment. 2–IN–4SAT is the decision problem asking whether a given formula φ is 2–in–4 satisfiable.

Lemma 8.2. *The problem* 2–IN–4SAT *is NP-complete for formulae without negations.*

Proof. The problem is obviously in NP. We prove its NP-completeness by reduction from 1–IN–3SAT, which is NP-complete for formulae without negations [91]. Let ψ be a formula given as an input for 1–IN–3SAT. Create a formula ψ' with a new variable v by adding v to each clause of ψ. We now show that ψ is 1–in–3 satisfiable if and only if ψ' is 2–in–4 satisfiable. If t is a 1–in–3 satisfying truth assignment for ψ, then by setting $t'(v) := 1$ and $t'(x) := t(x)$ for each variable $x \neq v$ of ψ', we obtain a 2–in–4 satisfying truth assignment for ψ'. Conversely, if t' is a 2–in–4 satisfying truth assignment for ψ', then either t' (in the case that $t'(v) = 1$) or $1 - t'$ (in the case that $t'(v) = 0$) restricted to variables of ψ is a 1–in–3 satisfying truth assignment for ψ. □

Now, we strengthen the result from the previous lemma.

Lemma 8.3. *The problem* 2–IN–4SAT *is NP-complete for planar formulae without negations.*

Proof. Suppose we have a formula φ with clauses of size four without negations. We now show that if the formula φ is not planar we can alter it in polynomial time so that the resulting formula φ' is planar and φ is 2–in–4 satisfiable if and only if φ' is 2–in–4 satisfiable. The formula φ' will contain some negations but we also define a planar formula ensuring $v = \neg v'$ for two of its variables v, v' and all 2–in–4 satisfying truth assignments. Hence by substitution of each edge representing the negative occurrence by this gadget we prove the lemma.

Let d be a drawing of the incidence graph of φ in the plane, such that any two edges cross at most once. We proceed by induction on the number of crossings in d. If there is no crossing, we are done. Now suppose there is some crossing and for all formulae having a drawing of their incidence graph with less crossings

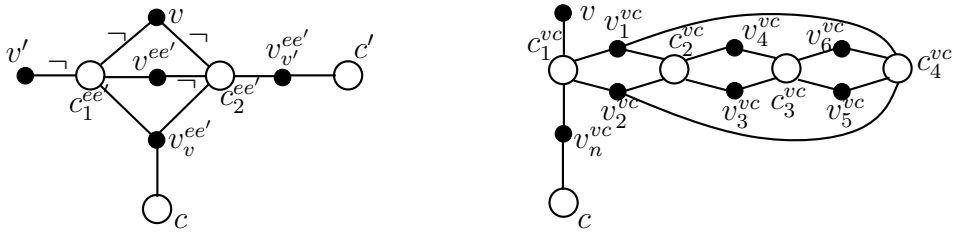


Figure 8.1: The crossing gadget for two edges $\{v, c\}$ and $\{v', c'\}$ (left) and the negation gadget for a negative occurrence of variable v in clause c (right). Empty circles represent clauses, and full circles represent variables. The symbol ¬ marks a negative occurrence.

the lemma holds. Consider an edge $e = (v, c)$ and the crossing with some edge $e' = (v', c')$ closest to v on the edge e. Create a new formula ψ by adding three new variables $v^{ee'}, v^{ee'}_v, v^{ee'}_{v'}$ and two clauses $c^{ee'}_1 := \neg v \vee \neg v' \vee v^{ee'} \vee v^{ee'}_v, c^{ee'}_2 := \neg v \vee \neg v^{ee'} \vee v^{ee'}_{v'} \vee v^{ee'}_v$. Then substitute occurrences of v in c by $v^{ee'}_v$, and occurrences of v' in c' by $v^{ee'}_{v'}$. See Figure 8.1 for an example of a gadget for two crossing edges.

After the substitution we clearly obtain a formula with a drawing with one less crossing. It remains to show that ψ is 2–in–4 satisfiable if and only if φ is 2–in–4 satisfiable (we get the rest using the induction). Let t be a 2–in–4 satisfying truth assignment for φ. Setting $t'(x) := t(x)$ for all variables x of φ, $t'(v^{ee'}_v) := t(v)$ and $t'(v^{ee'}_{v'}) = t'(v^{ee'}) := t(v')$, we obtain a 2–in–4 satisfying truth assignment for ψ. The other implication can be seen as follows. Let t' be a 2–in–4 satisfying truth assignment for ψ. We set $t(x) := t'(x)$ for each variable x of φ. If we show that $t'(v) = t'(v^{ee'}_v)$ and $t'(v') = t'(v^{ee'}_{v'})$, we immediately get that t is a 2–in–4 satisfying truth assignment for φ. We analyze two cases (the other two follow by symmetry):

- $t'(v) = t'(v') = 1$: In this case, $c^{ee'}_1$ has already two literals set to zero and so $t'(v^{ee'}_v) = t'(v^{ee'}) = 1$. Now looking at $c^{ee'}_2$ we see that two of its literals are set to zero and one literal is set to one. Thus $t'(v^{ee'}_{v'}) = 1$.

