The Effect of Representations on Constraint Satisfaction Problems

PhD Thesis

Chris Houghton
Department Of Computer Science
Royal Holloway, University Of London

Supervised by Prof. David Cohen

Declaration of Authorship

I Christopher James Houghton hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

Date:

Abstract

Constraint Satisfaction is used in the solution of a wide variety of important problems such as frequency assignment, code analysis, and scheduling. It is apparent that the modelling process is key to the success of any constraint based technique, and much work has been done on the identification of good models [FJHM05].

One of the key choices made during the modelling process is the selection of a constraint representation with which to express the constraints [HS02]. Whilst practitioners will commonly use an implicit representation, most existing structural tractability results are defined for explicit representations. We address a well-known anomaly in structural tractability theory, that acyclic instances are tractable when expressed explicitly, but may not be when expressed implicitly, and show that there is a link between representation and tractability.

We introduce the notion of *interaction width* in order to address this disconnect between theory and practice, and use this to define new tractable classes by applying existing structural tractability results to different constraint representations. We show that for a given succinct representation, a non-trivial class of instances with bounded interaction width can be transformed into an explicit representation in polynomial time so that existing structural tractability results may be applied. We compare our work to existing results for alternative succinct representations and show that the tractable classes we have defined are incomparable and novel, and can be used to derive new tractable classes for SAT.

Acknowledgements

I would like the thank my supervisor Prof. David Cohen for his help, direction and perseverance during the production of this thesis. I would also like to thank my good friend Dr. Martin Green for providing me with much needed help and motivation along the way.

I am grateful for the support and gentle pressure from my wife Annabel, without whom this thesis would probably have never been completed.

There are also many friends, relatives and colleagues who deserve a mention for providing invaluable assistance during the production of this thesis. There are too many of you to list without risking offence by leaving someone out - hopefully you all know who you are!

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Motivation

The constraint satisfaction problem is, in general, NP-hard. However, certain classes of problem instances have known structural or language based properties which can be identified, and which allow us to solve them efficiently. Much research has been concerned with identifying these properties and finding efficient algorithms to exploit them.

We have observed that the way in which problems are represented in the theoretical world, and the way in which the same problems are represented in practice, are not always the same. More specifically, problems in the theoretical world tend to be expressed explicitly, as opposed to implicitly in practice. This limits the practical applicability of most theoretical tractability results.

In any categorisation for classes of constraint satisfaction problem instances, into tractable and intractable, equivalent instances should be regarded as having the same complexity. For example, the renaming of variables or domain elements should not change the time required to solve an instance. However, there are anomalies within the current results: many properties which lead to efficient solution in the theoretical world do not hold when using implicit constraint representations. What is needed is a unification of theory and practice so that theoretical tractability results are generated for a wide range of practical constraint representations.

Research Achievements

The purpose of this thesis is to help bridge the gap between theory and practice. Of particular importance is the use of real world (succinct) constraint representations in the theoretical world, so that tractability results may still be applied when more practical constraint representations are used. We consider a succinct representation, called *Mixed*, which allows constraints to be represented by explicitly listing either the allowed assignments, or the disallowed assignments.

We define a new measure of structural width, called *interaction width*, which considers the level of constraint interaction within an instance. We show that although bounded interaction width alone does not of itself define structurally tractable classes, it allows the application of existing structural tractability results to more succinct constraint representations. We do this by constructing the algorithms necessary to convert an instance given in the mixed representation to the explicit theoretical representation, and performing a detailed complexity analysis of these algorithms in order to prove that this conversion can be performed in polynomial time for the class of instances with bounded interaction width.

We observed that there was no widely used framework for performing complexity analysis of algorithms, with much prior work concerned with optimisation rather than establishing membership of a complexity class. We construct such a framework based on the computational model of a Random Access Machine [CRR72] that provides data structures for which the complexity of certain operations is well defined.

By defining interaction width for relational structures, we are able to extend a result of Grohe [Gro07] to give a dichotomy result for the tractability of the mixed representation under bounded interaction width.

To show that we have defined novel structurally tractable classes using this technique, we compare our result with that of Chen and Grohe for another succinct representation, GDNF [CG06], and find them to be incomparable. We consider further the relative succinctness of these representations, and demonstrate that the mixed representation is strictly more structurally tractable than the GDNF representation.

As the mixed representation naturally expresses SAT problems, our result defines structurally tractable classes for SAT. We show that our result is also incomparable to that of Szeider [Sze03] whose result uses incidence width to define tractable SAT classes.

Overview of Thesis

- Chapter 2 is intended to familiarise the reader with the constraint satisfaction paradigm. We provide the constraint definitions used in this thesis, and discuss how problems may be modelled as CSPs. A brief survey of pre-processing and solutions techniques is also provided.
- Chapter 3 provides a survey of theoretical tractability results. Emphasis is placed on the structural definitions and results for tractability as these are used throughout the remainder of this thesis and form the basis for the main contributions.
- Chapter 4 describes the framework we have developed for the complexity analysis of algorithms. This framework will be used extensively for the proof of results in Chapter 5.
- Chapter 5 contains the main results of this thesis. We present the notion of *interaction width* for relational structures and use this to derive new tractability results for a more succinct representation of constraint satisfaction problems. We do this by developing the algorithms necessary to convert instances given in this succinct representation into the explicit representation used by theoretical tractability results. We then provide a detailed complexity analysis to prove that these algorithms run in polynomial time for classes with bounded interaction width. We demonstrate that our results are novel by showing them to be incomparable with existing results for other succinct representations.
- Chapter 6 discusses the conclusions drawn from our results and provides direction on how these may be extended.

Related Papers

The Effect of Constraint Representation on Structural Tractability [HCG06]

Principles and Practice of Constraint Programming (CP 2006) - Proceedings of the 12^{th} International Conference, Nantes, France, 25^{th}-29^{th} September 2006. Pages 726-730.

A short paper in which interaction width was introduced and defined on hypergraphs. An algorithm to convert an instance in the mixed representation to the positive representation is described that would preserve existing hypergraph tractability results and run in polynomial time for classes of CSP with bounded interaction width.

Constraint Representations and Structural Tractability [CGH09]

Principles and Practice of Constraint Programming (CP 2009) - Proceedings of the 15^{th} International Conference, Lisbon, Portugal, 20^{th}-24^{th} September 2009. Pages 289-303.

A full paper in which interaction width is redefined on relational structures in order to extend the results from the previously presented hypergraph definition. The dichotomy result for the tractability of the mixed representation under bounded interaction width is given along with an overview of the necessary proof explaining how the necessary algorithms would work. However, neither the algorithms, nor their analyses, are provided.

Chapter 2

Constraint Satisfaction

The aim of this chapter is to introduce the reader to the constraint satisfaction paradigm, and to give an overview of the core techniques used within the Constraint Satisfaction community to solve problem instances.

We start by discussing how certain types of real world problems can be naturally modelled as constraint satisfaction problems. We demonstrate that a given problem can be modelled using different sets of constraints, and that the constraints themselves can be represented in a variety of ways.

We provide an overview of common solution techniques for Constraint Satisfaction Problems, concentrating on those based on backtrack search. This will include some commonly used improvements to chronological backtrack such as algorithm enhancements, variable ordering heuristics and consistency processing, both before and during search.

2.1 The Constraint Satisfaction Problem

The Constraint Satisfaction paradigm [Mon74, Tsa93, Dec03, vHK06] is a system for modelling real life problems that views the world as a set of questions to be answered. Each of these questions requires an answer, or value, to be assigned to it and so are referred to as *variables*. The set containing all possible values that may be assigned to the variables is called the *domain*.

Definition 2.1.1. *For any set of variables X and set of values D, we call a mapping $a : X \to D$ an **assignment** to X.*

A *constraint* is a rule which limits the set of allowed assignments to a set of variables.

Definition 2.1.2. *A **constraint** is a pair $\langle \sigma, \rho \rangle$ where σ is a set of variables called the **scope** and ρ is a set of assignments called the **relation**.*

For ease of notation, the scope and relation of a constraint, c, may be referred to as $\sigma(c)$ and $\rho(c)$.

The restricted effect of a constraint over a subset of its scope is called a *projection* of a constraint.

Definition 2.1.3. *The **projection** of a constraint, $c = \langle \sigma, \rho \rangle$, onto a set of variables $X \subseteq \sigma$, denoted $\Pi_X c$, is $\langle X, \rho' \rangle$ where $\rho' = \{ z_{|X} \mid z \in \rho \}$.*

Similarly, the combined effect of multiple constraints is achieved by *joining* these constraints. This is a form of *constraint synthesis* [Fre78, YAAP03].

Definition 2.1.4. *Given two constraints, $c_0 = \langle \sigma_0, \rho_0 \rangle$ and $c_1 = \langle \sigma_1, \rho_1 \rangle$, the **join** of c_0 and c_1, denoted $c_0 \bowtie c_1$, is the constraint $\langle \sigma_0 \cup \sigma_1, \rho' \rangle$ where $\rho' = \{ z \mid z_{|\sigma_0} \in \rho_0 \wedge z_{|\sigma_1} \in \rho_1 \}$.*

A set of variables, the domain, and a set of constraints over subsets of the variables is called an *instance* [CJG08] of the *Constraint Satisfaction Problem*, a constraint network [Dec03], or simply a *CSP*.

Definition 2.1.5. *A **constraint satisfaction problem instance** (CSP) is a triple $\langle V, D, C \rangle$ where:*

- *V is a finite set of variables,*
- *D is a finite set, the domain of the instance, and*
- *C is a set of constraints*

An assignment is said to be *consistent* with respect to a constraint if the projection of the constraint onto the assigned variables that are in its scope allows the assignment to those variables. If an assignment is consistent with all of the constraints in a CSP, then it is called a *partial assignment* to that CSP.[1]

Definition 2.1.6. *Let $P = \langle V, D, C \rangle$ be a CSP and $X \subseteq V$. A **partial assignment**, z, on X is an assignment to X such that for all $c \in C$ where $Y = \sigma(c) \cap X$, $z_{\downarrow Y} \in \rho(\Pi_Y c)$.*

An assignment to all the variables in a CSP, which is supported by all of the constraints, is called a *solution*.

Definition 2.1.7. *A **solution**, s, to a CSP, $P = \langle V, D, C \rangle$, is a partial assignment on V.*

CSPs which have different sets of constraints may be seen as equivalent if they have the same solutions.

Definition 2.1.8. *Two CSPs over the same set of variables are **solution equivalent** if they have the same set of solutions.*

2.2 Modelling Problems as CSPs

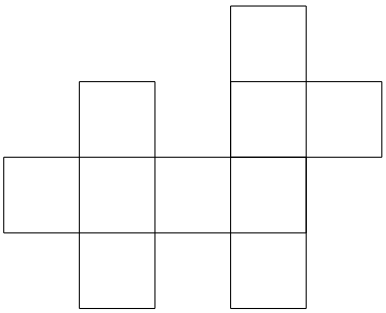
When given an instance of a real world problem to model as a CSP, it may not always be clear what the questions to model as variables are, or what the constraints between them should be. It may be that there are several ways in which the same problem instance can be modelled. What is more, some models may provide substantial improvement in solution time.

Example 2.2.1. Given the simple crossword grid and the set of possible words in Figure 2.1 we are required to fill in the grid with the words so that all of the grid squares are filled and each grid square only contains a single letter. □

There are at least two ways in which the problem instance given in Example 2.2.1 can be modelled.

Firstly, each grid square may be considered to be a variable, as shown in Figure 2.2(a). The domain would be the alphabet, and each constraint scope would be the set of possible grid squares that each word could be entered into. Each relation would be the set of variable assignments which make up whole words over the scope.

[1] This definition of a partial assignment is stronger than the more common usage (see the Handbook of Constraint Programming [RvBW06]) in which the assignments are not required to be consistent with the constraints.



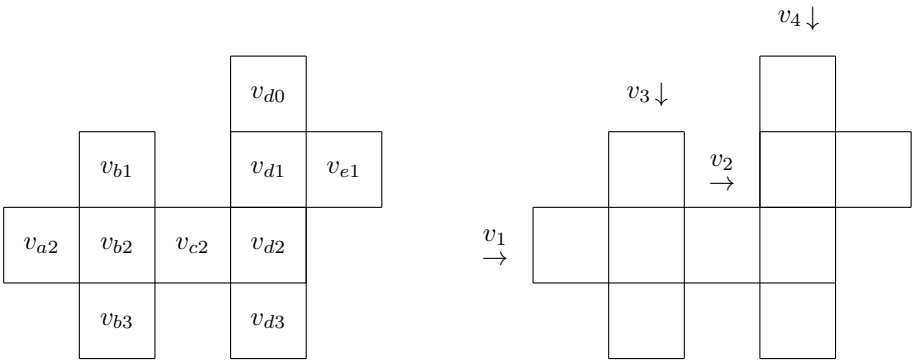
Possible words:

{on, no, in, we, ice,

cup, pot, ant, tin

gate, sink, soap, toga}

Figure 2.1: Simple Crossword Puzzle



(a) (b)

Figure 2.2: Possible Models for the Simple Crossword Game

Secondly, each set of grid squares that a word could fill may be considered to be a variable, as in Figure 2.2(b). The domain would be the set of possible words and each constraint would define the way in which the words may overlap. In this case each scope would be a pair of variables whose words overlap in the crossword, and each relation would be the set of pairs of words which have the same letter at the overlapping position.

These two simple models impart different levels of implicit knowledge about the underlying problem, for example, model (b) contains knowledge about which pairs of words can overlap in certain positions.

Modelling vs Solving

Although different constraint models may appear to represent the same problem, they are not necessarily as 'hard' to solve because knowledge applied during the process of modelling may have reduced the complexity of the original problem. That is, the modeller may have chosen to express the problem using types of constraints that the intended solver can process efficiently. See Beacham, Chen, Sillito and van Beek[BCSvB01] for an examination of how different models of crossword puzzles affect various solution techniques.

In the extreme case, a problem may be modelled by giving a single constraint over all variables, that only allows the set of solutions. However, producing this model would require solving the problem first. This trade-off between 'modelling effort' and 'solution effort' indicates that a stronger notion of equivalence between models is needed than simply whether they have the same set of solutions. Rossi, Dhar and Petrie[RDP90] provide an improved definition which states that two instances are equivalent if there is a polynomial transformation between their solutions. In Chapter 5 we will present CSP transformation methods whose results rely on there being a polynomial transformation from solutions of the transformed instance to the original.

2.3 Visualising CSPs

CSPs can be visualised as graph structures. Two common abstractions found in the literature are the *Gaifman* (or *primal*) graph and the *hypergraph*. These graphs not only provide the user with a visual aid, but are also used to aid solving CSPs (as will be shown in Section 2.6.2) and to identify tractable classes of instances (as will be shown in Chapter 3, and where more formal definitions will be provided).

In the Gaifman graph, each variable is a vertex and there is an edge between a pair of variables

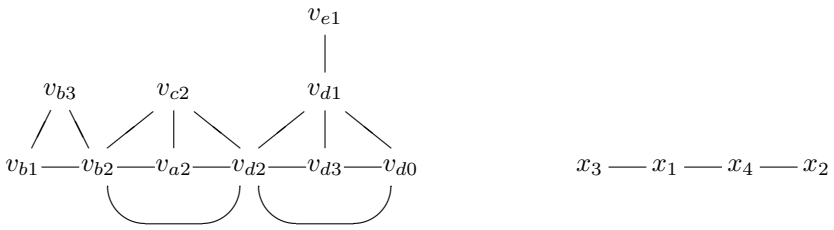


Figure 2.3: Primal Graphs

if both are present in the same scope of some constraint in the given CSP. Figure 2.3 shows the Gaifman graphs for the two models of the crossword problem from Example 2.2.1.

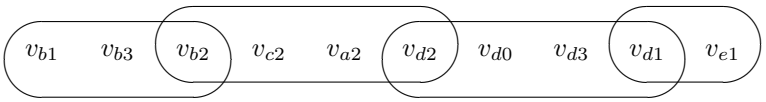
The hypergraph also has each variable as a vertex, but the edges cover sets of variables, each corresponding to a constraint scope in the given CSP. Figure 2.4 shows the hypergraphs for the two described models of the crossword problem from Example 2.2.1.

2.4 Constraint Representation

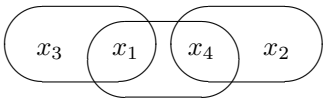
In Section 2.1 we described how a constraint could be expressed as a scope and a relation, where the scope defines which variables the constraint acts over, and the relation describes the restrictions imposed on assignments to these variables. The constraint relation is a formal way in which to express knowledge about the effects of the constraint. The ability to interpret this relation is required in order to determine whether a given assignment is supported by this constraint. The formal way in which this knowledge is expressed is called a *representation*, and a method, or algorithm, used to determine if an assignment is supported by a given constraint relation is called an *oracle*. The oracle is therefore dependent on the representation.

There are many different ways in which constraints may be represented, each of which can be categorised as being either *explicit* or *implicit*.

An explicit representation consists of a list of assignments. As such, support for any given



(a)



(b)

Figure 2.4: Constraint Hypergraphs

assignment can be determined by performing a membership check.

In some explicit representations, (such as tuples,) it may be shorter to express the disallowed assignments than the allowed ones. This is still valid because a relation constructed in this way still expresses the knowledge about the effect of the constraint. However, the oracle used to determine whether an assignment is supported will not be the same as when listing allowed assignments. As we shall show in Chapter 5, there are classes of CSPs which have efficient solution techniques when the relations are represented as allowed assignments, but which do not have efficient solution techniques when the relations are represented as disallowed assignments.

An implicit representation relies on there being some solver which is parameterised by the representational language used. Assuming the existence of this solver, a CSP can be expressed using the given representational language. This allows a shorter notation for commonly occurring constraint types to be used. These solvers are commonly known as *constraint propagators* [RvBW06], and the languages they accept consist of implicitly defined constraints for which there are good propagation algorithms, such as 'AllDifferent', 'AllEqual', and 'NotAllEqual', known as *global constraints*. (See the Global Constraint Catalogue [BCDP07] for a maintained list of global constraints.)

Some applications deal with constraints which are easily represented in implicit form. For example, satisfiability and constraint logic programming. It is natural to represent a Boolean predicate or clause implicitly, and it is hard to understand what knowledge a constraint of this

type reflects when written explicitly. It is also known that certain classes of constraints, such as linear equations, are easy to solve (in a way which we shall describe in Chapter 3) and have a natural implicit representation. (For example, $x + y \leq k$ for variables x, y and constant k.)

In the next section, we shall show that the time an algorithm requires to solve a CSP is usually given as a function of the input size. This implies that when solving a CSP it may be possible to perform more processing if the CSP is described using an explicit representation rather than an implicit one (while maintaining the same function of input size) due to the potential size difference of the representations.

However, while most theoretical results assume that a CSP is given in the larger explicit representation, this is very rarely the case in practice. In Chapter 5 we will show that theoretical tractability results do not always carry over to representations other than explicitly listed allowed assignments and examine the link between representation and tractability.

Example 2.4.1. Alternative Representations of Constraints

A *tuple* is a list, and the *arity* of a tuple is the number of items it contains. A relation of arity m is a set of m-ary tuples. Given a set, S, an m-ary relation over S is a subset of the Cartesian product S^m. In the *positive* tuples representation, the relation of a constraint is the set of allowed tuples. That is, it is an explicit relation.

A *mapping* is a single assignment of a value to a variable. Given a constraint over a set of variables, σ, a single assignment to σ can be expressed as the set of mappings to these variables, called a *labelling*. A relation is then represented as a set of labellings.

A constraint of arity m can be represented as an m-dimensional *Boolean matrix* in which each dimension represents a variable in the scope and each dimension is labeled with the domain values of the corresponding variable. This matrix can be used to express a relation by using the flags TRUE and FALSE at each position in the matrix to specify whether that combination of values for the m variables is an allowed assignment or not.

When all variables have Boolean domains, the constraints can be expressed as *clauses*. As each variable has only two possible values in its domain they may be viewed as truth values and referred to as TRUE and FALSE. Each variable and value pair can now be written as either just the variable, x, to express TRUE or the negation of the variable, $\overline{x}$, to express FALSE. These variable and value pairs are called *literals*. x is a positive literal, whereas $\overline{x}$ is a negative literal.

A clause is a disjunction of literals. For example, $(x \vee \overline{y} \vee z)$. A logic *formula* is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses. The *Satisfiability Problem* (SAT) is a well known NP-Complete decision problem[Coo71] whose instances are logic formulas in CNF. □

2.5 Complexity

When analysing algorithms, such as those which perform some action on CSPs, we are interested in their complexity. In other words, how much time they will take (or how much space they require) as a function of their input size. In order to compare algorithms, we usually consider their worst case complexity, or if this is not known exactly, a bound on their worst case complexity. Informally, the complexity of an algorithm is considered to be that of the component which dominates as the input size is increased. More formally, we use the Big-Oh (or Landau[HW84]) notation which describes the upper bound for a function (or algorithm):

Definition 2.5.1. *For input size n, the function $f(n)$ is $O(g(n))$ if there exist two constants N, K such that $\forall n \geq N$, $0 \leq f(n) \leq K.g(n)$.*

Classes of instances are often classified by the required complexity of those algorithms capable of performing certain functions on their members[GJ79].

The class of instances for which an answer to the decision problem (determining whether an instance has a solution) can be determined in polynomial time, with respect to the input size, is called **P** (or Polynomial-Time). The class of instances for which there is a polynomial time algorithm, with respect to the input size, that can determine whether a candidate solution is actually a solution is called **NP** (or Nondeterministic Polynomial-Time). By definition, $\mathbf{P} \subseteq \mathbf{NP}$, but it is an open question as to whether $\mathbf{P} = \mathbf{NP}$.

Two other classes of interest are NP-Complete and NP-Hard. **NP-Complete** is a subclass of NP. An instance, I, in NP is in NP-Complete only if all instances in NP can be converted to I in polynomial-time[2]. **NP-Hard** is the class of instances (not necessarily in NP) for which there is a polynomial-time conversion from an instance in NP-Complete (hence NP-Complete is a subclass of NP-Hard).

As stated in the Motivation section, the class containing all constraint satisfaction problems is, in general, NP-hard[Mac77][GJ79].

2.6 Solving CSPs

When solving a CSP, the goal may be to: determine if a solution exists (the decision problem), find a single solution (the search problem), find the best solution (optimisation), or even find all solutions to the instance. Normally, decision is equivalent to search [Coh04], so here we concentrate

[2]So, if P = NP-Complete, then P = NP

on finding a solution. There are two ways in which this can be done: either by using a constructive method such as backtrack based search, or a stochastic method such as simulated annealing. In this thesis, we are primarily concerned with solution techniques based on constructive search as it is both sound and complete.

When carrying out a constructive search technique, partial assignments are extended until a solution to the CSP is generated.

2.6.1 Search Algorithms

Backtracking

Chronological backtracking (BT) [GB65] is the simplest form of complete search, employing a depth first traversal of the search space. The algorithm builds up an assignment by selecting a variable and mapping it to a value from the domain. If this assignment is consistent with respect to the constraints, it selects the next variable and assigns it a value from the domain. If at any point the current assignment is not consistent, it selects an alternate domain value for the last assigned variable and continues. If all domain values have been tried for this variable, it *backtracks* to the previous variable and selects a new domain value before continuing. The algorithm stops when it has either built up a full assignment which is consistent (solution) or when there are no previous variables to roll back to. By modifying the algorithm to treat solutions in the same way as inconsistent assignments, it will continue searching to find all solutions.

In the worst case, this algorithm will have to consider every possible assignment in the search space. So, for a CSP with n variables and a domain containing d values, the time complexity of this algorithm is $O(d^n)$.

The following algorithms improve on the average case time complexity of Backtracking, however their worst case performance remains $O(d^n)$.

Forward Checking

Assume that the chronological backtrack algorithm is being used to solve a CSP with ten variables, $v_0, \ldots, v_9$, and that it is using a fixed variable ordering such that assignments are constructed from v_0 to v_9. It may be that instantiating variable v_2 with some value, d, leads to no solutions because there is no value for v_8 which gives a consistent partial assignment when v_2 has value d. BT will keep trying to build the assignment up to v_8 until it has tried all possible combinations of v_3 to v_7 which are consistent.

Forward Checking (FC) [HE79] works in a similar manner to chronological backtracking but tries to avoid these problems which are caused early in the assignment, but only detected later, by propagating the effect of each variable assignment (which forms a partial assignment) forward to the unassigned variables. The propagation in this algorithm is done by reducing the possible values which may be assigned to the remaining (unassigned) variables as the algorithm executes. To do this, the forward checking algorithm maintains a *current domain* for each future variable.

Whenever a variable is assigned a value to give a new partial assignment, then for every currently unassigned variable, v_j, this algorithm goes through the current domain of v_j and checks to see if the current partial assignment would extend to variable v_j with this value. If it does not, then it removes this value from the current domain of v_j. If the current domain of any unassigned variable becomes empty (*domain wipeout*) during this step, then the domain values removed by this step are restored and another value is attempted for the current variable. If there are no more values to try, then this algorithm backtracks in the same manner as chronological backtracking but with the additional step of restoring any domain values removed by the previous variables' assignment.

Forward checking will never have to try more variable assignments than chronological backtracking. However, the forward checking algorithm has a greater overhead due to the extra processing needed to propagate each assignment and the extra storage required to keep track of the current domains (see [KvB97]).

Backjumping

As with FC, Backjumping (BJ) [Gas79] also attempts to overcome the problem whereby the variable assignment responsible for a dead end may be further back in the assignment than simply the previous one. Rather than propagate the effect of variable assignments forward to the unassigned variables, it keeps records which allow it to determine the nearest culprit variable in the partial assignment when it comes to a dead end. So, rather than failing earlier by propagating information forwards as in forward checking, this algorithm potentially jumps back to an earlier point in the partial assignment than the previous variable when it rolls back.

Gaschnig's original Backjumping algorithm fails on leaf nodes only. For each variable, it keeps a record of the most recently instantiated earlier variable which was found to be incompatible with any of its values. Graph based Backjumping uses the structure of the graph and jumps back to the most recent connected variable (that is, to a variable in the same scope as the unsatisfied constraint). It will also perform internal back jumps if there are no more values to try for a

variable, rather than just roll back as in Gaschnig's original method.

Conflict Directed Backjumping

Conflict Directed Backjumping (CBJ) [Pro93] combines aspects of both Gaschnig's original Backjumping method and the graph based method. This algorithm maintains a *conflict set* for each variable. Whenever a previous variable fails a consistency check with the current variable, it is added to the conflict set of the current variable. If there are no more values to try for the current variable, then it jumps back to the least recently assigned variable in the conflict set (of the current variable). Upon jumping back, the contents of the conflict set of the current variable are added to the conflict set of the variable it has jumped back to (excluding itself). This is a simple learning method which builds up knowledge about why values have been removed from the domain of each variable.

2.6.2 Heuristics

We have seen that all of the backtrack based algorithms shown in this chapter build up a partial assignment one variable at a time. The order in which both the variables are assigned, and the values are selected, may have a substantial impact on the time taken to find a solution. For example, if it was possible to select all the values in a solution first time, then a solution would be found quickly without the need to backtrack. To achieve this consistently, the solutions would need to be known in advance, but there may be other attributes of a CSP which could be used to make more informed ordering decisions.

Commonly used variable orderings [EFW$^+$02] include: Random, Most constrained variable, Least constrained variable and Least remaining values (the variable with the smallest remaining domain, also known as fail-first). There are also several heuristics based on the structure of the underlying (primal) constraint graph.

Another consideration when choosing a variable ordering is whether it should be static or dynamic. Although determining an initial ordering by applying some metric only requires a single upfront calculation, it may be that the property this order was based upon changes dramatically as variables are assigned.

Given a CSP to solve, there may be some variable ordering for which the algorithm used could solve the CSP very quickly (that is, by using this ordering, the number of backtracks required is minimised), but there may be no easy way to determine which ordering this is. This idea is based on the notion of certain CSPs having a *backbone* or *backdoor* [WGS03], that is some set of variables

which, once assigned, consistently allow the rest of the CSP to be solved quickly. Analysis has found that *Random Restarts* [GSMT98] can be very effective for solving certain types of CSPs. With Random Restarts, an attempt is made to solve the CSP using some variable ordering for a fixed amount of time (or some other metric) and if it is not solved within that time, the variables are reordered (either randomly or using another heuristic) and another solution attempt is started.

2.6.3 Preprocessing and Consistency

The previous section stated that under certain circumstances the effectiveness of backtrack based solution algorithms can be improved by employing some form of heuristic. Depending on the type of CSP, it may also be possible to improve these algorithms by employing some form of preprocessing to build up a store of useful information about the CSP before attempting to solve them. This information can then be used to help direct the search algorithm.

During execution, it would not be sensible for an algorithm to select a value for a variable such that the assignment to that single variable is not supported by one of the constraints. A simple preprocessing step could be performed to reduce the domain of each variable to just those values which may exist in consistent assignments. This removal of values from the domains of individual variables achieves *node consistency* (or 1-consistency), and it can be done in linear time (for positive explicit representations).

Definition 2.6.1. *A CSP, P, is **node consistent** if, for all variables in P, each mapping from a single variable to any value in its domain is a partial assignment to P.*

Consider the case where an assignment is constructed by assigning a value to variable v_a, then considering variable v_b. If it is true that for every value which could be assigned to variable v_a, there is at least one value which could be assigned to variable v_b to form a partial assignment, then the variable v_a is said to be *arc consistent* (or 2-consistent) relative to variable v_b. A CSP, $\langle V, D, C \rangle$, is said to be arc consistent if all variables in V are arc consistent relative to all other variables in V.

Definition 2.6.2. *Two variables, v_i, v_j, in a CSP, P, are **arc consistent** if, for every value in the domain of v_i which is consistent with respect to the constraints of P, there exists a value in the domain of v_j such that this assignment to v_i, v_j is a partial assignment to P.*

*A CSP, $\langle V, D, C \rangle$, is said to be **arc consistent** if, for every pair of variables $v_i, v_j \in V$, where $v_i \neq v_j$, the pair v_i, v_j is arc consistent.*

If at some stage during the solution process the CSP is in a node consistent state with no domain wipeouts, then assigning any remaining domain value to any unassigned variable must be consistent with the current partial assignment. If the CSP is arc consistent at this point, then not only must there be a value for the variable being considered, but whatever value is selected will not cause a domain wipeout for any of the unassigned variables.

Recall the Forward Checking algorithm from Section 2.6 in which the current variable assignment is propagated to all unassigned variables. It makes the current variable assignment arc consistent with each of the unassigned variables. However, the set of unassigned variables may not all be arc consistent amongst themselves. Sabin and Freuder [SF94] presented a method by which arc consistency can be used in a solution algorithm, called Maintaining Arc Consistency (MAC). Whenever a variable is assigned, the entire instance, together with the current partial assignment, is made arc consistent by the removal of incompatible domain values. This prunes more of the incompatible assignments from the search space than forward checking, at the expense of more processing.

Arc consistency considers only a pair of variables. This can be extended to the general case, *k-consistency*, which states that for every valid assignment to $k-1$ variables, there must be some valid assignment to any k^{th} variable. This does not guarantee that the instance is j-consistent, where $j < k$. A CSP which is j-consistent for all values $j = 1, \ldots, k$ is called *strongly k-consistent.*

Definition 2.6.3. *A CSP, P, is* **k-*consistent*** *if, for any consistent partial assignment to $k-1$ variables of P, there exists a value in the domain of every remaining unassigned variable of P, such that the partial assignment can be extended to this value for this variable to create a new partial assignment (to k variables of P).*

A CSP is strongly k-consistent *if it is j-consistent, for $j = 1, \ldots, k$.*

If a CSP, P, is strongly n-consistent where n is the number of variables in P, then P is said to be *globally consistent.* If a CSP is globally consistent, then it can be solved without backtracking. Making a CSP globally consistent using preprocessing is therefore hard in general as it is equivalent to solving the instance.

Consistency may also be considered between pairs of constraints, rather than variables. A pair of constraints is said to be *pairwise consistent* if the projection of each constraint onto the intersection of their scopes is the same.

Definition 2.6.4. *A CSP, $P = \langle V, D, C \rangle$, is* ***pairwise consistent*** *[JJNV89] if, for any pair of constraints $c_0, c_1 \in C$, $\Pi_{\sigma(c_0) \cap \sigma(c_1)} c_0 = \Pi_{\sigma(c_0) \cap \sigma(c_1)} c_1$.*

As we shall see in Chapter 3, establishing pairwise consistency using a polynomial algorithm (quadratic-time for positive explicit representations) is strong enough to solve certain classes of CSPs.

2.7 Summary

In this chapter we have introduced the concept of Constraints and the Constraint Satisfaction paradigm. We have also introduced some of the basic notation which will be used throughout the remainder of this thesis.

We have explained that there are two distinct representation types for constraints: explicit and implicit. We have stated that explicit constraints can be used to represent any arbitrary finite CSP, but that some constraints are more naturally represented using implicit constraints.

We have given several common solution algorithms and described how heuristics can be used to optimise their use. We have also shown how propagation techniques which enforce minimum levels of consistency can be used to further improve these algorithms.

It has been our intention to show that there are many decisions to be made during the process of modelling a problem using constraints, and then solving it, and that these decisions are not straightforward. By making the correct decisions when modelling, we can affect the hardness of finding solutions, either by imparting some implicit knowledge into the construction or by noticing that the problem can be modelled using only constraints for which the solution algorithms work well.

We are interested as to whether it is true that the same problem modelled in different ways are equivalent. Recall that in the crossword example (Example 2.2.1) we were able to include implicit information about the length of words that will fit in each position simply by selecting to model the problem in a certain way.

In this thesis we intend to show that different models contain differing levels of implicit knowledge imparted by the modeler, and are therefore not the same constraint satisfaction problem. We intend to show that two CSPs are only truly equivalent if they are not only solution equivalent, but there must also be an efficient transformation between these different models.

Chapter 3

Efficiently Solvable Classes

In general, the Constraint Satisfaction Problem is NP-hard [Mac77][GJ79], so an efficient algorithm for solving all CSPs does not exist. A *class* of CSPs is a collection containing those CSPs which have some defined set of properties and, as is demonstrated in this chapter, much effort has gone into identifying classes of CSPs for which there are efficient solution algorithms.

Each CSP has a set of properties which are referred to as *parameters*. For example: the number of variables, the domain size, the arity (the maximum number of variables in any constraint scope), the constraint probability (for each subset of the variables, the probability of there being a constraint with that scope) and the constraint tightness (the ratio of the number of incompatible assignments to the total number of possible assignments).

A class of CSPs may be defined as having some fixed parameter values that are independent of the instance size. It may be that, for one of these parameters, the classes defined by increasing or decreasing its value go from having a very high probability of solution, to a very low probability of solution (or vice versa). If the threshold range of the parameter under which this occurs becomes more defined (smaller) as the instance size increases, then we call this phenomenon a *phase transition* [TCC$^+$91].

The instances within a phase transition region are commonly the hardest to solve since one side is underconstrained, and so may have many solutions, and the other overconstrained such that there is very little chance of there being a solution. Gent and Walsh showed this to be the case for k-SAT[GW94].

Although the phase transition region is a good indicator of how hard an instance is likely to be to solve, it provides no guarantees. It is simply a probability distribution, so there may well be hard to solve 'outliers' in the under and over constrained regions.

In this chapter we shall consider two different properties of constraint instances which can be used for classification; structure and language. These properties not only cross the bounds of the simple parameter view, but can also be used to soundly define classes for which there are efficient solution algorithms.

Traditionally, a class of CSPs for which there is an efficient solution algorithm has been referred to as *tractable*. We prefer to use a more modern definition which extends this so that a class of constraint problems is only tractable when there is both an efficient solution algorithm and an efficient algorithm for determining membership of this class for any general CSP.

Definition 3.0.1. *A class of CSPs, R, is* ***tractable*** *if there is both a polynomial-time algorithm for solving any instance in R, and a polynomial-time algorithm for determining whether any given instance is a member of R.*

Although many tractable subclasses have now been identified, there is still no general taxonomy of constraint classes which completely ties together structure and language. In recent years, research has started to move towards understanding the underlying reasons why some subclasses are tractable while others are not. Hopefully, some deeper property which unifies tractable classes will be identified.

3.1 Structure

Constraint Satisfaction Problem instances have an associated *hypergraph*. It is the properties of this graph that are referred to when talking about a CSP's structure.

Definition 3.1.1. *A* ***hypergraph****, H, is a pair $\langle V, E\rangle$, where V is a set, called the* vertices *of H, and E is a multiset of subsets of V, called the* hyperedges *of H.*

When used in structural tractability theory, hypergraphs are commonly defined such that the hyperedges are sets, rather than multisets, as these results assume that all constraints are provided in the same explicit representation. In Chapter 5, CSPs will be considered whose constraints are not all provided using the same representation. It is then necessary to be able to identify when constraints using more than one representation may be acting on a given scope.

Definition 3.1.2. *For any CSP, $P = \langle V, D, C\rangle$, the* ***structure*** *of P, denoted $\sigma(P)$, is the hypergraph $\langle V, \{\sigma \mid \langle \sigma, \rho\rangle \in C\}\rangle$.*

Definition 3.1.3. *A* ***graph****, G, is a hypergraph in which each edge contains exactly two vertices. A graph is called a* ***tree*** *if it contains no cycles.*

Definition 3.1.4. *The **Gaifman graph** (or **primal graph**) of a CSP is a graph in which the variables of the CSP are the vertices and there is an edge between pairs of vertices when they are both found in a constraint scope. That is, there is an edge, $(v_i, v_j) \in E$ if, and only if, for some scope σ_s, $v_i \in \sigma_s$ and $v_j \in \sigma_s$.*

Definition 3.1.5. *The **incidence graph** of a CSP $P = \langle V, D, C \rangle$ is a bipartite graph whose vertices are V and C. There is an edge in the incidence graph between a variable $v \in V$ and a constraint $c \in C$ if, and only if, $v \in \sigma(c)$.*

An edge between two variables in the Gaifman graph indicates that these two variables are both in at least one constraint, whereas an edge in the hypergraph is analogous to a constraint scope. As such, the Gaifman graph and the hypergraph are the same only for binary CSPs. This implies that the Gaifman graph is only a crude approximation of the structure of a CSP, whereas the hypergraph is an exact relational structure. For example it is not possible to identify whether a CSP contains only one constraint when considering only the Gaifman graph.