- $t'(v) = \neg t'(v') = 1$: If $t'(v^{ee'}) = 1$, then $c^{ee'}_1$ has two literals set one and one literal set to zero. Thus $t'(v^{ee'}_v)$ must be zero. But then $c^{ee'}_2$ has three literals set to zero and we can conclude that this cannot be the case. Hence $t'(v^{ee'}) = 0$ and $t'(v^{ee'}_v) = 1$ to satisfy $c^{ee'}_1$. Moreover $c^{ee'}_2$ has two literals set to one and one literal set to zero showing that $t'(v^{ee'}_{v'}) = 0$.

Now, it remains to show how to remove the negative occurrences from φ'. For each negative occurrence of variable v in clause c, we add seven new variables $v^{vc}_n, v^{vc}_1, \ldots, v^{vc}_6$ and four new clauses $c^{vc}_0 := v \vee v^{vc}_n \vee v^{vc}_1 \vee v^{vc}_2, c^{vc}_1 := v^{vc}_1 \vee v^{vc}_2 \vee v^{vc}_3 \vee v^{vc}_4, c^{vc}_2 := v^{vc}_3 \vee v^{vc}_4 \vee v^{vc}_5 \vee v^{vc}_6, c^{vc}_3 := v^{vc}_1 \vee v^{vc}_2 \vee v^{vc}_5 \vee v^{vc}_6$. See Figure 8.1 for an example of a created gadget. We also substitute the negative occurrence

of v in c by a positive occurrence of v_n^{vc}. Let φ'' be the resulting formula. It is straightforward to check that $t''(v) = \neg t''(v_n^{vc})$ in any 2–in–4 satisfying truth assignment t'' of φ'' and by setting $t''(v_1^{vc}) := t''(v_3^{vc}) := t''(v_5^{vc}) = t'(v)$ and $t''(v_2^{vc}) := t_4''^{vc} := t_6''^{vc} := t_n''^{vc} = \neg t'(v)$ we get a 2–in–4 satisfying truth assignment t'' of φ'' from a 2–in–4 satisfying truth assignment t' of φ'. □

Note that if we allowed multiple occurrences of one variable in a clause in the previous lemma the negation gadget would become trivial and for our purposes such a weaker lemma would be sufficient. But we decided to prove the stronger version as we find the lemma of independent interest.

8.3 NP-Hardness of Balanced Ordering Problems

In this section, we prove several NP-hardness results about balanced ordering problems.

Theorem 8.4. *The* PERFECT ORDERING *problem is NP-complete for planar graphs with maximum degree four.*

Proof. NP-hardness is proved by a reduction from 2–IN–4SAT for planar formulae without negations (see Lemma 8.3). Given a formula φ, create a graph G_φ with one vertex u_c for each clause c. For each variable v that occurs o_v times in φ, add a path on $3o_v + 1$ new vertices $p_1^v, \ldots, p_{3o_v+1}^v$ to G_φ, add o_v additional vertices $q_1^v, \ldots, q_{o_v}^v$, and connect $q_i^v, i \in \{1, \ldots, o_v\}$, with vertices p_{3i-2}^v and p_{3i}^v. The path with the additional vertices is called a *variable gadget*. Finally for each $i \in \{1, \ldots, o_v\}$, connect vertex p_{3i-2}^v of the path to u_c, where c is the clause corresponding to the i-th occurrence of the variable v. These edges are called *clause edges*. See Figure 8.2 for an example of this construction.

Observe that the maximum degree of G_φ is four. In particular, $\deg(u_c) = 4$, $\deg(q_i^v) = 2$ for all $i \in \{1, \ldots, o_v\}$, $\deg(p_{3i}^v) = 3$ for all $i \in \{1, \ldots, o_v\}$, $\deg(p_{3i-2}^v) = 4$ for all $i \in \{2, \ldots, o_v\}$, $\deg(p_{3i-1}^v) = 2$ for all $i \in \{1, \ldots, o_v\}$, $\deg(p_1^v) = 3$, and $\deg(p_{3o_v+1}^v) = 1$. Also note that the created graph is planar if the incidence graph of φ is planar.

We now prove that G_φ has a perfect ordering if and only if φ is 2–in–4 satisfiable. Suppose G_φ has a perfect linear ordering σ. For each variable v and for each $i \in \{1, \ldots, o_v\}$ the vertex p_{3i-1}^v has one neighbor to the left and one neighbor to the right in σ (since $\deg(p_{3i-1}^v) = 2$). Similarly, q_i^v has one neighbor to the left and one neighbor to the right in σ. Thus they must be placed between p_{3i-2}^v and p_{3i}^v. As p_{3i-1}^v and q_i^v are on one side (e.g., to the left) of vertex p_{3i-2}^v (p_{3i}^v) the other neighbors of the vertex must be on the other side. This implies that in σ, the path in each variable gadget is in the order given by its numbering or inverse numbering, and all the clause edges (the edges with exactly one endpoint

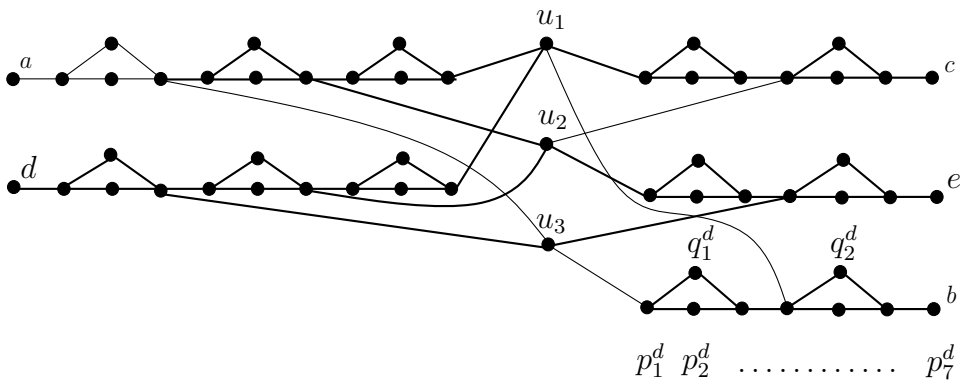


Figure 8.2: Constructed graph for formula $(a \vee b \vee c \vee d) \wedge (a \vee c \vee d \vee e) \wedge (a \vee b \vee d \vee e)$. The three clauses have numbers $1, 2, 3$ in the picture.

in the variable gadget) have a clause vertex on the same end (for example, the left end of each clause edge is a vertex of a path). If the path in the gadget for variable v is ordered according to its numbering, then set $t(v) := 0$. Otherwise set $t(v) := 1$. This truth assignment is 2–in–4 satisfying because each clause vertex has two neighbors on each side.

For a given truth assignment t, we can analogously construct a perfect linear ordering. First, place each variable gadget corresponding to a variable with $t(v) = 0$ with the path placed according to the inverse ordering, and put each vertex q_i^v immediately after vertex $p_{3i-1}^v, i \in \{1, \ldots, o_v\}$. Then place vertices u_c in an arbitrary order and finally the variable gadgets corresponding to variables with $t(v) = 1$ with the paths ordered according to the numbering and vertices q_i^v placed immediately after the vertex p_{3i-2}^v. □

The following two technical lemmas will be used later for removing parallel edges from a multigraph without changing an ordering with minimum imbalance.

Lemma 8.5. *Let G be the multigraph drawn in Figure 8.3 with two parallel edges added between the vertices a and b. Then there exists a minimum ordering of G such that a is the leftmost and b the rightmost vertex. Such an ordering is called* a natural ordering *of G.*

Proof. The ordering $a, 1, 2, 3, 4, 5, 6, b$ has imbalance 20. We claim that there is no ordering with smaller imbalance. Let $v_1, \ldots, v_8$ be some ordering of the vertices. We distinguish two cases:

1. There are two parallel edges between v_1 and v_2 (or symmetrically between v_7 and v_8). Because there is only one double-edge in our graph, $v_1 = a$ and $v_2 = b$ (the case $v_1 = b$ and $v_2 = a$ is the same) and we also know that there is at most one neighbor to the right of v_7. Since each vertex of G is connected to exactly one of a and b, there is only one neighbor to the left of v_3. Because

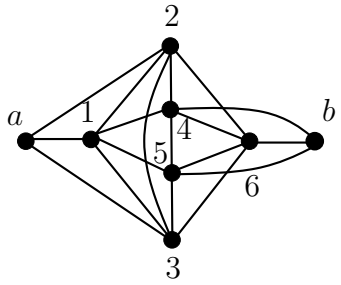


Figure 8.3: The triple edge gadget.

the other vertices have imbalance at least one (they have odd degrees), the imbalance of the ordering is at least $B(v_1) + B(v_3) + B(v_7) + B(v_8) + 4 = 5 + 3 + 3 + 5 + 4 = 20$.

2. There is no parallel edge between v_1 and v_2 nor between v_7 and v_8. In this case, there is at most one neighbor to the left of v_2, and at most one neighbor to the right of v_7. Hence the total imbalance is at least $B(v_1) + B(v_2) + B(v_7) + B(v_8) + 4 = 5 + 3 + 3 + 5 + 4 = 20$.

□

Lemma 8.6. *Let G be a 5-regular multigraph and let c be the number of triple-edges in G. Let G' be the graph obtained from G by replacing each triple-edge of G with endpoints a and b by the* triple-edge gadget *in Figure 8.3. The vertices a and b of the gadget are identified with the original end-vertices of the triple-edge. Then $M(G) = M(G') - 10 \cdot c$.*

Proof. Given an ordering of G with imbalance i, we can create an ordering of G' with imbalance $i + 10 \cdot c$ by inserting, for each triple-edge ab, the vertices $1, \ldots, 6$ from the gadget in Figure 8.3 between a and b and in this order (the imbalance of a and b does not change by the substitution). Thus $M(G) \geq M(G') - 10 \cdot c$. On the other hand, if we have a minimum ordering of G' with imbalance i', below we show that by changing the given ordering so that each gadget is in its natural ordering, we obtain an ordering with imbalance $\leq i'$. Hence, the new ordering has imbalance i', from the minimality of i'. By substituting each gadget with the triple-edge we obtain an ordering of G with imbalance $i' - 10 \cdot c$, proving that $M(G) \leq M(G') - 10 \cdot c$.