Example 3.1.6. Consider the following CSP over the domain $\{r, g, b\}$:

$$\sigma_0 = \{v_0, v_1\}$$
$$\rho_0 = \{ \quad \{v_0 \mapsto r, v_1 \mapsto b\}, \{v_0 \mapsto r, v_1 \mapsto g\}, \{v_0 \mapsto b, v_1 \mapsto r\}, \\ \{v_0 \mapsto b, v_1 \mapsto g\}, \{v_0 \mapsto g, v_1 \mapsto r\}, \{v_0 \mapsto g, v_1 \mapsto b\} \quad \}$$

$$\sigma_1 = \{v_1, v_2\}$$
$$\rho_1 = \{ \quad \{v_1 \mapsto r, v_2 \mapsto b\}, \{v_1 \mapsto r, v_2 \mapsto g\}, \{v_1 \mapsto b, v_2 \mapsto r\}, \\ \{v_1 \mapsto b, v_2 \mapsto g\}, \{v_1 \mapsto g, v_2 \mapsto r\}, \{v_1 \mapsto g, v_2 \mapsto b\} \quad \}$$

$$\sigma_2 = \{v_0, v_2, v_3\}$$
$$\rho_2 = \{ \quad \{v_0 \mapsto r, v_2 \mapsto b, v_3 \mapsto g\}, \{v_0 \mapsto r, v_2 \mapsto g, v_3 \mapsto b\}, \\ \{v_0 \mapsto b, v_2 \mapsto r, v_3 \mapsto g\}, \{v_0 \mapsto b, v_2 \mapsto g, v_3 \mapsto r\}, \\ \{v_0 \mapsto g, v_2 \mapsto r, v_3 \mapsto b\}, \{v_0 \mapsto g, v_2 \mapsto b, v_3 \mapsto r\} \quad \}$$

The primal graph and hypergraph for this CSP are shown in Figures 3.1 and 3.2. □

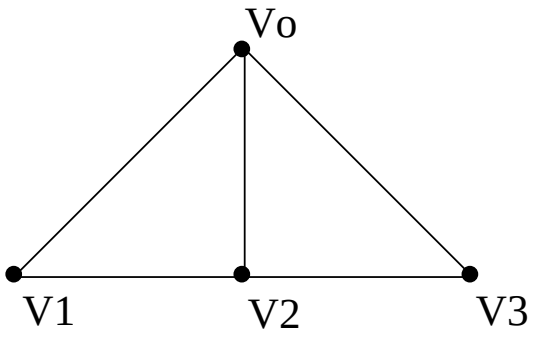


Figure 3.1: Primal Graph of Example 3.1.6

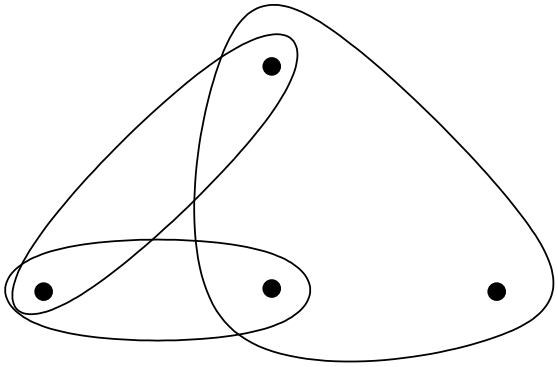


Figure 3.2: Hypergraph of Example 3.1.6

The information represented by both the Gaifman graphs and hypergraphs is purely structural. That is, they both show the connections between variables, but not the effect of assignments or domains. Even with just this structural information it is still possible to identify tractable classes as there are some structural properties which guarantee tractability regardless of the relations. For example, Grohe has shown [GM99] that if a class is defined just by limiting its Gaifman graph, then the class of CSPs which have an efficient solution is the class of CSPs whose Gaifman graph has bounded *treewidth*. (The more general definition of the treewidth of a hypergraph is given in Definition 3.1.10. However, it can be defined directly on a Gaifman graph such as by Freuder [Fre82].)

When considering structure, it is usual to only consider hypergraphs that are both reduced and connected as unreduced hypergraphs are structurally equivalent to their reduced forms (when considering positive explicit representations) and unconnected hypergraphs can be considered as a collection of separate connected hypergraphs.

Definition 3.1.7. *Consider a hypergraph, $\langle V, E\rangle$, and an edge $e \in E$. The edge e is* ***repeated*** *if the multiset E contains more than one copy of e. The edge e is* ***maximal*** *if it is contained in no other edge in E.*

A hypergraph in which all edges are maximal (and thus having no repeated edges) is ***reduced****.*

Definition 3.1.8. *A hypergraph $\langle V, E\rangle$ is said to have a* ***path*** *of length k between two vertices $v_0, v_{k-1} \in V$ if for each $i = 0, \ldots, k-2$ there exists an edge $e_i \in E$ such that $\{v_i, v_{i+1}\} \subseteq e_i$. If $v_0 = v_{k-1}$, then the path is called a* ***cycle****.*

Two edges, e_a and e_b, in a hypergraph are said to be ***connected*** *if there is a path from some vertex contained in e_a to some vertex contained in e_b. If there exists a path between every pair of vertices in the hypergraph, then the hypergraph itself is said to be* connected*.*

As with parameters, classes of CSPs may be defined based on the structural properties of their members.

Definition 3.1.9. *For any family of hypergraphs, $\mathcal{H}$, the class of all CSPs with structure contained in $\mathcal{H}$ is denoted $\Psi(\mathcal{H})$.*

3.1.1 Acyclicity

Some hypergraphs allow efficient solution because they can be represented as a tree-like structure. A *tree decomposition* is a mapping from a graph to a tree.

Definition 3.1.10. *Given a hypergraph $G = \langle V_G, E_G\rangle$ and a tree $T = \langle V_T, E_T\rangle$, $\langle T, \chi\rangle$ is a* ***tree decomposition*** *of G (where χ is a mapping function from the vertices in T to sets of vertices in G) if:*

1. *Every vertex in V_G is in $\chi(v)$ for some v in V_T.*
2. *For every two vertices, x, y, contained in an edge in E_G there exists v in V_T such that x and y are both in $\chi(v)$.*
3. *Given three vertices v_0, v_1, v_2 in V_T such that v_1 lies on a path in T from v_0 to v_2, $\chi(v_0) \cap \chi(v_2) \subseteq \chi(v_1)$.*

The ***width*** *of $\langle T, \chi\rangle$ is $\max(|\chi(v)| - 1), v \in V_T$. The* ***treewidth*** *of G, $tw(G)$, is the minimum width of all possible tree decompositions of G.*

A join tree of a hypergraph is a restricted form of tree decomposition in which each node is exactly an edge of the hypergraph. As a result, not all hypergraphs have a join tree.

Definition 3.1.11. *A **join tree** of a hypergraph, $H = \langle V, E \rangle$, is a connected tree, J, whose nodes are the edges of H. Whenever the vertex, $x \in V$, occurs in two edges, $e_1, e_2 \in E$, then x occurs in each node of the unique path connecting e_1 and e_2 in J.*

For a graph, G, we have that $\mathrm{tw}(G) = 1$ precisely when G is a tree. This implies that G has a join tree.

Definition 3.1.12. *A hypergraph is called **acyclic** if it has a join tree.*

Acyclicity is a structural property that defines a tractable subproblem. Any instance whose underlying hypergraph is acyclic can be solved in polynomial time [BFMY83].

It is possible to determine whether a hypergraph is acyclic by performing *Graham's Algorithm* [Gra79] (Algorithm 1). Graham's Algorithm repeatedly removes edges contained entirely within other edges (*non-maximal edges*) and vertices that are contained in only a single edge (*isolated vertices*) until it can no longer remove more edges or vertices. If it has successfully removed all vertices and edges, then there is a join tree and the algorithm returns TRUE, otherwise there is no join tree and the algorithm returns FALSE.

Definition 3.1.13. *A vertex of a hypergraph is **isolated** if it is contained in at most one edge.*

Algorithm 1: Graham's Algorithm

```
1 Takes H = ⟨V, E⟩, returns TRUE / FALSE
  1: repeat
  2:     Reduce H
  3:     Remove all isolated vertices from H
  4: until No edges or vertices are removed
  5: if E = ∅ then
  6:     Return TRUE
  7: else
  8:     Return FALSE
  9: end if
```

It is also possible to determine whether a hypergraph is acyclic by looking at its corresponding Gaifman graph. If the Gaifman graph of a CSP is chordal and conformal, then the hypergraph is acyclic.

Definition 3.1.14. *A graph is **chordal** (or* triangulated*) if, for all cycles with length greater than three, there is an edge between two vertices which are not adjacent in the cycle.*

Definition 3.1.15. *A **clique** is a set of vertices $S \subseteq V$ in a graph $\langle V, E \rangle$ such that every pair of vertices in S is connected by an edge in E. A clique is said to be **maximal** if it is not contained in another clique.*

Definition 3.1.16. *The Gaifman graph of a CSP is* ***conformal*** *if its maximal cliques are exactly the constraint scopes in the CSP.*

Several non-trivial classes of CSPs have been found which are tractable because there exists a *structural decomposition* method that reduces their structure to an acyclic hypergraph.

3.1.2 Reduction to Acyclic

Graham's Algorithm fails if it reaches a state in which there are no longer any isolated vertices or subsumed edges that can be removed, but the hypergraph is not empty. It may be possible to modify a hypergraph such that Graham's Algorithm does complete by removing vertices or edges, combining edges, replacing edges, or potentially adding edges.

By considering the different ways in which a hypergraph could be modified to achieve acyclicity, many reduction methods with varying degrees of power were formed. These methods can be ranked by some width parameter, as CSPs can be solved in a time that is polynomial in the size of the CSP with respect to its width [Fre82].

We shall now look at some of these commonly used reduction methods, starting with *Cycle Cutsets* [Dec92] whose width parameter is the number of vertices which need to be removed in order to make the hypergraph acyclic.

Cycle Cutsets

Given a hypergraph, $H = \langle V, E \rangle$, a Cycle Cutset is a set of vertices, $X \subseteq V$, such that the resulting hypergraph $H' = \langle V - X, E_{|_{V-X}} \rangle$ is an acyclic hypergraph.

To solve a CSP using the cycle cutset method, a consistent assignment to the variables represented by the vertices in X is chosen and then this assignment is extended to the remaining variables using a known polynomial-time algorithm for the acyclic hypergraph H'. If this does not yield a solution, another assignment to the variables represented by the vertices in X must be tried and this process repeated until either a solution is found or all consistent assignments have been tried (at which point all possible solutions have been found, or it has been demonstrated that there are none).

The class of CSPs whose minimum cycle cutset contains at most k vertices can therefore be solved in time $O\left(|D|^k\right)$ in general. Identifying the cutsets $O\left(|V|^k\right)$ time.

As well as removing vertices in order to reduce a hypergraph to an acyclic subgraph, as in Cycle Cutsets, edges may also be removed in a similar manner to form Cycle Hypercutsets [GLS00]. In

the case of Cycle Hypercutsets, the width parameter is the number of edges which need to be removed.

Next, we shall consider *Hinge Trees* [GJC94] where acyclicity is achieved by joining existing edges in the hypergraph.

Hinge Trees

The Hinge Tree decomposition identifies cyclic components in the hypergraph which are connected in the correct way as to form a tree.

A *hinge* of a hypergraph is a *connected* set of at least two edges such that every *connected component* of the remaining edges meets the hinge in exactly one edge (called a *separating edge*).

Definition 3.1.17 (Section 2.3, [GJC94]). *Let (V, E) be a hypergraph, let $H \subseteq E$, and let $F \subseteq E \setminus H$. F is called **connected** with respect to H if, for any two edges $e, f \in F$, there exists a sequence $e_1, \ldots, e_n$ of edges in F such that (i) $e_1 = e$; (ii) $e_n = f$; and (iii) for $i = 1, \ldots, n-1$, $e_i \cap e_{i+1}$ is not a subset of $\bigcup H$. The maximal connected subsets of $E \setminus H$ with respect to H are called the **connected components** of (V, E) with respect to H.*

Definition 3.1.18 (Definition 3.1, [GJC94]). *Let (V, E) be a reduced and connected hypergraph, and let H be either E or a proper subset of E containing at least two edges. Let $H_1 \ldots H_m$ be the connected components of (V, E) with respect to H. H is called a **hinge** if, for $i = 1 \ldots m$, there exists an edge h_i in H such that h_i contains all the vertices contained in H_i that are also contained in H.*

$$(\bigcup H_i) \cap (\bigcup H) \subseteq h_i$$

*The edge h_i is called a **separating edge** for H_i.*

A hypergraph is covered by a set of hinges if each of its edges is contained in at least one of the hinges. A hypergraph can always be covered by a set of hinges since the set of all edges is a hinge.

A *minimal hinge* of a hypergraph is a hinge which does not properly contain any other hinge. There may be many different minimal hinge covers for a particular hypergraph. Gyssens et al. [GJC94] proved that the size of the largest hinge in a minimal hinge cover is an invariant of the hypergraph. In other words, given a minimal hinge cover for a hypergraph, there is no other minimal hinge cover for this hypergraph whose largest hinge is smaller.

The *hinge width* of a hypergraph is the size of the largest hinge in any minimal hinge cover. This is referred to as its degree of cyclicity.

Gyssens et al. also demonstrate that there is a polynomial-time algorithm which can find a minimal hinge cover for any hypergraph. In fact, this algorithm generates a tree of minimal hinges that cover the hypergraph (i.e. a join tree of maximal hinges), called a *hinge tree*.

For a given degree of cyclicity, there exists a polynomial-time algorithm for solving any CSP whose underlying hypergraph is at most this degree. Members of this class can be identified in polynomial time by generating a hinge tree and observing the size of the largest node. A solution can then be found by synthesising constraints on the nodes of the hinge tree. Since the degree of cyclicity is limited, this constraint synthesis is polynomial time and the resultant acyclic CSP can be efficiently solved.

New edges may also be added to a hypergraph in order to make it acyclic. *Tree Clustering* [DP89] does this by triangulating cycles in the primal graph which are not chordal.

Tree Clustering

Given a CSP, edges may be added to its primal graph until it is triangulated, and hence acyclic. Each of the maximal cliques now corresponds to a subproblem of the CSP containing those constraints which have scope variables as nodes in the clique. Once these cliques have been solved, we can replace them with single constraints and then solve the acyclic hypergraph using a known polynomial-time algorithm. As with the cycle cutset method, we can find single solutions to each clique and try to solve the tree with these, finding alternate solutions to the cliques if we fail to find a solution until all possible solutions to the cliques have been attempted.

Given the class of CSPs which can be decomposed using tree clustering such that the maximal clique contains at most k vertices, we can now solve this class in polynomial time bounded by D^k as the largest subproblem we have to solve, for which we do not have a polynomial-time algorithm, has D^k possible solutions. However, finding the best tree clustering is hard.

Rather than considering the processes and properties which may generate a specific structural decomposition, we could instead consider only what it means to be a structural decomposition. That is, for a given input CSP, a structural decomposition of its underlying hypergraph must provide solution equivalence on the generated CSP. This gives the notion of a *generalized hypertree decomposition* [GGM$^+$05], which describes the most powerful structural decomposition. In other words, the generalized hypertree width is never larger than any other structural width parameter.

Hypertrees

Consider the hypergraph H, and its associated primal graph G. A *generalized hypertree decomposition* of H is a triple $\langle T, \chi, \lambda \rangle$ where $\langle T, \chi \rangle$ is a tree decomposition of G and λ is a mapping which assigns to every node t of T a set of hyperedges of H, $\lambda(v)$, such that each vertex $x \in \chi(t)$ is contained in some hyperedge $e \in \lambda(t)$. ($\lambda(t)$ is a *set cover* of $\chi(t)$.)

The width of a generalised hypertree decomposition $\langle T, \chi, \lambda \rangle$ is the size of the largest set $\lambda(t)$ over all nodes t of T. The *generalised hypertree width* $\mathrm{ghw}(H)$ of H is the minimum width over all generalised hypertree decompositions of H. However, it is known that even for generalised hypertree width at most three, no polynomial-time identification algorithm exists [GMS09]. A restricted form was developed satisfying a technical condition which makes identification polynomial time. These are simply known as hypertrees and, as $\mathrm{hw}(H) \leq 3.\mathrm{ghw}(H)$, they are a good approximation [AGG05].

In order to reason about structural decompositions later in this thesis, we shall use an equivalent decomposition framework: the guarded decomposition [CJG08]. Acyclic guarded decompositions are equivalent to generalised hypertrees.

Definition 3.1.19. *A **guarded block** of a hypergraph, H, is a pair $\langle \lambda, \chi \rangle$ where the **guard**, λ, is a set of hyperedges of H, and the **block**, χ, is a subset of the vertices of the guard.*

For any CSP, P, and any guarded block, $\langle \lambda, \chi \rangle$ of $\sigma(P)$, the constraint generated by P on $\langle \lambda, \chi \rangle$ is the constraint $\langle \chi, \rho \rangle$, where ρ is the projection onto χ of the relational join of all the constraints of P whose scopes are elements of λ.

*A set of guarded blocks, Ξ, of a hypergraph is called a **guarded decomposition** of H if for every CSP, $P = \langle V, D, C \rangle \in \Psi(H)$, the instance $P' = \langle V, D, C' \rangle$, where C' is the set of constraints generated by P on the members of Ξ, is solution equivalent to P.*

Definition 3.1.20. *A guarded block, $\langle \lambda, \chi \rangle$, of a hypergraph, $H = \langle V, E \rangle$, **covers** a hyperedge $e \in E$ if the vertices in e are all contained in χ.*

*A set of guarded blocks, Ξ, of a hypergraph, $H = \langle V, E \rangle$, is called a **guarded cover** for H if each hyperedge of H is covered by some guarded block of Ξ.*

*A set of guarded blocks, Ξ, of a hypergraph $H = \langle V, E \rangle$ is called a **complete guarded cover** for H if each hyperedge $e \in E$ occurs in the guard of some guarded block Ξ which covers e.*

*The **width** of a set of guarded blocks is the maximum number of hyperedges in any of its guards.*

It has been shown that a set of guarded blocks, Ξ, of a hypergraph, H, is a guarded decompo-

sition of H if and only if it is a complete guarded cover for H [CJG05]. A set of guarded blocks is acyclic if the set of blocks is an acyclic set of hyperedges over their vertices.

Definition 3.1.21. *A **join tree** of a set of guarded blocks, Ξ, of a hypergraph, $H = \langle V, E \rangle$, is a connected tree, J, whose nodes are elements of Ξ, such that, whenever the vertex $x \in V$ occurs in two blocks of Ξ then x occurs in each block of the unique path connecting them in J.*

A set of guarded blocks is acyclic if it has a join tree.

Guarded decompositions may be used to describe known structural decompositions, such as hypertrees, using simple restrictions of the types of guarded blocks allowed [CJG05].

The structural classes described in this section all have the common feature that they form a tree of components for which there is a backtrack-free search. Solving an entire instance, requires solving these components first, and then performing a backtrack-free search on the tree. These components are hard to solve, but are of a bounded size, so the tractable class of instances is always defined by the size of the largest component that is permitted. However, there are some structures which derive tractability without using acyclicity.

3.1.3 Polynomial Solution Size

An alternative structural reason for tractability was discovered that does not depend on a reduction to acyclic structure.