Suppose we have a triple-edge gadget in G' between vertices a and b, with a to the left of b. Each of the vertices has two neighbors u_1, u_2 (u'_1, u'_2 respectively) outside of the gadget. Let u_1 be to the left of u_2, and let u'_1 be to the left of u'_2. If u_1, u_2 are both to the right of a and u'_1, u'_2 are both to the left of b, then we are in the situation described by Lemma 8.5, and we can conclude that

after reordering the vertices of the gadget (so that vertices a and b retain their ordering with respect to all other vertices of the graph), we obtain an ordering with less or equal imbalance. If u_1 is to the left and u_2 to the right of a (u'_1 and u'_2 are both still to the left of b), then the imbalance of the gadget in the natural ordering in this situation is 18 and any ordering of the gadget cannot have a smaller imbalance, as in the new situation only the order of vertices of one of the edges incident with the gadget has changed. Hence, if we reorder the vertices of the gadget, we obtain an ordering of the graph with less or equal imbalance. Now consider the general situation. We have j neighbors of a outside of the gadget and j' neighbors of b outside of the gadget ($0 \leq j, j' \leq 2$) on the other side of a (b respectively). In the original situation the natural ordering of the gadget has imbalance $20 - 2 \cdot (j + j')$. There cannot be an ordering with smaller imbalance because in the situation only the order of vertices of the $j + j'$ edges incident with the gadget changed. Hence, we can conclude that reordering the gadget into the natural ordering cannot increase the imbalance of the graph regardless of the ordering of v, u_1, u_2 and v', u'_1, u'_2. □

For the next reduction, we use the 2–OR–3–IN-5SAT problem, which we proved to be NP-complete in Section 8.2.

Theorem 8.7. *The* PERFECT ORDERING *problem is NP-complete for 5-regular multigraphs.*

Proof. We prove NP-hardness by a reduction from 2–OR–3–IN-5SAT. Suppose that we are given a formula φ without negations and with all clauses of size five. Moreover, assume that each variable occurs in at least two different clauses in the formula. We can make a formula satisfy this condition by adding satisfied clauses of type $x \vee x \vee x \vee \neg x \vee \neg x$. Now create the following multigraph G from φ. For each clause c, add a new vertex v_c to G. For each variable x that occurs o_x times in φ, add a new path (called a *variable path*) with $2o_x - 2$ vertices $v^x_1, \ldots, v^x_{2o_x-2}$, where edges $v^x_{2i-1}v^x_{2i}, 1 \leq i \leq o_x - 1$, are triple-edges. Connect vertex $v^x_{2i}, 1 \leq i \leq o_x - 1$, of the path to the vertex corresponding to the clause with i-th occurrence of x. Furthermore, connect vertex $v^x_{2o_x-2}$ to the vertex corresponding to the clause with the o_x-th occurrence of x (because x was in at least two different clauses, we can without loss of generality assume that no parallel edges are created). Connect each vertex $v^x_{2i-1}, 1 \leq i \leq o_x - 1$, to the new vertex p^x_i, and connect each vertex v^x_1 to the new vertex p^x_0. Now, the only vertices which have degree other than five are in the set $P := \{p^x_j : x \text{ is a variable}, 0 \leq j \leq o_x - 1\}$ and these have degree one. By running the following procedure two times for the set P, all the vertices will have degree five.

$n := |P|$
Arbitrarily number the vertices in P by $1, \ldots, n$.
`while` $|P| \geq 3$ `do`
  Let u_i, u_j, u_k be three vertices in P.

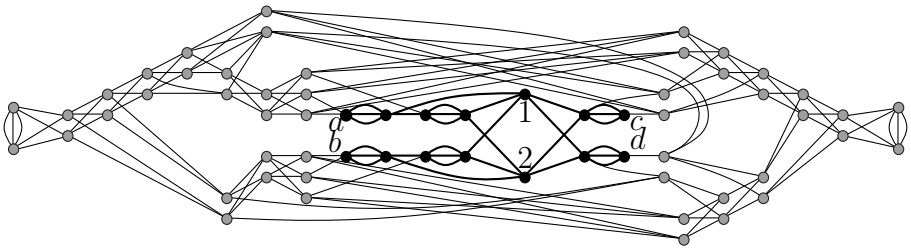


Figure 8.4: Constructed graph for formula $(a \vee a \vee b \vee c \vee d) \wedge (a \vee b \vee b \vee c \vee d)$. Clause vertices are marked 1 and 2. Clause vertices and variable paths are drawn in black color, vertices p_i^x and vertices added by the procedure are in gray color.

$P := P \setminus \{u_i, u_j, u_k\} \cup \{u_{n+1}, u_{n+2}\}$
Add a complete bipartite graph on u_i, u_j, u_k and u_{n+1}, u_{n+2} to G.
$n := n + 2$
`end`
`//` *Now* $P = \{u_i, u_j\}$
Add to G a complete bipartite graph on u_i, u_j and two new vertices s_1, s_2.
Add a triple-edge $s_1 s_2$ to G.

Let n_0 denote the value of n at the beginning of the procedure and n_1 the value of n at the end of the procedure. It is easy to check that G is 5-regular. We now show that G has a perfect ordering if and only if φ was 2-or-3-in-5 satisfiable. Suppose we have a perfect ordering of G. It holds for every ordering that $B(s_1) + B(s_2) > 2$. Since the ordering is perfect, the ordering begins s_1, s_2 without loss of generality. By a similar argument, the ordering ends with vertices s_2', s_1', where s_1' and s_2' are the vertices added at the end of the second run of the procedure on P. Because all other vertices are balanced, we know that every variable path is either in its natural ordering or reversed. Moreover all edges between the variable path and clauses have clause vertices to the right (or to the left in the reversed case). Because all clause vertices are balanced we get a 2-or-3-in-5 satisfying truth assignment of φ by assigning $t(x) := 0$ to the variables whose path is naturally ordered and $t(x) := 1$ to the variables whose path is reversed. For the converse, suppose we have a 2-or-3-in-5 satisfying truth assignment t of φ. First, we place vertices $s_1, s_2, u_{n_1}, \ldots, u_{n_0+1}$ added in the first run. We continue by placing vertices p_j^x where x is a variable with $t(x) = 0$ and $0 \leq j \leq o_x - 1$. Then we place variable paths for variables x such that $t(x) = 0$ in their natural ordering and after them the clause vertices. We finish by placing symmetrically the rest of the paths and vertices added in the second run. It is straightforward to check that this ordering is perfect. □

Corollary 8.8. *It is NP-hard to find a minimum ordering for 5-regular graphs.*

Proof. Construct the multigraph G as in the reduction in the proof of Theorem 8.7. Say G has c triple edges. Construct G' from G by substituting each triple-edge by a triple-edge gadget. Observe that G' remains 5-regular and is a simple graph. From Lemma 8.6 we know that orderings of G' with imbalance $|V| + 10 \cdot c$ correspond to perfect orderings of G. This proves NP-hardness of finding the ordering with such imbalance. Hence finding a minimum ordering for 5-regular graphs is NP-hard. □

8.4 Algorithm

In this section, we present an algorithm that determines in polynomial time whether a given multigraph has an ordering with constant imbalance. First, we introduce a key lemma.

Lemma 8.9. *There is an $O(n+m)$ time algorithm to test whether a multigraph G with n vertices and m edges has an ordering in which a given list of vertices imbalanced $= (v_1, \ldots, v_k)$ are the only imbalanced vertices, and $\sigma(v_i) < \sigma(v_{i+1})$ for all $1 \leq i \leq k-1$.*

Proof. The vertices not in the list *imbalanced* are called *balanced*. The algorithm works as follows: First, we check that all odd-degree vertices are in the *imbalanced* list. If not, then we can reject since every odd-degree vertex must be imbalanced. Now assume that all balanced vertices have even degrees. Then start building an ordering σ from left to right. We append to σ those vertices that have not been placed yet and have half of their neighbors already placed. Such vertices are called *saturated* and are stored in the set *saturated*. Because saturated vertices are balanced each saturated vertex must be placed before any of its unplaced neighbors. In particular, saturated vertices must form an independent set. Hence, we cannot make a mistake when placing any saturated vertices. If there is no saturated vertex, the vertex which is placed next will be imbalanced and hence it must be the first unused vertex from the *imbalanced* list. It remains to prove that it is not better to place some vertices from the *imbalanced* list while there are still some saturated vertices. If the order of vertices of any edge does not change then we have an equivalent ordering. Otherwise it does change, in which case some balanced vertex becomes imbalanced (as the order of vertices in an edge can change only for the edges which contain at least one balanced vertex) and we must reject. □

The following theorem is a consequence of Lemma 8.9.

Theorem 8.10. *There is an algorithm that, given an n-vertex m-edge multigraph G, computes a minimum ordering of G with at most k imbalanced vertices (or answers that there is no such ordering) in time $O(n^k \cdot (m+n))$.*

Proof. The algorithm is simple: just try all the possible choices of k imbalanced vertices and their orderings. For each such choice run the procedure from Lemma 8.9 and select the ordering with minimum imbalance from those orderings. There are $O(n^k)$ k-tuples of imbalanced vertices, and for each such k-tuple, by Lemma 8.9, we can check in $O(m+n)$ time whether there is an ordering with the chosen vertices imbalanced, and if so, compute the imbalance of the ordering. □

Corollary 8.11. *There is a polynomial time algorithm to determine whether a given multigraph G has an ordering with imbalance less than a fixed constant c.*

Proof. Apply the algorithm from Theorem 8.10 with $k = c - 1$. If the algorithm rejects the multigraph or produces an ordering with imbalance greater than c, then the graph does not have an ordering with imbalance less than c (because any ordering with imbalance less than c must have at most $c-1$ imbalanced vertices). Otherwise the algorithm outputs some ordering with imbalance less than c, and we are done. □

Corollary 8.12. *The* PERFECT ORDERING *problem is solvable in time $O(n^2(n+m))$ for any n-vertex m-edge multigraph with all vertices of even degree.*

Proof. Apply the algorithm from Theorem 8.10 with $k = 2$, and then check whether the achieved imbalance is equal to that required by the PERFECT ORDERING problem. A perfect ordering of a multigraph with even degrees must have exactly two imbalanced vertices (assuming there is at least one edge). □

8.5 Conclusion and Open Problems

In this chapter, we have considered the problems of checking the existence of a perfect ordering for planar graphs with maximum degree four and 5-regular multigraphs. Both these problems were shown to be NP-complete, thus answering a number of questions raised by Biedl *et al.* [8]. We have also established that it is NP-hard to find an ordering with minimum imbalance for 5-regular simple graphs. We have also introduced an algorithm for determining an ordering with imbalance smaller than k running in time $O(n^k(n+m))$. It would be interesting to obtain a fixed-parameter-tractable (FPT) algorithm for this problem (as one cannot hope for a polynomial solution unless P=NP).