Fractional Edge Covers

A *fractional edge cover* [GM06] is the assignment of weights to the edges of a hypergraph such that each variable has a total (edge) weight of at least one.

The *fractional edge cover number* of a hypergraph is the smallest total weight which can be used to form a fractional edge cover.

Grohe and Marx [GM06] have shown that the class of CSPs whose hypergraphs have a bounded fractional edge cover number can be solved in polynomial time. This is done by showing that given a CSP, all solutions can be enumerated in polynomial time for a fixed fractional edge cover number. Fractional edge covers are extended to *fractional hypertrees*, but in this case identification is hard.

3.2 Language

We have seen that it is possible to define tractable classes of CSPs by imposing restrictions on their structure. That is, on the interaction of their constraint scopes. However, there are other restrictions which can be used to identify tractable classes.

Tractable classes of CSPs may be defined by imposing restrictions on the relations of the constraints. A restricted set of relations is called a *constraint language*.

In this section, we shall give a brief overview of the fundamental language-based definitions. However, we shall not give a full review of state of the art research in relational tractability as our work in this thesis is not concerned with extending current relational theory. For a more in-depth survey of current theory, the interested reader may wish to refer to the works presented at 'The Constraint Satisfaction Problem: Complexity and Approximability' seminar [BGKK10].

Definition 3.2.1. *A **constraint language** over domain D is a set of relations (over D). For a constraint language, Γ, we denote by* $\mathrm{CSP}(\Gamma)$ *the set of all CSPs whose constraint relations are in* Γ.

A finite constraint language is tractable if there is a polynomial-time algorithm to solve any instance from $\mathrm{CSP}(\Gamma)$. *An infinite constraint language is said to be tractable if each finite subset of the language is tractable.*

All tractable languages can be defined in terms of closure operators called polymorphisms.

Definition 3.2.2. *Given a domain, D, a k-ary **operation** on D, ϕ, is a function $\phi : D^k \to D$.*

Assuming that assignments in a relation have the same ordering, such as when represented as tuples, then any operation on D can be extended to an operation on a relation over D by applying the operation pointwise across sets of assignments.

Definition 3.2.3. *Let $\phi : D^k \to D$ be a k-ary operation on D, and let R be an n-ary relation over D.*

For any set of tuples of size k, $\{t_0, t_1, \ldots t_{k-1}\} \subseteq R$, in which repeated members are allowed, the n-ary tuple $\phi(t_0, t_1, \ldots, t_{k-1})$ is:

$$\langle \phi(t_0[0], t_1[0], \ldots, t_{k-1}[0]), \phi(t_0[1], t_1[1], \ldots, t_{k-1}[1]), \ldots, \phi(t_0[n-1], t_1[n-1], \ldots, t_{k-1}[n-1]) \rangle$$

$\phi(R) = \{\phi(t_0, t_1, \ldots, t_{k-1}) \mid \{t_0, t_1, \ldots, t_{k-1}\} \subseteq R\}$. *$R$ is ϕ-**closed** if $\phi(R) \subseteq R$.*

There are well known tractable relational classes, such as those defined over a constraint language that is closed under some semilattice operator [JCG97].

Definition 3.2.4. *Let D be a partially ordered set. For $a, b, c \in D$, c is the* ***least upper bound*** *of a and b, denoted $a \sqcup b$, if*

- $a \leq c$ *and* $b \leq c$
- $\forall x \in D$ *such that* $a \leq x$ *and* $b \leq x$, $c \leq x$

Definition 3.2.5. *A* ***semilattice****, is a partially ordered set, D, where every pair of elements in D have a least upper bound also in D. The* ***semilattice operator*** *for D, $\Phi : D^2 \rightarrow D$, returns the least upper bound of its two arguments. A constraint language Γ over D is* ***semilattice-closed*** *if each relation in Γ is closed under Φ.*

A CSP can be made pairwise consistent (recall Definition 2.6.4) in $O\left(c^2t^2\right)$ time, where c is the number of constraints and t is the number of tuples in the constraint which has the most tuples (cardinality) [JJNV89]. Jeavons, Cohen and Gyssens [JCG97] showed that after performing pairwise consistency on a CSP in which all of the constraints are semilattice-closed, the resulting CSP would also be semilattice-closed.

A solution to a CSP in which all of the constraints are semilattice-closed can be found in polynomial time by firstly imposing pairwise consistency, and then iterating over all remaining tuples while keeping track of the largest least upper bound value so far seen for each variable. Once all tuples have been iterated over, the assignment in which each variable takes its largest observed least upper bound value is a solution to the original CSP.

There can be no more tuples in each constraint of the resulting pairwise consistent CSP than in each corresponding constraint of the original, so iterating over all tuples in the pairwise consistent CSP is $O\left(c.t\right)$ time, hence for a given semilattice operator, the class of CSPs whose constraint relations are closed under this operator is tractable.

Expressibility

When a variable is assigned a value during the process of solving a CSP the set of possible values that may be assigned to the currently unassigned values is often modified accordingly (such as when using a look-ahead strategy). This is done such that it is more likely (or often guaranteed) that any future assignments are consistent with the current assignment. The process of iterating over the currently unassigned variables and restricting their set of possible assignments based

on the value assigned to another variable is called *propagation*. That is, the effect of a variable assignment is propagated to the currently unassigned variables. A constraint *propagator* is the method (or algorithm) by which the effect of a variable assignment can be propagated to (some or all of) the unassigned variables.

By restricting ourselves to a language consisting of relations for which we have efficient propagators, we are able to make solving CSPs expressed in this language easier. However, by restricting ourselves in this way, we also reduce the number of CSPs which we are able to express. As such, a constraint language can be seen as a tradeoff between efficient solution, and expressibility.

Structural tractability requires us to formulate the problem and then try to identify if the resulting instance lies in a tractable class. With a language-based approach, we can restrict ourselves in advance to a known tractable language and then try to model a given problem in it.

Definition 3.2.6. *A tractable constraint language is called* ***maximally tractable*** *if the addition of any new relation to the language gives a language which is not tractable.*

Proving that a language is maximally tractable is very difficult as it depends on the algorithms used to solve problems modeled in this language and, as we shall demonstrate in Chapter 5, this depends on the way in which the constraints are represented.

Although a particular relation may not be in a constraint language, Γ, it is still sometimes possible to express this relation using a combination of relations which are in Γ. This is done by constructing a CSP which is solution equivalent to the relation we wish to express, but whose constraint relations are all members of Γ. In doing this, we are required to associate each position in the relation we wish to model with a variable. This may be done using 'hidden variables' (that is, additional variables which can be projected out of a solution so the constraint relation of the reduced solution is the same as the relation we are trying to express). The CSP used to represent another relation is known as a *Gadget*.

Definition 3.2.7. *Given a constraint language,* Γ*, and a relation, R, which is not in* Γ*, a* ***gadget*** *for R in* Γ *with construction site X is a CSP* $G \in \mathrm{CSP}(\Gamma)$ *such that the projection of* $\mathrm{Sol}(G)$ *onto X is the relation R.*

Example 3.2.8. Gadget

Consider a simple CSP with three variables, $\{v_0, v_1, v_2\}$, over a Boolean domain and two constraints in both of which the relation is not-equals. If the scopes of these two constraints are $\langle v_0, v_1 \rangle$ and $\langle v_1, v_2 \rangle$ then there is an implied binary equality constraint between v_0 and v_2. Thus, in this CSP, whose language contains only the not-equals constraint, we are able to express the

equality constraint on the construction site $\langle v_0, v_2 \rangle$. □

When using a given constraint language, it is useful to know the entire set of relations that can be expressed using it, both directly and using Gadgets. The construction of this set of relations is possible (for a finite constraint language over a finite domain) using the *Universal Gadget* (or *Indicator problem*) [JCG96].

3.3 Fixed Parameter Tractability

A problem may have several input parameters which define the input size, for example the number of variables, size of the domain or maximum arity of the constraints. Downey and Fellows [DF99] have developed strong theory which applies to classes for which certain parameters are bounded (or fixed).

Definition 3.3.1. *A* ***parameter*** *for class C is a function which maps instances of C to the natural numbers.*

A class is called ***fixed parameter tractable*** *(FPT) with respect to parameter k if there exists a polynomial p and a solution algorithm that runs in time $f(k) \times p(n)$, where f is a function of k which is independent of the instance size n.*

Many complexity results rely on a standard complexity theoretical assumption, that the class W[1] is not equal to the class FPT. This is the parameterised complexity analogue of the assumption that NP is not equal to P.

3.4 Relational Structure

We have seen that classes of CSPs may be defined in terms of their structure or language (relations), and that both of these techniques are powerful enough to define tractable classes of CSPs. It is also possible to combine both techniques, by specifying both a structure and a language, to further sub-classify CPSs. Furthermore, tractable classes may be defined by the combination of a structure and a language, both of which are too general to define a tractable class on their own.

In particular, a CSP can be considered as a pair of algebraic structures called *relational structures.*

Definition 3.4.1. *A* ***relational structure*** $\langle U, R_1, \ldots, R_m \rangle$ *consists of a set U called the* universe *and a list of* relations *over that universe.*

The first (or left hand side) relational structure denotes the hypergraph structure of the CSP, but with the additional restriction that the constraints whose scopes are in the same relation of the relational structure must also have the same constraint relation. The second (or right hand side) relational structure has the same signature (list of arities of the relations) and gives the constraint relations over these scopes. In this thesis we consider what restrictions need to be made to the left hand side to obtain tractability for given representations of the right hand side.

In the remainder of this chapter, we consider the structural properties of CSPs in terms of left hand side relational structures. A relational structure permitted by a CSP is a labelled ordered hypergraph of the constraint scopes: hyperedges with the same label having the same relation.

Definition 3.4.2. *CSP* $\langle V, D, C\rangle$ ***permits*** *structure* $\langle U, R_1, \ldots, R_m\rangle$ *exactly when* $U = V$ *and there is a partition* $C = C_1 \cup \cdots \cup C_m$ *where, for each* C_i*, every constraint in* C_i *has the same relation and* $R_i = \{\sigma \mid \langle \sigma, \rho\rangle \in C_i\}$*.*

A class $\mathcal{H}$ *of relational structures is tractable if the class of all instances permitting a structure in* $\mathcal{H}$ *is tractable.*

It will be convenient to consider the *structural hypergraph* of a relational structure, which just considers any *relational tuple* as the set of its components.

Definition 3.4.3. *For any tuple* t *with arity* r *we define* $\{t\} = \{t[1], \ldots, t[r]\}$*.*
For any relation ρ *we define* $\{\rho\} = \{\{t\} \mid t \in \rho\}$*. For any relational structure* $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ *we define* $\mathrm{hyper}(\mathbb{S}) = \bigcup_{i=1}^{m} \{R_i\}$ *and the* ***structural hypergraph*** $H(\mathbb{S}) = \langle U, \mathrm{hyper}(\mathbb{S})\rangle$*.*

Example 3.4.4. Consider the SAT instance whose implicit logical clauses are:

$v_1 \vee v_2$, $\overline{v_2} \vee \overline{v_3} \vee v_4$ and $v_1 \vee v_4$.

These three clauses may be represented as tuples by the following three constraints:

$$\langle\langle v_1, v_2\rangle, \{\langle T, T\rangle, \langle T, F\rangle, \langle F, T\rangle\}\rangle,$$
$$\langle\langle v_2, v_3, v_4\rangle, \{\langle T, T, T\rangle, \langle T, F, T\rangle, \langle T, F, F\rangle, \langle F, T, T\rangle, \langle F, T, F\rangle, \langle F, F, T\rangle, \langle F, F, F\rangle\}\rangle,$$
$$\langle\langle v_1, v_4\rangle, \{\langle T, T\rangle, \langle T, F\rangle, \langle F, T\rangle\}\rangle$$

This instance permits (at least) two structures: $\langle V, \{\langle v_1, v_2\rangle, \langle v_1, v_4\rangle\}, \{\langle v_2, v_3, v_4\rangle\}\rangle$ and $\langle V, \{\langle v_1, v_2\rangle\}, \{\langle v_1, v_4\rangle\}, \{\langle v_2, v_3, v_4\rangle\}\rangle$.

The structural hypergraph of these two relational structures has vertex set V and set of hyperedges $\{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3, v_4\}\}$. □

The relational structure of an instance captures precisely the fact that scopes may be of different types. Instead of having just one hyperedge relation we have several: one for each type of constraint. Here theory and practice meet.

We can now extend the hypergraph theory from Section 3.1 to the structural hypergraphs of relational structures.

Definition 3.4.5. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ be any relational structure.*

*The **Gaifman graph**, $G(\mathbb{S})$, of $\mathbb{S}$ has vertex set U. A pair of vertices v and w are an edge of $G(\mathbb{S})$ if there is a hyperedge of* $\mathrm{hyper}(\mathbb{S})$ *containing v and w.*

*Given an ordering $\langle v_1, \ldots, v_n \rangle$ of U the **induced graph** for this ordering is obtained from the Gaifman graph by processing the vertices, from last to first; when vertex v is processed, all its earlier neighbours are connected.*

*After this process, the **width** of any $v \in U$ is the number of its earlier neighbours. The width of $\mathbb{S}$, for this ordering, is the maximum width of any $v \in U$.*

*The **treewidth**,* $\mathrm{tw}(\mathbb{S})$*, of $\mathbb{S}$ is its minimal width over all orderings. For a class, $\mathcal{H}$, of relational structures, we denote by* $\mathrm{tw}(\mathcal{H})$ *the maximum treewidth of any structure in $\mathcal{H}$. We say* $\mathrm{tw}(\mathcal{H}) = \infty$ *if the treewidth is unbounded.*

Definition 3.4.6. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ be any relational structure and let $L(\mathbb{S}) = \{\langle i, t \rangle \mid i \in \{1, \ldots, m\}, t \in R_i\}$.*

*The **incidence graph**,* $\mathrm{IncG}(\mathbb{S})$*, of $\mathbb{S}$ is the bipartite graph, $\langle V, E \rangle$, where*

- *$V = U \cup L(\mathbb{S})$, and*
- *$E = \{\langle v, \langle i, t \rangle \rangle \mid v \in U, \langle i, t \rangle \in L(\mathbb{S}), v \in \{t\}\}$.*

*The **incidence width** of relational structure $\mathbb{S}$, denoted* $\mathrm{iw}(\mathbb{S})$*, is the treewidth of its incidence graph, that is,* $\mathrm{tw}(\mathrm{IncG}(\mathbb{S}))$.

Szeider [Sze03] uses incidence width to define tractable SAT classes. We shall revisit this result in more detail in Chapter 5 where we compare new tractable classes to these SAT classes.

Definition 3.4.7. *A relational structure $\mathbb{A} = \langle A, R_1, \ldots, R_m \rangle$ is a **substructure** of a relational structure $\mathbb{B} = \langle B, R'_1, \ldots, R'_m \rangle$ if $A \subseteq B$ and, for each i, $R_i \subseteq R'_i$.*

*A **homomorphism** from a relational structure $\mathbb{A} = \langle A, R_1, \ldots, R_m \rangle$ to a relational structure $\mathbb{B} = \langle B, R'_1, \ldots, R'_m \rangle$ is a mapping $h : A \to B$ such that for all i and all tuples $t \in R_i$ we have $h(t) \in R'_i$.*

A relational structure $\mathbb{S}$ *is a* ***core*** *if there is no homomorphism from* $\mathbb{S}$ *to a proper substructure of* $\mathbb{S}$*. A core of a relational structure* $\mathbb{S}$ *is a substructure* $\mathbb{S}'$ *of* $\mathbb{S}$ *such that there is a homomorphism from* $\mathbb{S}$ *to* $\mathbb{S}'$ *and* $\mathbb{S}'$ *is a core. It is well known that all cores of a relational structure* $\mathbb{S}$ *are isomorphic. Therefore, we often speak of the core,* $\mathrm{Core}(\mathbb{S})$*, of* $\mathbb{S}$*. For a class,* $\mathcal{H}$*, of relational structures, we denote by* $\mathrm{Core}(\mathcal{H})$ *the class of relational structures* $\{\mathrm{Core}(\mathbb{S}) \mid \mathbb{S} \in \mathcal{H}\}$*.*

Example 3.4.8. Consider the relational structures

$$\mathbb{S} = \langle U = \{0,1,2,3\}, R_0 = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 0,2\rangle, \langle 2,0\rangle, \langle 1,3\rangle, \langle 3,1\rangle, \langle 2,3\rangle, \langle 3,2\rangle\}\rangle,$$

$$\mathbb{S}' = \langle U = \{0,1,2\}, R_0' = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 0,2\rangle, \langle 2,0\rangle\}\rangle$$

and

$$\mathbb{S}'' = \langle U = \{0,1\}, R_0'' = \{\langle 0,1\rangle, \langle 1,0\rangle\}\rangle.$$

$\mathbb{S}'$ is a substructure of $\mathbb{S}$ as there is a homomorphism from $\mathbb{S}$ to $\mathbb{S}'$ with the mapping:

$$\{0 \to 0, 1 \to 1, 2 \to 2, 3 \to 0\}$$

However, $\mathbb{S}'$ is not a core as there is a homomorphism from $\mathbb{S}'$ to $\mathbb{S}''$ with the mapping:

$$\{0 \to 0, 1 \to 1, 2 \to 1\}$$

$\mathbb{S}''$ is a core as there is no homomorphism from $\mathbb{S}''$ to any proper substructure of $\mathbb{S}''$.

□

For bounded arity, the structural classes are precisely determined by the following theorem.

Theorem 3.4.9 (Corollary 19 of Grohe [Gro07])**.** *Assuming that W[1] is not FPT. For every recursively enumerable class* $\mathcal{H}$ *of relational structures*[1] *of bounded arity, the set of CSPs which permit a structure in* $\mathcal{H}$ *is tractable (for the extensional representation) if and only if* $\mathrm{tw}(\mathrm{Core}(\mathcal{H})) < \infty$*.*

[1] Non-recursively enumerable are of no practical interest.

3.5 Summary

In this chapter we have given a brief survey of current tractability results. We have seen that there have been two main approaches for identifying tractable classes: structure, in which properties of a CSPs underlying hypergraph are considered, and language, in which the relations used to express the constraints are considered. We have also seen that there is a hybrid approach, relational structure, in which a combination of both the hypergraph structure and the constraint relations may be used to define tractable classes.

In the real world, constraint practitioners will use a constraint representation which is appropriate for modeling the problem in hand, and the results given in this section do not always carry over to non-theoretical representations. The constraint representation required for a particular class of problems to have an efficient solution technique is critical to the uptake of any tractability result.

Chapter 4

Algorithmic Complexity Analysis Model

Chapter 5 contains proofs which rely on a detailed complexity analysis of several non-trivial algorithms. In order to provide consistent data structures with known operational complexities between these algorithms, they are analysed with respect to the framework presented in this chapter.

Analysing the computational complexity [CLRS09, End10, BBJ07, Fer09] of an algorithm is performed with respect to a Turing equivalent [Tur36] computational model. The Random Access Machine [CRR72] models the Von Neumann architecture [vN93], which is analogous to modern computers.

Definition 4.0.1. *A **Random Access Machine** (RAM) [CRR72] consists of an unbounded sequence of addressable registers capable of storing arbitrary integers, a processor capable of performing a set of elemental operations, an addressable list of elemental operations to perform, and a pointer to the current operation being executed.*

The set of registers is called the memory*, and the address of a register is an integer which may be stored as the value of another register. When complex data structures are stored over several contiguous registers, the address of the first register in the block is called the **base address** of the data structure. The processor contains a single internal register called the* accumulator *which is used for storing intermediate calculated values during processing.*

Each elemental operation requires some constant amount of time to perform, so are defined

*as taking **unit time**. The elemental operations allow us to consider the following to require unit time:*

- *Input or output of a single register.*
- *Moving the execution pointer to another position in the list.*
- *Moving the execution pointer to another position in the list only if the accumulator is 0.*
- *Reading from, or writing to, a single register.*
- *Integer arithmetic (addition, subtractions, multiplication and division).*

Under the RAM model, the complexity of the algorithm can be determined by considering the number of elemental operations required, as a function of the input size, to produce the expected output.

For readability and convenience, the algorithms presented in this thesis are expressed using pseudocode and complex data structures. This provides a higher level of abstraction than elemental operations on integer registers, and allows the general complexity of certain concepts common to several of the algorithms to be considered in advance.

Proposition 4.0.2. *Let D be a data structure of size at most d. Reading D from memory is $O(d)$ time, and writing D to memory is $O(d)$ time.*

Proof. By Definition 4.0.1, reading from or writing to a single register takes unit time. When stored in memory, D occupies d contiguous registers, so d individual reads or writes must be performed to either store or retrieve D. □

Comparing two complex data structures can be performed by comparing them pointwise.

Proposition 4.0.3. *Let D_A and D_B be two data structures of size at most d, and let b_A and b_B be their respective base addresses. Equality comparison of D_A and D_B is $O(d)$ time.*

Proof. By Definition 4.0.1, D_A is stored in a contiguous block of registers from b_A to $b_A + d - 1$, and D_B in a contiguous block of registers from b_B to $b_B + d - 1$. Equality between each pair $b_A + i$ and $b_B + i$ can be performed in unit time for $i = 0 \ldots d - 1$. □

An array is a complex data structure which may be used as a container for other complex data structures. So that the required contiguous block of registers can be allocated, it is required that both the maximum number of elements in the array, and the maximum size of the data structures it contains, are known in advance.

Definition 4.0.4. *Let D be a data structure with maximum size d, and let x be an integer. An **array** containing at most x elements of type D is constructed using a contiguous block of $d.x + 3$ registers as follows: The first three registers are reserved for the current number of elements held in the array, the maximum number of elements the array can hold (x), and the maximum size of each element (d) respectively. The remaining registers are used to store elements of type D such that the base address of the element in the i^{th} position is $d.i + 3$ relative to the base address of the array. The space required to store an array is $O(d.x)$.*

New elements are always inserted into the first available position. Whenever an element that is not in the last occupied position is removed, the element from the last occupied position is moved into the position vacated by the element being removed.

As we are only required to initialise the first three registers of an array, we can construct an empty array of arbitrary size in constant time.

Proposition 4.0.5. *Let D be a data structure of size at most d, and let x be an integer. Constructing an array capable of containing at most x elements of size at most d is $O(1)$ time.*

Proof. Dy Definition 4.0.1 of a RAM, it takes unit time for each write of 0, x and d to the first three registers respectively. □

Because a maximum data structure size is imposed, addressing elements of the array can be performed in constant time.

Proposition 4.0.6. *Let D be a data structure with maximum size d, and A be an array containing elements of type D. Addressing the element occupying the i^{th} position of A is $O(1)$ time.*

Proof. By Definition 4.0.4 the base address of an element can be calculated from its index using integer arithmetic, and by Definition 4.0.1 for a RAM, integer arithmetic operations can be performed in constant time. □

The array has been defined such that its parameters and current number of elements are explicitly stored. It can be seen that doing this only adds a unit time operation to each offset calculation, but may provide optimisations where these values do not need to be inferred. As the array maintains the condition that if there are x elements, then they occupy the first x positions, the the next available position to insert at can always be determined from the stored count of the current number of elements (rather than needing to perform a scan of the array).

Proposition 4.0.7. *Let D be a data structure with maximum size d, and A be an array containing elements of type D. Let b_A be the base address of A, and let c be the count of the current number of elements in A, stored at b_A. Determining the base address for the insertion of a new element of type D can be performed in time $O(1)$*

Proof. As by Definition 4.0.4 elements are inserted into the first unoccupied position, the base address of the next available position can be determined using integer arithmetic as $b_A+c.d+3$. By Definition 4.0.1 for a RAM, integer arithmetic operations can be performed in constant time. □

As determining the array position into which a new element will be inserted is constant time, the cost of insertion will always be the cost of writing the element data structure.

Proposition 4.0.8. *Let D be a data structure with maximum size d, and A be an array containing elements of type D. Inserting an element of type D into A can be performed in time $O(d)$.*

Proof. The result follows from Proposition 4.0.7 that it requires $O(1)$ time to locate the base register at which to write the element, and Proposition 4.0.2 that it requires time $O(d)$ to write a data structure of type D. By Definition 4.0.1 for a RAM, the integer arithmetic required to increment the array membership counter requires $O(1)$ time. □

Moving the last element to take the position of one being deleted requires time. However, if a count of the current number of elements was not maintained, then insertion would requires scanning the array for the next free position. This means that deleting would have to overwrite the affected registers with zeros in order to 'free' the position, which would require the same order of time as moving an element.

Proposition 4.0.9. *Let D be a data structure with maximum size d, and A be an array containing elements of type D. Deleting the element at position i can be performed in time $O(d)$.*

Proof. By Definition 4.0.4, determining the position of the last element in the array is $O(1)$ time. If i is not the last element, then the last element is written over the i^{th} element. By Proposition 4.0.6, addressing the last element is $O(1)$ time, and by Proposition 4.0.2 both reading the data structure from the last position and writing it to the i^{th} position is $O(d)$ time. By Definition 4.0.1 for a RAM, the integer arithmetic required to decrement the array membership counter is $O(1)$ time. □

In the algorithms considered in this thesis, projection of an array onto a subset of its positions will be required.

Proposition 4.0.10. *Let D be a data structure with maximum size d, and A be an array containing elements of type D. Given an array of at most x integer values representing indices of A, constructing A' as the projection of A onto these positions in the given order can be performed in $O(d.x)$.*

Proof. By Proposition 4.0.5, constructing an empty array that can contain at most x data structures of size at most d is $O(1)$ time. By Proposition 4.0.6 locating an element in A by index is $O(1)$ time. By Proposition 4.0.2 reading a data structure of size at most d is $O(d)$ time. By Proposition 4.0.8, inserting a data structure of size at most d into an array is $O(d)$ time. For each of the x positions, the value in A at that position is located and read, then inserted into A'. This therefore requires $O((1 + d + d).x)$, so $O(d.x)$ time. □

An array may be used to represent a set by enforcing that it may not contain two elements that are the same. Determining whether a candidate is already in the set requires scanning the array to determine whether it is already a member. This is equivalent to determining the position at which the element first occurs.

Proposition 4.0.11. *Let D be a data structure of size at most d, and let x be an integer. Determining the index of the first instance of an element in an array, A, containing at most x elements of type D requires $O(d.x)$ time.*

Proof. By Proposition 4.0.3, comparing any two elements of type D is $O(d)$. By Definition 4.0.1 for a RAM, performing the integer arithmetic required to increment an iterator over the elements of A requires unit time, and the elements stored in at most x sequential index locations will be compared to the target element. □

Sets may contain the same elements, but in a different order. For container data structures such as this, an equivalence comparison is required as the equality comparison considered in Proposition 4.0.3 would only evaluate to true for two equivalent sets if the members were recorded in the same order.

Proposition 4.0.12. *Let D be a data structure of size at most d, and let x_1, x_2 and x_{min} be integers. Let A_1 and A_2 be two set arrays containing at most x_1 and x_2 elements of type D respectively, and let x_{min} be the smaller of x_1 and x_2. Determining whether A_1 and A_2 contain the same elements requires $O\left({x_{min}}^2.d\right)$ time.*

Proof. A_1 and A_2 contain the same members if they both contain the same number of elements and each element of A_1 is in A_2. If this is true, then both A_1 and A_2 cannot contain more than

$x_{\min}$ elements (being the maximum size of the smaller array). To determine whether each element of A_1 is in A_2 it is necessary to compare each of the at most $x_{\min}$ elements of A_1 against each of the at most $x_{\min}$ elements of A_2. This is ${x_{\min}}^2$ comparisons, each of which takes $O(d)$ time by Proposition 4.0.3, so is $O\left({x_{\min}}^2.d\right)$ time. □

Several of the algorithms analysed in this thesis contain set union operations of the type $X \leftarrow X \cup Y$, that is a set union in which one of the input sets is replaced with the result. If the elements of each set are of the same type, and we know that the maximum number of elements in the result set may never be greater than the maximum number of elements in the replaced set, then we may perform the set union operation in place without having to allocate a new array.

Proposition 4.0.13. *Let S_A and S_B be two sets containing at most x_A and x_B elements of size at most d respectively. If $\max(|S_A \cup S_B|) = \max(|S_A|)$, then inserting the required elements from S_B into S_A such that S_A becomes the union of S_A and S_B ($S_A \leftarrow S_A \cup S_B$) is $O\left(x_B.x_A.d^2\right)$*

Proof. By Proposition 4.0.11, scanning S_A to determine membership of an element is $O(d.x_A)$ time. At most, this is performed for x_B elements in S_B, and in the worst case each element will be inserted into w_A, which by Proposition 4.0.8 requires $O(d)$ time. □

Similarly, several of the algorithms in this thesis contain set difference operations of the type $X \leftarrow X \setminus Y$, that is a set difference in which one of the input sets is replaced with the result. Again, this operation may be performed in place.

Proposition 4.0.14. *Let S_A and S_B be two sets containing at most x_A and x_B elements of size at most d respectively. Removing the required elements from S_A such that it becomes the set difference of S_A and S_B ($S_A \leftarrow S_A \setminus S_B$) is $O\left(x_B.x_A.d^2\right)$*

Proof. By Proposition 4.0.11, scanning S_A to determine membership of an element is $O(d.x_A)$. At most, this is performed for x_B elements in S_B and in the worst case each iteration will require an element from S_A to be deleted. By Proposition 4.0.9, deleting a data structure of size d from an array requires time $O(d)$. □

An associative array can be used to provide a dictionary data structure in which one data structure may be indirectly addressed by the value of another.

Definition 4.0.15. *Let D_k be a data structure with maximum size d_k, let D_v be a data structure with maximum size d_v, and let x be an integer. An **associative array** mapping at most x keys of type D_k to values of type D_v is constructed using a contiguous block of $(d_k.x + 3) + (d_v.x + 3)$*

registers as follows: The first $d_k.x + 3$ registers starting from the base address contain an array, A_k, with x elements of type D_k, and the remaining $d_v.x + 3$ registers contain an array, A_v, with x elements of type D_v. For each index position $i = 0 \ldots x - 1$, the key $A_k[i]$ is associated with the value $A_v[i]$. Given the index position of a key in A_k, a_k, the base address of the corresponding value in A_v as an offset from the base address of the associative array is $(d_k.x + 3) + (d_v.a_k + 3)$.

The space required to store the key and value arrays is $O(d_k.x)$ and $O(d_v.x)$ respectively, so the space required to store an associative array is $O(x.(d_k + d_v))$.

As an associative array is defined in terms of a key and value array, its construction is no harder than constructing two arrays.

Proposition 4.0.16. *Let D_k be a data structure of size at most d_k, D_v be a data structure of size at most d_v, and let x be an integer. Constructing an empty associative array capable of containing at most x mappings from elements of size at most d_k to elements of size at most d_v is $O(1)$.*

Proof. By definition, an associative array is constructed from two arrays. By Proposition 4.0.5, constructing each empty array is $O(1)$, so constructing an empty associative array is also $O(1)$. □

Addressing a value in an associative array requires finding the position of the key in the key array.

Proposition 4.0.17. *Let D_k be a data structure with maximum size d_k, and let D_v be a data structure with maximum size d_v. Let M be an associative array mapping at most x keys of type D_k to values of type D_v. Given a data structure of type D_k, finding the base address of the corresponding value in M is $O(d.x)$.*

Proof. By Proposition 4.0.11, finding the index position of the given key in M is $O(d.x)$. By Definition 4.0.15 for an associative array, finding the base address of a value given the index position of the key can be performed in unit time using integer arithmetic. □

The associative array is reliant in the fact that insertions into both the key and value arrays occur at the same position. As both arrays must always contain the same number of elements, this is trivially maintained by the insertion and deletion properties of the arrays. As such, inserting into an associative array is exactly equivalent to performing an insertion into the key array and an insertion into the value array.

Proposition 4.0.18. *Let D_k be a data structure with maximum size d_k, and let D_v be a data structure with maximum size d_v. Let M be an associative array mapping at most x keys of type*

D_k to values of type D_v. Inserting a key value pair where the key is not already in the key array requires $O(d_k + d_v)$ time.

Proof. By Definition 4.0.4 for an array, all new insertions are performed to the next available position. By Definition 4.0.15 for an associative array, both the key and value arrays contain the same number of elements at any given time, so inserting into the next available position for both maintains the correct structure. By Proposition 4.0.8 inserting into the key arrays will take time $O(d_k)$, and into the value array will take time $O(d_v)$. □

A common data structure used by the algorithms in this thesis as that of an associative array which maps from some data structure to and array of some other data structure. As such, it is convenient to consider the common initialisation case where a key is inserted and the corresponding value is initialised as an empty array at the same time.

Proposition 4.0.19. *Let D_k be a data structure with maximum size d_k, and let A_v be an array data structure with maximum size d_v. Let M be an associative array mapping at most x keys of type D_k to values of type A_v.*

Inserting a key of type D_k and corresponding value of type A_v into M requires time $O(d_k)$ when the key does not already exist in the key array and the value value is an empty set of type A_v.

Proof. By Definition 4.0.15 for an associative array, the insertion position in the value array will be the same as for the key array. By Definition 4.0.4 for an array, the next insertion position can be determined from the current membership count. By Proposition 4.0.8, inserting into the key array requires time $O(d_k)$. The base address for the new value can be determined from the base address of the newly inserted key, and the counter of the value array incremented, using integer arithmetic. By Proposition 4.0.5 constructing an empty array of type A_v requires time $O(1)$. □

Chapter 5

Interaction Width

Most theoretical structural tractability results, such as those detailed in Chaper 3, tacitly rely on the fact that constraint relations are listed explicitly. However, since table constraints have such poor propagation algorithms, practitioners prefer to use more succinct implicit constraint representations such as global constraints [vHK06] or SAT clauses [GW02, CV12].

The important theorem which states that the class of instances with acyclic hypergraphs is tractable [BFMY83] is not true for implicit representations. The class of CSPs which contain an 'anything goes' constraint over all variables is tractable when listed explicitly because the 'anything goes' constraint dominates the size of the instance, but this constraint has only a small constant size in an implicit representation. This is an anomaly, a break between theory and practice, which must be addressed.

The discrepancy between the theoretical and practical world that is caused by this anomaly is not just limited to border cases. In fact, when naturally represented as clauses, the acyclicity result does not even hold for a class as important as SAT since any SAT instance can be reduced in polynomial time to a pair of acyclic SAT instances.

Example 5.0.20. Given any SAT instance P we construct two acyclic SAT instances:
- Add the universal clause disallowing 'all F' to P to build the acyclic instance P_F.
- Add the universal clause disallowing 'all T' to P to build the acyclic instance P_T.

If P has a solution, then at least one of the two acyclic instances, P_F and P_T, will have a solution. Conversely, if at least one of the two acyclic instances has a solution, then this will be a solution to the original instance P. □

In this chapter we begin to address the discrepancies between theory and practice by developing theory that gives structural tractability results for some implicit representations.

Binary Decision Diagrams [Lee59] are rooted directed acyclic graphs which may be used to represent Boolean functions, such as SAT clauses. Each internal node represents a variable, and has two edges representing an assignment of either TRUE or FALSE. The leaves indicate whether the clause represented by the path from the root to the leaf is a valid assignment. Bryant developed a more concise representation, known as Ordered Binary Decision Diagrams [Bry92], in which the variable order is fixed and the graph is reduced such that there are only two leaves indicating validity and the graph for any given function and variable ordering is unique. In [US94], Uribe and Stickel provide an experimental comparison of the performance of solving SAT problems using Ordered Binary Decision Diagrams verses the standard Davis-Putnam algorithm [DLL62] commonly used in SAT solvers.

The Trie data structure (first described by de Kleer in [dK92]) is another reduced graph structure that van be used to concisely represent Boolean functions. In the Trie structure each internal node is a clause operator (such as $\wedge$ or $\vee$), and each edge is labeled with a set of literals and negated literals such that each clause is represented by some path from the root node to a leaf node. Zhang and Stickel show how the Davis-Putnam algorithm can be implemented using Tries in [ZS94]

More recently, there has been a series of papers describing more succinct representations of higher order constraints and attempts to improve their applicability in constraint solvers.

A known compression algorithm for the positive representation has been used in the literature in the form of *compressed tuples* [FM01, KB05]. Katsirelos and Walsh [KW07] consider the feasibility of *Generalised Arc Consistency* [BR97, Mac77] on this compressed representation.

Chen and Grohe [CG06] described the *GDNF* representation which is exactly the compressed tuples representation. They also described a more succinct *Decision Diagram* representation which extends Ordered Binary Decision Diagrams to higher order constraints. In both cases they identify precisely the tractable structural classes.

We shall consider a simple extensional representation which naturally expresses SAT. In particular, we consider the analogous negative extensional representation and a mixed representation which allows both the positive and negative extensional representations.

We shall introduce a new width measure called *interaction width* and show that by bounding this we are able to convert instances from the mixed representation into the positive representation in polynomial time. We shall show that this allows us to generate new structurally tractable classes.

We shall extend the current tractability results for SAT by comparing our classes with a result by Szeider [Sze03] and showing that they are incomparable. A similar comparison to the GDNF results of Chen and Grohe [CG06] show that our results are also incomparable. By defining interaction width on relational structures we develop a complete dichotomy for the so called *merged* structure in the mixed representation.

5.1 Structural Observations

As previously stated in Chapter 3, the treewidth of a CSP's hypergraph is often seen as a measure of its complexity.

Example 5.1.1. The hypergraph of a CSP consisting of a single constraint of arity n has a treewidth of $n-1$, and the hypergraph of a CSP consisting of several constraints which make up a clique over n variables also has a treewidth of $n-1$. However, while the CSP with a single constraint is clearly very easy to solve when presented using an explicit positive representation, the CSP whose constraints form a clique is potentially much harder to solve. □

The treewidth of a CSP's hypergraph is based on the connectivity of variables, so has its lowest possible value determined by the largest constraint scope. However, we have just demonstrated that a high arity does not imply high complexity of solution.

Some structural classes use reduced structures to generate results. Recall Theorem 3.4.9 which gives the dichotomy result by Grohe [Gro07]. This result is defined by the treewidth of the core of the structure.

For a given instance, the treewidth of its core may be smaller than that of its original structure. However, structural homomorphisms, which define the core, do not affect the arity of the resulting constraints. For example, the CSP consisting of a single constraint is its own core.

We observe that we might reduce the treewidth of a CSP's structure by merging variables so that the arity of the largest constraint scope is reduced. However, this merge must be performed in polynomial time in order to generate useful complexity results.