Bibliography

[1] J. F. Allen, Maintaining knowledge about temporal intervals, *Communications of the ACM* **26(11)**, 1983, 832–843.

[2] J. F. Allen, *Natural Language Understanding*, Benjamin Cummings, 1994.

[3] D. M. Arnold and K. M. Rangaswamy (Eds.), *Abelian Groups and Modules*, Dekker, New York, 1996.

[4] Christer Bäckström and Peter Jonsson, A Unifying Approach to Temporal Constraint Reasoning, *Artif. Intell.* **102(1)** (1998), 143–155.

[5] J. Bang-Jensen, B. Reed, M. Schacht, R. Šámal, B. Toft, U. Wagner, On six problems posed by Jarik Nešetřil, in: *Topics in Discrete Mathematics* (Dedicated to Jarik Nešetřil on the occasion of his 60th birthday), M. Klazar, J. Kratochvíl, M. Loebl, J. Matoušek, P. Valtr, R. Thomas, eds., *Algorithms and Combinatorics* **26** (2006), Springer, 613–627.

[6] G. di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing*, Prentice-Hall, 1999.

[7] P. van Beek, Reasoning about qualitative temporal information, *Artificial Intelligence* **58** (1992), 297–326.

[8] T. C. Biedl, T. Chan, Y. Ganjali, M. TaghiHajiaghayi, and D. R. Wood, Balanced vertex-orderings of graphs, *Discrete Applied Mathematics* **148(1)** (2005), 27–48.

[9] T. C. Biedl and G. Kant, A better heuristic for orthogonal graph drawings, *Comput. Geom.* **9(3)** (1998), 159–180.

[10] T. C. Biedl and M. Kaufmann, Area efficient static and incremental graph drawings, In *Proc. 5th Ann. European Symp. on Algorithms (ESA '97)*, *LNCS* **1284** (1997), 37–52.

[11] M. Bodirsky, *Constraint satisfaction with infinite domains*, PhD thesis, Humboldt Universität zu Berlin, 2004.

[12] M. Bodirsky, Cores of Countably Categorical Structures, to appear in *Logical Methods of Computer Science*.

[13] M. Bodirsky and V. Dalmau, Datalog and Constraint Satisfaction with Infinite Templates, *Proceedings of the 23rd International Symposium on Theoretical Aspects of Computer Science (STACS'06)*, *LNCS* **3884** (2006), 646–659.

[14] M. Bodirsky, C. Dangelmayr, and J. Kára, Representing Series-parallel Graphs as Intersection Graphs of Line Segments in Three Directions, In: *Innovative Applications of Information Technology for Developing World*, Kathmandu, Nepal, 2006.

[15] M. Bodirsky and J. Kára, The complexity of equality constraint languages, *Proceedings of the International Computer Science Symposium in Russia (CSR'06)*, *LNCS* **3967** (2006), 114–126.

[16] M. Bodirsky and J. Kára, The Complexity of Equality Constraint Languages, journal version, submitted.

[17] M. Bodirsky and J. Kára, A Fast Algorithm and Lower Bound for Temporal Reasoning, submitted.

[18] M. Bodirsky and J. Nešetřil, Constraint satisfaction with countable homogeneous templates, *J. of Logic and Comput.* **16(3)** (2006), 359–373.

[19] A. Brandstädt, V. Bang Le, and J. P. Spinrad, *Graph Classes: A Survey*, SIAM Monographs on Discrete Mathematics and Applications, 2000.

[20] Andrei A. Bulatov, A dichotomy theorem for constraint satisfaction problems on a 3-element set, *J. of ACM* **53** (2006), 66–120.

[21] A. Bulatov, A. Krokhin, and P. Jeavons, Classifying complexity of constraints using finite algebras, *SIAM J. Comput.* **34(3)** (2005), 720–742.

[22] H.-J. Bürckert and B. Nebel, Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra, *J. of the ACM* **42(1)** (1995), 43–66.

[23] P. J. Cameron, *Oligomorphic Permutation Groups*, Cambridge University Press, 1990.

[24] N. de Castro, F. J. Cobos, J. C. Dana, and A. Márquez. Triangle-Free Planar Graphs as Segment Intersection Graphs, *J. of Graph Algorithms and Applications* **6** (2002), 7–26.

[25] J. Chalopin, D. Gonçalves, and P. Ochem, *Planar Graphs are in 1-STRING*, to appear in Proceedings of SODA'07.

[26] D. Cohen, P. Jeavons, P. Jonsson, and M. Koubarakis, Building tractable disjunctive constraints, *J. of the ACM* **47(5)** (2000), 826–853.

[27] N. Creignou, M. Hermann, A. Krokhin, and G. Salzer, *Complexity of Clausal Constraints Over Chains*, Research Report, Ecole Polytechnique, 2005.

[28] N. Creignou, S. Khanna, and M. Sudan, *Complexity Classifications of Boolean Constraint Satisfaction Problems*, SIAM Monographs on Discrete Math. and Appl. 7, 2001.

[29] V. Dalmau, A new tractable class of constraint satisfaction problems, *Ann. Math. Artif. Intell.*, accepted.

[30] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.

[31] R. Dechter and P. van Beek, Local and global relational consistency, *TCS* **173(1)** (1997), 283–308.

[32] R. Diestel, *Graph Theory*, Springer-Verlag, New York, 1997.