Example 5.1.2. Consider a CSP consisting of a single constraint over n variables. The n variables can be replaced with a single variable whose domain is the set of tuples in the original constraint relation. The original constraint is then replaced with a single unary constraint on the new variable which allows all values in its domain. This merged version of the CSP has a treewidth of 0. □

In this example, each variable only occurs in the scope of the single constraint. The natural extension to CSPs with more than a single constraint is to consider merging those variables which

are members of precisely the same set of constraint scopes. As such, the effects that the interactions between these constraints have on the complexity of the merging process must be examined.

The interactions between constraints have been considered before, for example in the dual hypergraph [Dec03]. However, our observation is that the constraint representation is also critical to applying this structural compression.

Merging variables which share the same interaction cannot affect the edge-based notions of structural width, such as hypertree width. In fact, it cannot alter the tractability at all because it is also possible to *unmerge* the merged variables in polynomial time. Therefore, merging variables only aims to simplify and describe properties of structures, namely similarities between variables.

Example 5.1.3. Consider the class of CSPs consisting of a single constraint. The hypergraph structure of any CSP in this class has a hypertree width of 1 and so this class is easy to solve. The merged structure of these hypergraphs has a single vertex, and thus a treewidth of 0, so are also easy to solve. However, the class of hypergraphs with a hypertree width of 1 contains all acyclic hypergraphs, yet the merged structure can have arbitrary treewidth and so the treewidth of the merged structure is not as general as the hypertree width because the merged structure must also be tractable. (In fact, they are still acyclic.) □

So, whilst merging variables cannot alter complexity, it does allow different views of the same tractable classes. It is not clear what benefit this brings to the positive extensional representation, but it does allow the same simplification to be applied to other extensional representations for which far fewer tractable classes are known.

Existing structural tractability results may be applied to classes of constraints under any representation for which there is a polynomial time transformation to the positive extensional representation required for the result, provided that the applicable structural properties are not altered. In this chapter we consider a particular succinct representation for which this transformation is not polynomial time in general. We define a new structural property, called *interaction width*, which when bounded allows us to partition the variables of an instance given in this representation in such a way that the instance produced by merging the variables in each of these sets not only allows for a polynomial time transformation into the positive extensional representation, but also preserves existing structural properties.

5.2 Tractability with Respect to Representation

Constraint relations are often expressed by listing all allowed assignments. For finite CSPs, it is also possible to express a relation by listing all disallowed assignments.

Definition 5.2.1. *Let $\langle V, D, C\rangle$ be a CSP, and let $c = \langle \sigma, \rho\rangle$ be a constraint of arity r from C.*

*The **positive** (or* extensional*) **representation** of c is a list of the variables in σ followed by the tuples in ρ.*

*The **negative representation** of c is a list of the variables in σ followed by the tuples in $D(\sigma[1]) \times \ldots \times D(\sigma[r])$ that are not in ρ.*

*The **mixed representation** of c is a Boolean flag, which if T is followed by the positive representation of c, and if F by the negative representation.*

*The **GDNF representation** [CG06] of c is a list of the variables in σ followed by a list of expressions of the form $A[1] \times \cdots \times A[r]$ where ρ is the union of these set products.*

In the negative (and hence the mixed) representations, domain values which do not occur as literals in any forbidden tuple could appear in solutions. All such (missing) literals in any domain are equivalent in the sense that they are interchangeable in any solution. As a general method, the symbol + is used to denote a domain which contains some value(s) that may occur in a solution, but which do not occur explicitly in the chosen representation. Conversely, − is used to denote a domain in which every value that may occur in a solution also occurs explicitly in the representation.

The encoding of a CSP may now be described using any suitable constraint representation.

Definition 5.2.2. *Let Θ be any constraint representation. A Θ **instance** encoding is comprised of a list of variables, the list of domain types corresponding to these variables (in the same order), and a list of constraints represented using Θ.*

Example 5.2.3. Recall the SAT instance from Example 3.4.4 whose logical clauses are: $v_1 \vee v_2$, $\overline{v_2} \vee \overline{v_3} \vee v_4$ and $v_1 \vee v_4$.

A positive representation of this instance is the following:

$$
\begin{array}{c}
[v_1, v_2, v_3, v_4][-,-,-,-] \\
\langle v_1, v_2\rangle, \langle T,T\rangle, \langle T,F\rangle, \langle F,T\rangle \\
\langle v_2, v_3, v_4\rangle, \langle T,T,T\rangle, \langle T,F,T\rangle, \langle T,F,F\rangle, \langle F,T,T\rangle, \langle F,T,F\rangle, \langle F,F,T\rangle, \langle F,F,F\rangle \\
\langle v_1, v_4\rangle, \langle T,T\rangle, \langle T,F\rangle, \langle F,T\rangle
\end{array}
$$

A negative representation of this instance is the following:

$$
\begin{aligned}
&[v_1, v_2, v_3, v_4][+, -, +, +] \\
&\langle v_1, v_2 \rangle, \langle F, F \rangle \\
&\langle v_2, v_3, v_4 \rangle, \langle T, T, F \rangle \\
&\langle v_1, v_4 \rangle, \langle F, F \rangle
\end{aligned}
$$

A mixed representation of this instance may be the following:

$$
\begin{aligned}
&[v_1, v_2, v_3, v_4][-, -, +, +] \\
&\quad T, \langle v_1, v_2 \rangle, \langle T, T \rangle, \langle T, F \rangle, \langle F, T \rangle \\
&F, \langle v_2, v_3, v_4 \rangle, \langle T, T, F \rangle \\
&\quad F, \langle v_1, v_4 \rangle, \langle F, F \rangle
\end{aligned}
$$

A GDNF representation of this instance may be the following:

$$
\begin{aligned}
&[v_1, v_2, v_3, v_4][-, -, -, -] \\
&\quad \langle v_1, v_2 \rangle, \{T\} \times \{F, T\}, \\
&\qquad \{F, T\} \times \{T\} \\
&\quad \langle v_2, v_3, v_4 \rangle, \{F\} \times \{F, T\} \times \{F, T\}, \\
&\qquad \{F, T\} \times \{F\} \times \{F, T\}, \\
&\qquad \{F, T\} \times \{F, T\} \times \{T\} \\
&\quad \langle v_1, v_4 \rangle, \{T\} \times \{F, T\}, \\
&\qquad \{F, T\} \times \{T\}
\end{aligned}
$$

□

In order to refer to a class of instances under one of these representations, we define the following:

Definition 5.2.4. *For any class $\mathcal{C}$ of CSPs we denote by:*

- Positive($\mathcal{C}$) *the set of positive representations of these instances.*
- Negative($\mathcal{C}$) *the set of negative representations of these instances.*
- Mixed($\mathcal{C}$) *the set of mixed representations of these instances.*
- GDNF($\mathcal{C}$) *the set of GDNF representations of these instances.*

We can now consider the relative tractability of classes of instances under these representations.

Definition 5.2.5. *For a representation, Θ, we say that a class $\mathcal{T}$ of Θ instances is **tractable** if there exists a solution algorithm that runs in time polynomial in the size of the input Θ instance.*

*A representation Θ is **more structurally tractable** than a representation Θ' if every structural class of CPSs that is Θ' tractable is also Θ tractable.*

As complexity of solution is defined relative to the input size, the relative succinctness of representations is important.

Definition 5.2.6. *Let Θ and Θ' be two methods for representing CSPs. We say that Θ' is **as succinct** as Θ if there is a polynomial p for which, given any instance P represented in Θ with size $|P|_\Theta$ there exists a representation of P in Θ' of size at most $p(|P|_\Theta)$.*

Corollary 5.2.7. *Let Θ' be a representation that is as succinct as a representation Θ and let there be a polynomial time reduction from Θ to Θ'. Any class which is structurally tractable for Θ' is also structurally tractable for Θ.*

In other words, where polynomial time reductions are known and as the representation becomes (strictly) more succinct, the tractable structural classes become smaller. This allows us to make a very important observation following Grohe's theorem (Theorem 3.4.9).

Corollary 5.2.8. *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be a recursively enumerable class of relational structures with bounded arity.* Negative($\mathcal{H}$), Mixed($\mathcal{H}$) *and* GDNF($\mathcal{H}$) *are tractable for any bounded domain size if and only if* $\mathrm{tw}(\mathrm{Core}(\mathcal{H})) < \infty$.

Proof. For bounded arity and domain size, the positive representation is as succinct as each of these three other representations and polynomial time reductions from these to the positive representation exist. □

For any succinct representation, it is thus only classes of structures with unbounded arity, or classes of CSPs with unbounded domain size, whose tractability must still be characterised. In the case of GDNF, such a characterisation has been found [CG06].

Theorem 5.2.9 (Theorem 14 of Chen and Grohe [CG06])**.** *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be a recursively enumerable class of relational structures. Then* GDNF($\mathcal{H}$) *is tractable if and only if the cores of the structures in $\mathcal{H}$ have bounded incidence width (recall Definition 3.4.6).*

5.3 Interaction Width of Relational Structures

In earlier work, we defined Interaction Width in relation to hypergraphs by considering the effects of a mixed representation on structural decompositions [HCG06].

The following example shows that when considering succinct representations it is not enough to consider just the hypergraph structure.

Example 5.3.1. Consider any class $\mathcal{H}$ of hypergraphs of unbounded arity. The class includes the hypergraph structure of each of the relational structures $\{\mathbb{A}_i \mid i = 1, 2, \ldots\}$, where $\mathbb{A}_n$ has universe $\{1, \ldots, n\}$ and relations $R_1, \ldots, R_n$, where each $R_i = \{\langle 1, \ldots, n\rangle\}$. Since $\mathbb{A}_n$ is a core and has incidence width precisely n we know, by Theorem 5.2.9, that the class GDNF($\mathcal{H}$) is intractable.

On the other hand, consider the class of relational structures $\{\mathbb{B}_i \mid i = 1, 2, \ldots\}$, where $\mathbb{B}_n$ has universe $\{1, \ldots, n\}$ and just one n-ary relation containing just one tuple. This class of relational structures has unbounded arity but the set of GDNF instances permitting some structure in this class is tractable.

So, just considering hypergraph structure there are no tractable structural classes of unbounded arity. By considering relational structure we find many tractable structural classes. □

To compare our results with those presented by Chen and Grohe for GDNF [CG06], whose results relate to the cores of the structures, we now define Interaction Width with respect to relational structures [CGH09].

When considering only the hypergraph of a CSP, a region is a maximal set of variables with the following property: No hypergraph edge contains some, but not all, of these variables. By considering the relational structure, these regions may be further refined.

The notion of Interaction Width is the same for both versions: the maximum number of distinct regions occurring on any constraint in the CSP.

Definition 5.3.2. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ be a relational structure. We say that $\{v', w'\} \subseteq U$ is $\mathbb{S}$-**similar** to $\{v, w\} \subseteq U$ if there exists some $R_i \in \{R_1, \ldots, R_m\}$ and $t, t' \in R_i$ where for some j, k we have that $t\,[j,k] = \langle v, w \rangle$ and $t'\,[j,k] = \langle v', w' \rangle$.*

For the relational structure $\mathbb{S}$ with universe U, $\mathbb{S}$-similar is a symmetric binary relation over the set of unordered pairs from U. For each pair of variables in the universe, it defines the set of pairs with which they can be interchanged in at least one relation. Trivially, every pair of variables that both occur together in a tuple of some relation is $\mathbb{S}$-similar-related to itself.[1]

Example 5.3.3. Let R_1, R_2 and R_3 be three relations over the universe $U = \{v_a, v_b, v_c, v_d, v_e, v_f, v_g, v_h, v_i, v_j, v_k\}$:

$$R_1 = \{\langle v_a, v_b, v_c, v_d \rangle, \langle v_e, v_f, v_c, v_d \rangle\}$$
$$R_2 = \{\langle v_e, v_f, v_g, v_h \rangle, \langle v_i, v_j, v_g, v_h \rangle\}$$
$$R_3 = \{\langle v_j, v_k \rangle\}$$

Now let $\mathbb{S}_A$ and $\mathbb{S}_B$ be two relational structures:

$$\mathbb{S}_A = \langle U \setminus \{v_k\}, R_1, R_2 \rangle$$
$$\mathbb{S}_B = \langle U, R_1, R_2, R_3 \rangle$$

In $\mathbb{S}_A$, $\{v_a, v_b\}$ is $\mathbb{S}_A$-similar to $\{v_e, v_f\}$ because R_1 contains $\langle v_a, v_b, v_c, v_d \rangle$ and $\langle v_e, v_f, v_c, v_d \rangle$. Likewise, in $\mathbb{S}_B$, $\{v_a, v_b\}$ is also $\mathbb{S}_B$-similar to $\{v_e, v_f\}$. □

The $\mathbb{S}_A$-similar and $\mathbb{S}_B$-similar relations for $\mathbb{S}_A$ and $\mathbb{S}_B$ from Example 5.3.3 are shown in Figure 5.1 on page 65.

Definition 5.3.4. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ be a relational structure. The equivalence relation $\mathbb{S}$-**equivalent** is the transitive closure of $\mathbb{S}$-similar.*

For a relational structure $\mathbb{S}$, the $\mathbb{S}$-equivalent relation partitions all possible unordered pairs from the universe into disjoint equivalence classes. We will enforce the condition that if a pair of variables in some class are in the same interaction region, then this must also be true for all other pairs of variables in that class.

The $\mathbb{S}_A$-equivalent and $\mathbb{S}_B$-equivalent relations for $\mathbb{S}_A$ and $\mathbb{S}_B$ are shown in Figure 5.2.

[1] This also holds, without affecting the result, if we allow the unordered pairs to include the multisets where $v = w$.

Definition 5.3.5. *For any relational structure* $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ *we define, for any vertex* $v \in U$, $\tau(v)$ *to be* $\{\langle i, t \rangle \mid t \in R_i, v \in \mathrm{set}(t)\}$.

For each variable in the universe of a relational structure, the τ function gives a maximal set of pairs consisting of: a tuple from a relation of which the variable is a member, and an identifier telling us which relation this tuple is a member of. For $\mathbb{S}_A$ of Example 5.3.3, $\tau_A(v_a)$ would be the set $\{\langle 1, \langle v_a, v_b, v_c, v_d \rangle \rangle\}$. Two variables can only be in the same region if their τ functions are the same, that is, they must always appear together in the same tuples of the same relations in the relational structure. As $\tau_A(v_b)$ is also $\{\langle 1, \langle v_a, v_b, v_c, v_d \rangle \rangle\}$ in $\mathbb{S}_A$, v_a and v_b may be members of the same region provided that all other conditions are met.

The results of the τ functions as applied to the variables in $\mathbb{S}_A$ and $\mathbb{S}_B$ are shown in Figure 5.3.

Definition 5.3.6. *Let* $\mathbb{S}$ *be a relational structure over universe* U. *We say that* $v \in U$ *and* $w \in U$ *are* ***τ-equivalent*** *if either* $v = w$ *or, for every set* $\{v', w'\} \subseteq U$ *which is* $\mathbb{S}$*-equivalent-related to* $\{v, w\}$, *we have* $\tau(v') = \tau(w')$. *This is an equivalence relation for* U *and we denote by* $\mathrm{IntReg}(v)$ *the* ***interaction region****, or* τ*-equivalent class, of* v.

Figures 5.2 and 5.3 show that in $\mathbb{S}_A$, the pair $\{v_a, v_b\}$ is $\mathbb{S}_A$-equivalent-related to $\{v_a, v_b\}$, $\{v_e, v_f\}$, and $\{v_i, v_j\}$ and that the τ_A function applied to both v_a and v_b is $\{1, \langle v_a, v_b, v_c, v_d \rangle\}$. As the τ_A function also matches for v_e and v_f, and v_i and v_j, it follows that v_a and v_b are in the same interaction region. However, in $\mathbb{S}_B$, v_a and v_b are not in the same interaction region. Due to $\mathbb{S}_B$-equivalent, we would require that the τ_B function matches on v_i and v_j, which is not the case.

The interaction regions for $\mathbb{S}_A$ and $\mathbb{S}_B$ are shown in Figure 5.4.

Analogous to the hypergraph definition, the interaction width of a relational structure is the maximum number of regions occurring on any relational tuple.

Definition 5.3.7. *Let* $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ *be a relational structure over universe* $U = \{v_1, \ldots, v_k\}$. *The* ***interaction width*** *of* $\mathbb{S}$ *is:*

$$\mathrm{intw}(\mathbb{S}) = \max_{\langle v_1, \ldots, v_k \rangle \in R_i}^{i=1,\ldots,m} |\{\mathrm{IntReg}(v_1), \ldots, \mathrm{IntReg}(v_k)\}| \quad .$$

For a class, $\mathcal{H}$, *of relational structures, we denote by* $\mathrm{intw}(\mathcal{H})$ *the maximum interaction width of any structure in* $\mathcal{H}$. *We say that* $\mathrm{intw}(\mathcal{H}) = \infty$ *if the interaction width is unbounded.*

There are many similarities between $\mathbb{S}_A$ and $\mathbb{S}_B$. Let $\mathbb{S} \in \{\mathbb{S}_A, \mathbb{S}_B\}$. We have:

$\{v_a, v_b\}$	is $\mathbb{S}$-similar-related to	$\{v_a, v_b\}$ and $\{v_e, v_f\}$
$\{v_a, v_c\}$	...	$\{v_a, v_c\}$ and $\{v_c, v_e\}$
$\{v_a, v_d\}$	...	$\{v_a, v_d\}$ and $\{v_d, v_e\}$
$\{v_b, v_c\}$	...	$\{v_b, v_c\}$ and $\{v_c, v_f\}$
$\{v_b, v_d\}$	...	$\{v_b, v_d\}$ and $\{v_d, v_f\}$
$\{v_c, v_d\}$	...	$\{v_c, v_d\}$
$\{v_c, v_e\}$	...	$\{v_a, v_c\}$ and $\{v_c, v_e\}$
$\{v_c, v_f\}$	...	$\{v_b, v_c\}$ and $\{v_c, v_f\}$
$\{v_d, v_e\}$	...	$\{v_a, v_d\}$ and $\{v_d, v_e\}$
$\{v_d, v_f\}$	...	$\{v_b, v_d\}$ and $\{v_d, v_f\}$
$\{v_e, v_f\}$	...	$\{v_a, v_b\}$, $\{v_e, v_f\}$ and $\{v_i, v_j\}$
$\{v_e, v_g\}$	...	$\{v_e, v_g\}$ and $\{v_g, v_i\}$
$\{v_e, v_h\}$	...	$\{v_e, v_h\}$ and $\{v_h, v_i\}$
$\{v_f, v_g\}$	...	$\{v_f, v_g\}$ and $\{v_g, v_j\}$
$\{v_f, v_h\}$	...	$\{v_f, v_h\}$ and $\{v_h, v_j\}$
$\{v_g, v_h\}$	...	$\{v_g, v_h\}$
$\{v_g, v_i\}$	...	$\{v_e, v_g\}$ and $\{v_g, v_i\}$
$\{v_g, v_j\}$	...	$\{v_f, v_g\}$ and $\{v_g, v_j\}$
$\{v_h, v_i\}$	...	$\{v_e, v_h\}$ and $\{v_h, v_i\}$
$\{v_h, v_j\}$	...	$\{v_f, v_h\}$ and $\{v_h, v_j\}$
$\{v_i, v_j\}$	...	$\{v_e, v_f\}$ and $\{v_i, v_j\}$

In $\mathbb{S}_B$ only, $\{v_j, v_k\}$ is $\mathbb{S}_B$-similar-related to $\{v_j, v_k\}$.

Figure 5.1: The $\mathbb{S}_A$-similar and $\mathbb{S}_B$-similar relations for $\mathbb{S}_A$ and $\mathbb{S}_B$ from Example 5.3.3

There are many similarities between $\mathbb{S}_A$ and $\mathbb{S}_B$. Let $\mathbb{S} \in \{\mathbb{S}_A, \mathbb{S}_B\}$. We have:

$\{v_a, v_b\}$	is $\mathbb{S}$-equivalent-related to	$\{v_a, v_b\}$, $\{v_e, v_f\}$ and $\{v_i, v_j\}$
$\{v_a, v_c\}$	...	$\{v_a, v_c\}$ and $\{v_c, v_e\}$
$\{v_a, v_d\}$	...	$\{v_a, v_d\}$ and $\{v_d, v_e\}$
$\{v_b, v_c\}$	...	$\{v_b, v_c\}$ and $\{v_c, v_f\}$
$\{v_b, v_d\}$	...	$\{v_b, v_d\}$ and $\{v_d, v_f\}$
$\{v_c, v_d\}$	...	$\{v_c, v_d\}$
$\{v_c, v_e\}$	...	$\{v_a, v_c\}$ and $\{v_c, v_e\}$
$\{v_c, v_f\}$	...	$\{v_b, v_c\}$ and $\{v_c, v_f\}$
$\{v_d, v_e\}$	...	$\{v_a, v_d\}$ and $\{v_d, v_e\}$
$\{v_d, v_f\}$	...	$\{v_b, v_d\}$ and $\{v_d, v_f\}$
$\{v_e, v_f\}$	...	$\{v_a, v_b\}$, $\{v_e, v_f\}$ and $\{v_i, v_j\}$
$\{v_e, v_g\}$	...	$\{v_e, v_g\}$ and $\{v_g, v_i\}$
$\{v_e, v_h\}$	...	$\{v_e, v_h\}$ and $\{v_h, v_i\}$
$\{v_f, v_g\}$	...	$\{v_f, v_g\}$ and $\{v_g, v_j\}$
$\{v_f, v_h\}$	...	$\{v_f, v_h\}$ and $\{v_h, v_j\}$
$\{v_g, v_h\}$	...	$\{v_g, v_h\}$
$\{v_g, v_i\}$	...	$\{v_e, v_g\}$ and $\{v_g, v_i\}$
$\{v_g, v_j\}$	...	$\{v_f, v_g\}$ and $\{v_g, v_j\}$
$\{v_h, v_i\}$	...	$\{v_e, v_h\}$ and $\{v_h, v_i\}$
$\{v_h, v_j\}$	...	$\{v_f, v_h\}$ and $\{v_h, v_j\}$
$\{v_i, v_j\}$	...	$\{v_a, v_b\}$, $\{v_e, v_f\}$ and $\{v_i, v_j\}$

In $\mathbb{S}_B$ only, $\{v_j, v_k\}$ is $\mathbb{S}_B$-equivalent-related to $\{v_j, v_k\}$.

Figure 5.2: The $\mathbb{S}_A$-equivalent and $\mathbb{S}_B$-equivalent relations for $\mathbb{S}_A$ and $\mathbb{S}_B$ from Example 5.3.3

There are many similarities between $\mathbb{S}_A$ and $\mathbb{S}_B$. Let $\tau \in \{\tau_A, \tau_B\}$. We have:

$$\begin{array}{rcl}
\tau(v_a) & = & \{\langle 1, \langle v_a, v_b, v_c, v_d\rangle\rangle\} \\
\tau(v_b) & = & \{\langle 1, \langle v_a, v_b, v_c, v_d\rangle\rangle\} \\
\tau(v_c) & = & \{\langle 1, \langle v_a, v_b, v_c, v_d\rangle\rangle, \langle 1, \langle v_e, v_f, v_c, v_d\rangle\rangle\} \\
\tau(v_d) & = & \{\langle 1, \langle v_a, v_b, v_c, v_d\rangle\rangle, \langle 1, \langle v_e, v_f, v_c, v_d\rangle\rangle\} \\
\tau(v_e) & = & \{\langle 1, \langle v_e, v_f, v_c, v_d\rangle\rangle, \langle 2, \langle v_e, v_f, v_g, v_h\rangle\rangle\} \\
\tau(v_f) & = & \{\langle 1, \langle v_e, v_f, v_c, v_d\rangle\rangle, \langle 2, \langle v_e, v_f, v_g, v_h\rangle\rangle\} \\
\tau(v_g) & = & \{\langle 2, \langle v_e, v_f, v_g, v_h\rangle\rangle, \langle 2, \langle v_i, v_j, v_g, v_h\rangle\rangle\} \\
\tau(v_h) & = & \{\langle 2, \langle v_e, v_f, v_g, v_h\rangle\rangle, \langle 2, \langle v_i, v_j, v_g, v_h\rangle\rangle\} \\
\tau(v_i) & = & \{\langle 2, \langle v_i, v_j, v_g, v_h\rangle\rangle\}
\end{array}$$

$\mathbb{S}_A$:

$$\tau_A(v_j) \quad = \quad \{\langle 2, \langle v_i, v_j, v_g, v_h\rangle\rangle\}$$

$\mathbb{S}_B$:

$$\begin{array}{rcl}
\tau_B(v_j) & = & \{\langle 2, \langle v_i, v_j, v_g, v_h\rangle\rangle, \langle 3, \langle v_j, v_k\rangle\rangle\} \\
\tau_B(v_k) & = & \{\langle 3, \langle v_j, v_k\rangle\rangle\}
\end{array}$$

Figure 5.3: The τ functions for the variables of $\mathbb{S}_A$ and $\mathbb{S}_B$ in Example 5.3.3

The Regions of $\mathbb{S}_A$:

$$\begin{array}{lcl} \mathrm{IntReg}(v_a) = \mathrm{IntReg}(v_b) = \{v_a, v_b\} \\ \mathrm{IntReg}(v_c) = \mathrm{IntReg}(v_d) = \{v_c, v_d\} \\ \mathrm{IntReg}(v_e) = \mathrm{IntReg}(v_f) = \{v_e, v_f\} \\ \mathrm{IntReg}(v_g) = \mathrm{IntReg}(v_h) = \{v_g, v_h\} \\ \mathrm{IntReg}(v_i) = \mathrm{IntReg}(v_j) = \{v_i, v_j\} \end{array}$$

The Regions of $\mathbb{S}_B$:

$$\begin{array}{lcl} & \mathrm{IntReg}(v_a) = \{v_a\} \\ & \mathrm{IntReg}(v_b) = \{v_b\} \\ \mathrm{IntReg}(v_c) = & \mathrm{IntReg}(v_d) = \{v_c, v_d\} \\ & \mathrm{IntReg}(v_e) = \{v_e\} \\ & \mathrm{IntReg}(v_f) = \{v_f\} \\ \mathrm{IntReg}(v_g) = & \mathrm{IntReg}(v_h) = \{v_g, v_h\} \\ & \mathrm{IntReg}(v_i) = \{v_i\} \\ & \mathrm{IntReg}(v_j) = \{v_j\} \\ & \mathrm{IntReg}(v_k) = \{v_k\} \end{array}$$

Figure 5.4: The Interaction Regions for the Relational Structures in Example 5.3.3

We can now define the resultant relational structure that is produced by merging the variables in each interaction region. By construction, the τ-equivalent relation induces an equivalence on the columns of the relations of any relational structure. Hence the following are well defined.

Definition 5.3.8. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ be a relational structure. For any R_i in $R_1, \ldots, R_m$ choose* $\mathrm{mergeCol}(R_i)$ *to be any tuple of indices of representatives of the distinct τ-equivalent classes of the columns of R_i.*

For any $t \in R_i$ we now define $\mathrm{mergeCol}(t)$ *to be the tuple of τ-equivalent classes of the variables in t, which occur at the indices of the tuple* $\mathrm{mergeCol}(R_i)$*. Then* $\mathrm{Mrg}(R_i) = \{\mathrm{mergeCol}(t) \mid t \in R_i\}$.

We can then define the ***merged structure*** *of $\mathbb{S}$ to be*

$$\mathrm{Mrg}(\mathbb{S}) = \langle \{\mathrm{IntReg}(v) \mid v \in U\}, \mathrm{Mrg}(R_1), \ldots, \mathrm{Mrg}(R_m) \rangle \ .$$

For a class, $\mathcal{H}$, of relational structures, we denote by $\mathrm{Mrg}(\mathcal{H})$ *the set of merged structures of the structures in $\mathcal{H}$.*

Any CSP that permits a relational structure, $\mathbb{S}$, may be merged so that is permits the merged structure of $\mathbb{S}$ by merging the variables in each interaction region. The merged structures for $\mathbb{S}_A$ and $\mathbb{S}_B$ are shown in Figure 5.5.

$$
\begin{aligned}
\mathrm{Mrg}(\mathbb{S}_A) \quad = \quad & \langle \{\{v_a, v_b\}, \{v_c, v_d\}, \{v_e, v_f\}, \{v_g, v_h\}, \{v_i, v_j\}\}, \\
& R_1 = \{\langle \{v_a, v_b\}, \{v_c, v_d\}\rangle, \langle \{v_e, v_f\}, \{v_c, v_d\}\rangle\}, \\
& R_2 = \{\langle \{v_e, v_f\}, \{v_g, v_h\}\rangle, \langle \{v_i, v_j\}, \{v_g, v_h\}\rangle\}\rangle \\
\\
\mathrm{Mrg}(\mathbb{S}_B) \quad = \quad & \langle \{\{v_a\}, \{v_b\}, \{v_c, v_d\}, \{v_e\}, \{v_f\}, \{v_g, v_h\}, \{v_i\}, \{v_j\}, \{v_k\}\}, \\
& R_1 = \{\langle \{v_a\}, \{v_b\}, \{v_c, v_d\}\rangle, \langle \{v_e\}, \{v_f\}, \{v_c, v_d\}\rangle\}, \\
& R_2 = \{\langle \{v_e\}, \{v_f\}, \{v_g, v_h\}\rangle, \langle \{v_i\}, \{v_j\}, \{v_g, v_h\}\rangle\}, \\
& R_3 = \{\langle \{v_j\}, \{v_k\}\rangle\}\rangle
\end{aligned}
$$

Figure 5.5: The Merged Structures of $\mathbb{S}_A$ and $\mathbb{S}_B$ from Example 5.3.3

It is our assertion that merging the variables in each interaction region allows for a polynomial conversion from the Mixed representation to the Positive representation for classes of bounded interaction width. This gives a natural extension of Grohe's result stated as Theorem 3.4.9 that allows classes of unbounded arity and, in particular, includes the class of CSP instances with precisely one constraint.

Theorem 5.3.9. *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be any recursively enumerable class of relational structures with bounded interaction width, and let $\mathcal{C}$ be the class of CSPs which permit a structure in $\mathcal{H}$. The class* $\mathrm{Mixed}(\mathcal{C})$ *is tractable if and only if* $\mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H}))) < \infty$.

By using an auxiliary structure, the *extended merged structure*, that allows us to 'unmerge' an instance of the required type by providing both the number and names of the variables in each region, we are firstly able to prove this theorem in one direction.

Definition 5.3.10. *Let $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ be a relational structure, and let the list of regions* $\mathrm{IntReg}(v_1), \ldots, \mathrm{IntReg}(v_q)$ *be an enumeration of the distinct τ-equivalent classes. For $i = 1, \ldots, q$ we define the binary relation* $\mathrm{IntReg}_i = \{\langle \mathrm{IntReg}(v_i), y\rangle \mid y \in \mathrm{IntReg}(v_i)\}$.

We define the ***extended merged structure*** *of* $\mathbb{S}$, $\mathrm{ExtMrg}(\mathbb{S})$, *to be*

$$\langle U \cup \{\mathrm{IntReg}(v) \mid v \in U\}, \mathrm{Mrg}(R_1), \ldots, \mathrm{Mrg}(R_m), \mathrm{IntReg}_1, \ldots, \mathrm{IntReg}_q\rangle \ .$$

Proposition 5.3.11. *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be any recursively enumerable class of relational structures with bounded interaction width, and let $\mathcal{C}$ be the class of CSPs which permit a structure in $\mathcal{H}$. The class* $\mathrm{Mixed}(\mathcal{C})$ *is intractable if* $\mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H}))) = \infty$.

Proof. By definition, $\mathrm{Positive}(\mathcal{C})$ is a subset of $\mathrm{Mixed}(\mathcal{C})$, so it is enough to show that $\mathrm{Positive}(\mathcal{C})$ is intractable if $\mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H}))) = \infty$.

For all $\mathbb{S} \in \mathcal{H}$, $\mathrm{Mrg}(\mathbb{S})$ is a substructure of $\mathrm{ExtMrg}(\mathbb{S})$ by definition. So, $\mathrm{tw}(\mathrm{Core}(\mathrm{ExtMrg}(\mathcal{H}))) \geq \mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H})))$.

By definition, the arity of $\text{Mrg}(\mathcal{H})$ is equal to $\text{intw}(\mathcal{H})$. The construction of $\text{ExtMrg}(\mathcal{H})$ only adds binary constraints to $\text{Mrg}(\mathcal{H})$. So, $\text{ExtMrg}(\mathcal{H})$ has bounded arity. By Grohe's result, Theorem 3.4.9, Positive($\mathcal{C}$) is intractable.

Given any instance P in Positive($\mathcal{C}$), for each assignment to an interaction region variable in the extended merged structure, we choose a representative assignment for each of the variables in that interaction region, allowed by P. We use these extensions to generate constraint tuples in an 'unmerged' version of P.

This reduction 'unmerges' P into an instance of Positive($\mathcal{C}$) in polynomial time. Positive($\mathcal{C}$) is thereby shown to be an intractable subset of Mixed($\mathcal{C}$). □

In order to prove the case where $\text{tw}(\text{Core}(\text{Mrg}(\mathcal{H}))) < \infty$ we will provide, and perform a formal complexity analysis of, a method by which a given instance in Mixed($\mathcal{C}$) may be solved in polynomial time with respect to its input size. This is an involved proof which requires systematic analysis of non-trivial algorithms, and we suggest that the reader familiarise themselves with the computational model presented in Chapter 4 first. We then bring these complexity results together on Page 116 to give a short proof completing that of Theorem 5.3.9.

Overview of Method

Given as input a relational structure $\mathbb{S}$ and a CSP P in the mixed representation that permits $\mathbb{S}$, we give a general method for constructing a derived CSP P' in the positive representation that permits $\text{Mrg}(\mathbb{S})$ such that any solution to P' can be transformed to a solution to P in polynomial time with respect to the size of P. Furthermore, if there is a solution to P, then there will be a solution to P'.

We start by determining the interaction regions of the relational structure $\mathbb{S}$. To generate P' we merge P with respect to the interaction regions of $\mathbb{S}$, and then convert each constraint that is in the negative representation to a constraint in the positive representation. It may not be possible to directly convert each of these constraints in the case where domains inferred from the merged relations do not contain all of the values which may occur in a solution. In these cases, we must synthesise constraints in such a way as to preserve solutions and allow P' to permit the relational structure $\text{Mrg}(\mathbb{S})$. To do this, we run two pre-processing steps before merging: one to convert any negative constraints to positive where there is already a positively represented constraint with the same relation, and the other to construct a partition of the constraints in P such that if all the constraints in the same part have the same relation, then the relational structure $\mathbb{S}$ is permitted. This is not necessarily the actual partition that P has, as many partitions may satisfy

the requirement, but it is the strictest in that it contains the smallest possible number of parts that would permit $\mathbb{S}$. Once P has been merged, we run a post-processing step which constructs a partition of the merged constraints in $\text{Mrg}(P)$. It does this by starting with the partition made for P and separating parts where the way in which the constraints have been merged with respect to $\text{Mrg}(\mathbb{S})$ indicate that constraints could not be in the same part. Again, this gives the strictest partition that would allow $\text{Mrg}(P)$ to permit $\text{Mrg}(\mathbb{S})$.

We now convert any negatively represented constraints in $\text{Mrg}(P)$ into equivalent positively represented constraints where we have enough information about the domains to do so. The constraints for which we cannot do this are those which contain some variable which only occurs in the scope of negatively represented constraints, and which has values in its domain which may occur in a solution, but which do not occur in any of the disallowed assignments. However, because the domains of these variables contain some value which may occur in any solution to the remaining variables, the constraints whose scopes contain these variables cannot be placing any restrictions onto any subsets of their scope which does not contain these variables. We give these variables a special '$*$' domain which only contains a single value and then synthesise positively represented constraints in such as way as not to place any additional restrictions on solutions, but such that the resulting converted instance, P', still permits $\text{Mrg}(\mathbb{S})$. To do this, we refer to the partition we constructed earlier.

Any solution to P' now has the special '$*$' value assigned to the merged variables for which we know any solution to the remaining merged variables must extend, so actual values can be found by generating and testing until we find one that is permitted by the constraints we were unable to directly convert. The values assigned to the merged variables then translate directly to assignments to sets of unmerged variables in P

Analysis

Definition 5.3.12 (p)**.** *Let $P = \langle V, D, C \rangle$ be a CSP instance in* Mixed($\mathcal{H}$)*, and let* $\mathbb{S} = \langle U, R_1 \dots R_m \rangle$ *be a relational structure permitted by P. For each variable $v \in V$, the domain $D(v)$ contains all values that may be assigned to v. The parameters of P and $\mathbb{S}$ are defined as follows:*

r *: the largest arity of any constraint in C. By Definition 3.4.1, it is also the largest arity of any relation in $R_1 \dots R_m$.*

k *: the largest number of values in any domain in D.*

n *: the number of variables in V. By Definition 3.4.1, it is also the number of variables in U.*

c *: the number of constraints in C.*

t *: the number of tuples in the constraint in C with the greatest number of tuples.*

m *: the number of relations in $\mathbb{S}$.*

w *: the interaction width of $\mathbb{S}$.*

Variable names and domain values are of constant size, so are assumed to have a size of 1. All parameters are assumed to have a value of at least 1*.*

It is valid to assume that all parameters have a value of at least one as if any, other than t, have no value, then the CSP cannot have a solution. A valid CSP in the Mixed representation may have no tuples in any of its constraints, however in this case they must be a combination of negatively represented 'anything goes' constraints, or positively represented 'all disallowed' constraints. If they are all negatively represented 'anything goes' constraints, then the solution is trivial, i.e. assign any domain value to each of the variables, and if this is not the case (i.e. there is at least one 'all disallowed' constraint), then there cannot be a solution.

It is also valid to assume that all variables of the input CSP have domains of type '$-$'. If the input CSP did contain a variable whose domain was of type '$+$', then that variable could only be in the scopes of negatively represented constraints, and have at least one unlisted domain value. Any unlisted domain values are equivalent within a domain, and so could be replaced with a single value. If there is more than one variable whose domain is '$+$', then these can all be converted to '$-$' domains independently as there can be no restriction on the simultaneous assignment of any of these unlisted domain values to these variables.

Throughout this section we will provide the implementations (with respect to the computational model in Chapter 4) of the major data structures that are assumed by the presented algorithms. This removes the need to redefine these structures in the analysis of each algorithm that uses them, and ensures that they are consistent between algorithms.