[33] W. Dowling and J. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *J. of Logic Programming* **1(3)** (1984), 267–284.

[34] R. G. Downey, M. R. Fellows, *Parameterized Complexity*, Springer Verlag, 1999.

[35] T. Drakengren and P. Jonsson, Twenty-One Large Tractable Subclasses of Allen's Algebra, *Artificial Intelligence* **93**, 1997, 297–319.

[36] I. Düntsch, Relation algebras and their application in temporal and spatial reasoning, *Artificial Intelligence Review* **23**, 2005, 315–357.

[37] N. W. Dunkin, J. E. Bater, P. G. Jeavons, and D. A. Cohen, Towards high order constraint representations for the frequency assignment problem, Technical Report CSD-TR-98-05, Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, UK, 1998.

[38] H.-D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer, Second edition, 1999.

[39] I. Fary, On Straight Lines Representation of Planar Graphs, *Acta Sci. Math. Szeged* **11** (1948), 229–233.

[40] T. Feder and M. Y. Vardi, The computational structure of monotone monadic SNP and constraint satisfaction: A study through Datalog and group theory, *SIAM J. Comput.* **28** (1998), 57–104.

[41] M. Fisher, D. Gabbay, and L. Vila *Handbook of Temporal Reasoning in Artificial Intelligence*, Elsevier, 2005.

[42] H. de Fraysseix and P. Ossona de Mendez, Contact and Intersection Representations, GD 2004, *LNCS* **3383** (2005), 217–227.

[43] H. de Fraysseix, P. Ossona de Mendez, and J. Pach, Representation of planar graphs by segments, In *Colloquia Mathematica Societatis János Bolyai, Intuitive Geometry*, Szeged (Hungary), 1991.

[44] H. de Fraysseix, J. Pach, and R. Pollack, How to Draw a Planar Graph on a Grid, *Combinatorica* **10** (1990), 41–51.

[45] M. R. Garey, D. S. Johnson, *Computers and intractability: a guide to NP-completeness*, Freeman, San Francisco, California, 1979.

[46] A. Garg and R. Tamassia, On the Computational Complexity of Upward and Rectilinear Planarity Testing, In *Graph Drawing (Proc. GD '94)*, *LNCS* **894** (1995), 286–297.

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

[48] G. Gottlob, L. Leone, and F. Scarcello, Hypertree decomposition and tractable queries, *J. Computat. System Sci.* **64** (3) (2002), 579–627.

[49] R. L. Graham, B. L. Rothschild, J. H. Spencer, *Ramsey Theory*, Wiley-Interscience, 1990.

[50] M. Grohe, The complexity of homomorphism and constraint satisfaction problems seen from the other side, *Foundations of Computer Science (Boston, MA, 2003)*, IEEE Comput. Soc., 2003, 552–561.

[51] G. Hajós, Über eine Art von Graphen, *International Math. Nachrichtung* **11** (1957), Problem 65.

[52] I. Ben-Arroyo Hartman, I. Newman, and R. Ziv. On grid intersection graphs, *Discrete Math.* **87** (1991), 41–52.

[53] P. Hell and J. Nešetřil, *Graphs and Homomorphisms*, Oxford University Press, 2004.

[54] P. Hell and J. Nešetřil, On the complexity of H-coloring, *J. Combin. Theory Ser. B* **48** (1990), 92–110.

[55] W. Hodges, *A shorter model theory*, Cambridge University Press, Cambridge, 1997.

[56] S. Janson, T. Luczak, and A. Rucinski, *Random Graphs*, John Wiley and Sons, New York, 2000.

[57] P. Jeavons, P. Jonsson, and A. Krokhin, Reasoning About Temporal Relations: The Tractable Subalgebras of Allen's Interval Algebra, J. of ACM **50(5)**, 2003, 591–640.

[58] P. G. Jeavons, On the algebraic structure of combinatorial problems, *Theoret. Comput. Sci.* **200** (1998), 185–204.

[59] M. Junker and M. Ziegler, *The 116 reducts of (Q,¡,a)*, MODNET preprints **19** (2006).

[60] G. Kant, *Algorithms for Drawing Planar Graphs*, University of Utrecht.

[61] G. Kant, Drawing planar graphs using the canonical ordering, *Algorithmica* **16** (1996), 4–32.

[62] G. Kant and X. He, Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems, *Theoretical Computer Science* **172(1–2)** (1997), 175–193.

[63] J. Kára and J. Kratochvíl, Fixed Parameter Tractability of Independent Set in Segment Intersection Graphs, In *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006*, *LNCS* **4169** (2006), 166–174.

[64] J. Kára, J. Kratochvíl, and D. R. Wood, On the complexity of the balanced vertex ordering problem, *Proc. of 11th Annual International Conference, COCOON 2005*, *LNCS* **3595** (2005), 849–858.

[65] J. Kára, J. Kratochvíl, and D. R. Wood, On the complexity of the balanced vertex ordering problem (extended journal version), submitted.

[66] H. Kautz, P. van Beek, and M. Vilain, Constraint propagation algorithms: A revised report, *Qualitative Reasoning about Physical Systems*, 1990, 373–381.

[67] H. Kautz and B. Selman, Planning as satisfiability, *Tenth European Conference on Artificial Intelligence (Vienna, 1992)*, John Wiley and Sons, Chichester, 1992, 359–363.

[68] Ph. G. Kolaitis and M. Y. Vardi, Conjunctive-query containment and constraint satisfaction, *J. Comput. System Sci.* **61** (2000), 302–332.

[69] M. Koubarakis, Tractable Disjunctions of Linear Constraints: Basic Results and Applications to Temporal Reasoning, *Theoretical Computer Science* **266**, 2001, 311–339.

[70] J. Kratochvíl, A special planar satisfiability problem and a consequence of its NP–completeness, *Discrete Appl. Math.* **52** (1994), 233–252.

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

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

[73] J. Kratochvíl, J. Nešetřil, INDEPENDENT SET and CLIQUE problems in intersection defined classes of graphs, *Comment. Math. Univ. Carolin.* **31** (1990), 85–93.

[74] A. Krokhin, A. Bulatov, P. Jeavons, The complexity of constraint satisfaction: An algebraic approach, *Proceedings of the NATO Advanced Study Institute on Structural Theory of Automata, Semigroups and Universal Algebra, NATO Science Series II: Mathematics, Physics and Chemistry* **207** (2005), 181–213.

[75] P. B. Ladkin and R. D. Maddux, On binary constraint problems, *J. of ACM* **41(3)**, 1994, 435–469.

[76] D. Marker, *Model Theory: An Introduction*, Springer, New York, 2002.

[77] D. Marx, Parametrized Complexity of Independence and Domination on Geometric Graphs, In *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006*, *LNCS* **4169** (2006), 154–165.

[78] T. A. McKee and F. R. McMorris, *Interesection Graph Theory*, SIAM Monographs on Discrete Mathematics and Applications, 1999.

[79] R. van der Meyden, The Complexity of Querying Indefinite Information about Linearly Ordered Domains, *J. of Computer and Systems Science* **54(1)**, 1997, 113–135.

[80] U. Montanari, Networks of constraints: Fundamental properties and applications to picture processing, *Inform. Sci* **7** (1974), 95–132.

[81] B. A. Nadel, Constraint satisfaction in Prolog: complexity and theory-based heuristics, *Inform. Sci* **83** (3–4) (1995), 113–131.

[82] J. Nešetřil and V. Rödl, *Mathematics of Ramsey Theory*, Springer, Berlin, 1998.

[83] R. Niedermeier, *Invitation to Fixed-Parameter Algorithms*, Oxford University Press, 2006.

[84] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[85] A. Papakostas and I. G. Tollis, Algorithms for area-efficient orthogonal drawings, *Computational Geometry: Theory and Applications* **9** (1998), 83–110.

[86] A. Papakostas and I. G. Tollis, Algorithms for incremental orthogonal graph drawing in three dimensions, *J. Graph Algorithms Appl.* **3(4)** (1999), 81–115.

[87] R. Pöschel and L. A. Kalužnin, *Funkcionen- und Relationenalgebren*, DVW, Berlin, 1979.

[88] F. S. Roberts, Indifference graphs, *Proof Techniques in Graph Theory*, F.Harary, ed., Academic Press, New York 1969, 139–146.

[89] N. Robertson and P. D. Seymour, Graph Minors I. Excluding a forest, *J. Combinatoral Theory Series B* **35** (1983), 39–61.

[90] I. G. Rosenberg, Minimal clones I: the five types, *Lectures in Universal Algebra (Proc. Conf. Szeged 1983)*, Colloq. Math. Soc. Janos Bolyasi **43** (1986), 405–427.

[91] T. J. Schaefer, The complexity of satisfiability problems, *Tenth ACM Symposium on Theory of Computing (San Diego, CA, 1978)*, ACM Press, New York, 1978, 216–226.

[92] M. Schaefer, E. Sedgwick, and M. Štefankovic, Recognizing string graphs in NP, *Journal of Computer and System Sciences* **67** (2003), no. 2, 365–380.

[93] E. R. Scheinerman, *Intersection classes and multiple intersection parameters of graphs*, PhD thesis, Princeton University, 1984.

[94] W. Schnyder, Embedding Planar Graphs on the Grid, In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms* (1990), 138–148.

[95] E. Schwalb and L. Vila, Temporal constraints: a survey, *Constraints* **3** (2–3) (1998), 129–149.

[96] J. P. Spinrad, *Representations of graphs*, book manuscript, Vanderbilt University, 1997.

[97] S. K. Stein, Convex Maps, *Proc. Amer. Math. Soc.* **2** (1951), 464–466.

[98] A. Szendrei, Clones in Universal Algebra, *Seminaries de Mathematiques Superieures* **99**, University of Montreal (1986).

[99] R. Tamassia, On Embedding a Graph in the Grid with the minimum Number of Bends, *SIAM J. Comput.* **16 (3)** (1987), 421–444.

[100] K. Wagner, Bemerkungen zum Vierfarbenproblem, **Jahresbericht der Deutschen Mathematiker-Vereinigung 46** (1936), 26–32.

[101] D. R. Wood, Minimizing the number of bends and volume in three-dimensional orthogonal graph drawings with a diagonal vertex layout, *Algorithmica* **39** (2004), 235–253.

[102] D. R. Wood, Optimal three-dimensional orthogonal graph drawing in the general position model, *Theoretical Computer Science* **299(1–3)** (2003), 151–178.