The purpose of the algorithms and complexity analyses presented in this section is to prove our

assertion that merging the variables in each interaction region allows for a polynomial conversion from the Mixed representation to the Positive representation for classes of bounded interaction width. As such, the simplicity of algorithms and data structures has been preferred to optimal efficiency.

We start by considering the four algorithms used to determine the interaction regions of a relational structure: 'Generate $\mathbb{S}$-similar' (Algorithm 2), 'Generate $\mathbb{S}$-equivalent' (Algorithm 3), 'Generate Tau Relation' (Algorithm 4), and 'Generate Interaction Regions' (Algorithm 5).

Generate $\mathbb{S}$-similar

Algorithm 2: Generate $\mathbb{S}$-similar

input : A relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$
output: SSimilar - A function from each $\{v, w\}$ in $U \times U$ to the set $\{\{v', w'\} \mid \{v, w\}$ is $\mathbb{S}$-similar-related to $\{v', w'\}\}$

```
1  begin
2      SSimilar ← ∅
3      foreach {v, w} ∈ U × U do
4          SSimilar ({v, w}) ← ∅
5      foreach R_i ∈ {R_1, ..., R_m} do
6          foreach j ← 1 ... Arity(R_i) do
7              foreach k ← 1 ... Arity(R_i) do
8                  foreach t_1 ∈ R_i do
9                      foreach t_2 ∈ R_i do
10                         SSimilar ({t_1 [j], t_1 [k]}) ←
11                                 SSimilar ({t_1 [j], t_1 [k]}) ∪ {t_2 [j], t_2 [k]}
12     return SSimilar
```

Data Structure 5.3.13. : Relational Structure

Let $\mathbb{S} = \langle U, R_1 \ldots R_m \rangle$ be a relational structure. The universe, U, is stored as an array of n variable names, and each relation, R_i, is stored as an array of at most c tuples. Each tuple is stored as an array of at most r variable names. By Definition 4.0.4, the size of the array for U is $O(n)$, for each tuple is $O(r)$, and so for each R_i is $O(c.r)$. The relational structure $\mathbb{S}$ is stored as the array for the universe followed by at most m relation arrays, so the combined size of a relational structure is $O(n + m.c.r)$. ⌟

Data Structure 5.3.14. : $\mathbb{S}$-similar and $\mathbb{S}$-equivalent

Let $\mathbb{S} = \langle U, R_1 \dots R_m \rangle$ be a relational structure. Both $\mathbb{S}$-similar and $\mathbb{S}$-equivalent are stored as an associative array that maps from pairs of variables in the universe, U, to the set of pairs of variables that they are either $\mathbb{S}$-similar-related to or $\mathbb{S}$-equivalent-related to. So, there are n^2 pairs of variables over U in the key array, and the value array itself contains arrays each of which may contain at most n^2 pairs of variables. By Definition 4.0.4, each of the arrays in the value array is of size $O\left(n^2\right)$, so by Definition 4.0.15 the sizes of the key and value arrays are $O\left(n^2\right)$ and $O\left(n^4\right)$ respectively. The combined size of the associative array is therefore $O\left(n^4\right)$. ⌟

Algorithm Analysis 5.3.15. : Generate $\mathbb{S}$-similar (Algorithm 2)

By Proposition 4.0.16, the construction of SSimilar (Data Structure 5.3.14) on line 2 is $O\left(1\right)$ time. The loop on line 3 will execute at most n^2 times. For each execution of the associative array insertion statement on line 4, the insertion key will be unique and the value is an empty array of size at most n^2. The key is a pair so, by Proposition 4.0.19, each insertion into SSimilar is $O\left(1\right)$ time.

The loop on line 5 will execute at most m times, the loop on line 6 at most r times, and the loop on line 7 at most r times. For a relational structure permitted by P, the total number of tuples in each relation is bounded by the number of constraints, so the loop on line 8 will execute at most c times, and the loop on line 9 at most c times.

Construction of each pair of variables for both the keys and values on lines 10 and 11 requires two array lookups by index, which by Proposition 4.0.6 are $O\left(1\right)$ time, and by Proposition 4.0.2 reading each variable is $O\left(1\right)$ time.

The key is a constant sized pair so, by Proposition 4.0.17, lookup of the corresponding mapped set in SSimilar on lines 10 and 11 is $O\left(n^2\right)$ time. By Proposition 4.0.13, as the result replaces one of the input sets and the other input set always has only a single binary element, the union operation on lines 10 and 11 is $O\left(n^2\right)$ time.

We can now summarise the time complexity analysis for Algorithm 2 (Generate $\mathbb{S}$-similar):

$$\begin{aligned} & O\left(n^2 + m.r^2.c^2\left(n^2 + n^2\right)\right) \\ = \; & O\left(n^2 + m.r^2.c^2.n^2\right) \\ = \; & O\left(m.r^2.c^2.n^2\right) \end{aligned}$$

⌟

Lemma 5.3.16. *Given a relational structure* $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$ *as input, Algorithm 2 ('Generate* $\mathbb{S}$*-similar') outputs the* $\mathbb{S}$*-similar relation.*

Proof. The construction of $\mathbb{S}$-similar requires that the variables in every pair of positions for every pair of tuples are considered for every relation. The loops on lines 5 to 9 ensure that this condition is met. The union operation on line 10 then ensures that every unique pair considered is then recorded. □

Generate $\mathbb{S}$-equivalent

Algorithm 3: Generate $\mathbb{S}$-equivalent

input : A relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$
input : SSimilar - A function from each $\{v, w\}$ in $U \times U$ to the set $\{\{v', w'\} \mid \{v, w\}$ is $\mathbb{S}$-similar-related to $\{v', w'\}\}$
output: SEquivalent - A function from each $\{v, w\}$ in $U \times U$ to the set $\{\{v', w'\} \mid \{v, w\}$ is $\mathbb{S}$-equivalent-related to $\{v', w'\}\}$

```
1  begin
2      SEquivalent ← ∅
3      Pairs ← Domain (SSimilar) as List
4      N ← |Pairs|
                /* Transitive closure algorithm requires matrix data structures */
5      M^0 ← a new N × N Boolean matrix
6      foreach i ← 1 ... N do        /* Convert SSimilar to matrix data structure */
7          foreach j ← 1 ... N do
8              if i = j or Pairs[j] ∈ SSimilar (Pairs[i]) then
9                  M^0[i][j] ← true
10             else
11                 M^0[i][j] ← false
12     foreach k ← 1 ... N do                          /* Perform transitive closure */
13         M^k ← a new N × N Boolean matrix
14         foreach i ← 1 ... N do
15             foreach j ← 1 ... N do
16                 M^k[i][j] ← M^{k-1}[i][j] ∨ (M^{k-1}[i][k] ∧ M^{k-1}[k][j])
17     foreach i ← 1 ... N do                  /* Convert from matrix data structure */
18         SEquivalent (Pair[i]) ← ∅
19         foreach j ← 1 ... N do
20             if M^n[i][j] = true then
21                 SEquivalent (Pair[i]) ← SEquivalent (Pair[i]) ∪ {Pair[j]}
22     return SEquivalent
```

Algorithm Analysis 5.3.17. : Generate $\mathbb{S}$-equivalent (Algorithm 3)

By Proposition 4.0.16, the construction of SEquivalent (Data Structure 5.3.14) on line 2 is $O(1)$ time. Neither Pairs nor SSimilar are modified during the execution of the algorithm. As such, the array of keys in SSimilar may be used as the data structure for Pairs and the operation on line 3 requires no more than a unit time assignment to record the memory offset location to a register, so $O(1)$ time. By Definition 4.0.4, the number of elements in Pairs is already recorded as part of the array of keys in SSimilar, and is at most n^2. Recording this value for N on line 4 is $O(1)$ time.

An $N \times N$ Boolean array can be represented as a length N array of length N Boolean arrays. By Proposition 4.0.5, construction of an empty array is $O(1)$ time, so construction of the outer array is $O(1)$ time. To this array, at most n^2 inner arrays are added. By Proposition 4.0.5, construction of each inner array is also $O(1)$ time, and by Proposition 4.0.8 inserting each empty inner array into the outer array is $O(1)$. The algorithm does not require the inner arrays to be initialised at this point as all values will subsequently be written to before being read. As such, the construction of the $N \times N$ Boolean array on line 5 requires $O\left(n^2\right)$ time.

Lines 6 to 16 describe a known adaptation of the Floyd-Warshall algorithm for computing the transitive closure of a directed graph [CLRS09]. When given an $N \times N$ Boolean matrix, this algorithm is known to execute exactly N^3 operations, so here executes exactly n^6 operations as the matrix is $n^2 \times n^2$, so is $\Theta\left(n^6\right)$ time.

The loop on line 17 will execute at most n^2 times. For each execution of the associative array insertion statement on line 18, the insertion key will be unique and the value is an empty array of size at most n^2. The key is a pair so, by Proposition 4.0.19, each insertion into SEquivalent is $O(1)$ time.

The loop on line 19 will execute at most n^2 times. On line 20, finding the base address of the inner array at the i^{th} position of the outer array, followed by the base address of the Boolean value at the j^{th} position of the inner array. By Proposition 4.0.6, both these operations are $O(1)$ time. By Proposition 4.0.2, reading this value then requires $O(1)$ time, and by Proposition 4.0.3 comparing it to *true* also requires $O(1)$ time. Evaluation of the condition on line 20 is therefore $O(1)$ time.

On line 21, the lookups by index of Pairs required to determine the base addresses of the keys and values are $O(1)$ by Proposition 4.0.6. The key is a pair of variables so, by Proposition 4.0.17, lookup of the corresponding mapped set in SEquivalent is $O\left(n^2\right)$. By Proposition 4.0.13, as the

result replaces one of the input sets and the other input set always has only a single binary element, the union operation is $O\left(n^2\right)$.

We can now summarise the time complexity analysis for Algorithm 3 (Generate $\mathbb{S}$-equivalent):

$$\begin{aligned} & O\left(n^2\right) + \Theta\left(n^6\right) + O\left(n^2\left(n^2\left(n^2+n^2\right)\right)\right) \\ = \; & O\left(n^2\right) + \Theta\left(n^6\right) + O\left(n^6\right) \\ = \; & O\left(n^6\right) \end{aligned}$$

⌟

Lemma 5.3.18. *Given a relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$, and the $\mathbb{S}$-similar relation as input, Algorithm 3 ('Generate $\mathbb{S}$-equivalent') outputs the $\mathbb{S}$-equivalent relation.*

Proof. $\mathbb{S}$-equivalent is the transitive closure of the $\mathbb{S}$-similar relation. Lines 6 to 16 describe a known adaptation of the Floyd-Warshall algorithm for computing the transitive closure of a directed graph [CLRS09]. Lines 17 to 21 then perform a translation from the internal matrix structure of this algorithm to our more explicit $\mathbb{S}$-equivalent data structure as defined in Data Structure 5.3.14. □

Generate Tau Relation

Algorithm 4: Generate Tau Relation

input : A relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$
output: Tau - A function from each v in U to the set of tuples $\{\langle i, t \rangle \mid t \in R_i, v \in \text{set}(t)\}$

```
1 begin
2     Tau ← ∅
3     foreach v ∈ U do
4         Tau (v) ← ∅
5     foreach R_i ∈ {R_1, ..., R_m} do
6         foreach t ∈ R_i do
7             foreach v ∈ t do
8                 Tau (v) ← Tau (v) ∪ {⟨i, t⟩}
9     return Tau
```

Data Structure 5.3.19. : τ relation

Let $P = \langle V, D, C \rangle$ be a CSP instance, and let $\mathbb{S} = \langle U, R_1 \ldots R_m \rangle$ be a relational structure permitted by P. The τ relation for $\mathbb{S}$ is stored as an associative array that maps from the n variables in U to a set of pairs. Each pair consists of a tuple from one of the relations in $R_1 \ldots R_m$, and an integer, $i \in 1 \ldots m$, identifying which relation the tuple is from. Each tuple has at most r

members, so the size of each pair is $O(r)$. By Definition 3.4.1, the total number of tuples which may belong to a relation of a relational structure permitted by P is bounded by the number of constraints, so each set may contain at most $c.m$ pairs. By Definition 4.0.4, the size of each set is $O(r.c.m)$. By Definition 4.0.15, the array of keys is $O(n)$, and the array of values is $O(r.c.m.n)$, so the combined size of the associative array is $O(r.c.m.n)$. ⌟

Algorithm Analysis 5.3.20. : Generate Tau Relation (Algorithm 4)
By Proposition 4.0.16, the construction of Tau (Data Structure 5.3.19) on line 2 is $O(1)$ time. The loop on line 3 will execute at most n times. For each execution of the associative array insertion statement on line 4, the insertion key will be unique and the value is an empty array of size at most $O(r.c.m)$. The key is a single variable so, by Proposition 4.0.19, each insertion into Tau is $O(1)$ time.

The loop on line 5 will execute at most m times, the loop on line 6 at most c times, and the loop on line 7 at most r times.

The key is a single variable so, by Proposition 4.0.17, lookup of the corresponding mapped set in Tau on line 8 is $O(n)$ time. By Proposition 4.0.13, as the result replaces one of the input sets and the other input set always has only a single pair of size $O(r)$, the union operation on line 8 is $O\left(c.m.r^2\right)$.

We can now summarise the time complexity analysis for Algorithm 4 (Generate Tau Relation):

$$\begin{aligned} & O(n) + O\left(m.c.r\left(c.m.r^2\right)\right) \\ = \; & O(n) + O\left(m^2.c^2.r^3\right) \\ = \; & O\left(n + m^2.c^2.r^3\right) \end{aligned}$$

⌟

Lemma 5.3.21. *Given a relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$ as input, the algorithm 'Generate Tau Relation' (Algorithm 4) outputs the τ relation.*

Proof. For each variable in U, the τ relation lists as pairs the tuples of the relational structure containing that variable, along with an identifier describing which relation the tuple is a member of. This requires us to consider each tuple in each relation for each variable, which is performed by the loops on lines 4 to 6. The union operation on line 8 ensures that each tuple is recorded. □

Generate Interaction Regions

Algorithm 5: Generate Interaction Regions

input : A relational structure $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$
output: Regions - A function from each v in U to the set of variables $X \subseteq U$ such that $\forall x \in X, x$ is in the same region as v

```
1  begin
2      SSimilar ← Generate 𝕊-similar (𝕊)
3      SEquivalent ← Generate 𝕊-equivalent (SSimilar)
4      Tau ← Generate Tau Relation (𝕊)
5      Regions ← ∅
6      Pairs ← Domain (SEquivalent)
7      foreach v ∈ U do
8          Regions (v) ← {v}
9      foreach {v, w} ∈ Pairs do                    /* Check for τ-equivalance */
10         equiv ← true
11         foreach {v′, w′} ∈ SEquivalent ({v, w}) do
12             if Tau (v′) ≠ Tau (w′) then
13                 equiv ← false
14         if equiv = true then
15             Regions (v) ← Regions (v) ∪ {w}
16             Regions (w) ← Regions (w) ∪ {v}
17     return Regions
```

Data Structure 5.3.22. : Interaction Regions

The interaction regions are stored as an associative array that maps from variables in the universe to the set of variables that are in the same interaction region. The key array contains the the n variables of U, and the value array contains n arrays each of which contains at most r variables. By Definition 4.0.15 the sizes of the key and value arrays are $O(n)$ and $O(n.r)$ respectively. The combined size of the associative array is therefore $O(n.r)$. ⌟

Algorithm Analysis 5.3.23. : Generate Interaction Regions (Algorithm 5)

On line 2 we call Algorithm 2 (Generate $\mathbb{S}$-similar) whose time complexity is shown to be $O\left(m.r^2.c^2.n^2\right)$ in Algorithm Analysis 5.3.15. On line 3 we call Algorithm 3 (Generate $\mathbb{S}$-equivalent) whose time complexity is shown to be $O\left(n^6\right)$ in Algorithm Analysis 5.3.17. On line 4 we call Algorithm 4 (Generate Tau Relation) whose time complexity is shown to be $O\left(n + m^2.c^2.r^3\right)$ in Algorithm Analysis 5.3.20.

By Proposition 4.0.16, the construction of Regions (Data Structure 5.3.22) on line 5 is $O(1)$ time. Neither Pairs nor SEquivalent are modified during the execution of the algorithm. As such, the array of keys in SEquivalent may be used as the data structure for Pairs and the operation on line 6 requires no more than a unit time assignment to record the memory offset location to a register. By Definition 4.0.4, the number of elements in Pairs is already recorded as part of the array of keys in SEquivalent, and is at most n^2.

The loop on line 7 will execute at most n times. Each execution of the associative array insertion statement on line 3, can be considered as an operation to insert an empty set as the value, followed by adding a single element to the value array. For the first operation, the insertion key will be unique and the value is an empty array of size at most $O(r)$. The key is a single variable so, by Proposition 4.0.19, each insertion into Regions is $O(1)$ time. For the second operation, the base address of the value array has already been determined in the first operation so, by Proposition 4.0.8, inserting the single variable to the value array is $O(1)$ time.

The loop on line 9 will execute at most n^2 times. By Proposition 4.0.2 storing the Boolean value on line 10 is $O(1)$ time.

The loop on line 11 will execute at most n^2 times. The Tau Relation data structure (Data Structure 5.3.19) is an associative array mapping from variables to sets of at most $c.m$ data structures of size $O(r)$. The condition on line 12 is evaluating whether two of the set arrays stored as values in the Tau associative array are equivalent. By Proposition 4.0.17, the lookup of each value array by a single variable key is $O(n)$ time, and by Proposition 4.0.12 determining whether these two sets are equivalent requires $O\left((c.m)^2 r\right)$ time. By Proposition 4.0.2 storing the Boolean value on line 13 is $O(1)$ time.

The condition on line 14 is comparing two Boolean values, so by Proposition 4.0.3 is $O(1)$ time. On lines 15 and 16, the keys are single variables so, by Proposition 4.0.17, lookup of the corresponding mapped sets requires $O(n)$ time. By Proposition 4.0.13, as the results replace one of the input sets, and the other input sets contain a single unary element, the union operations on line 15 and 16 are $O\left(1.r.1^2\right)$, so $O(r)$ time.

We can now summarise the time complexity analysis for Algorithm 5 (Generate Interaction Regions):

$$\begin{aligned}
& O\left(m.r^2.c^2.n^2 + n^6 + n + m^2.c^2.r^3 + \left(n + n^2\left(n^2\left(2.n + 2.\left((c.m)^2.r\right)\right) + 2.r\right)\right)\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + \left(n + n^2\left(n^2\left(n + c^2.m^2.r\right) + r\right)\right)\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + \left(n + n^2\left(n^3 + n^2.c^2.m^2.r + r\right)\right)\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n + n^5 + n^4.c^2.m^2.r + n^2.r\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r\right)
\end{aligned}$$

⌟

Lemma 5.3.24. *Given a relational structure $\mathbb{S} = \langle U, R_1, \ldots R_m \rangle$ represented using Data Structure 5.3.13, the $\mathbb{S}$-equivalent relation represented using Data Structure 5.3.14, and the τ-equivalent relation represented using Data Structure 5.3.19 as input, the algorithm 'Generate Interaction Regions' (Algorithm 5) outputs the interaction regions of $\mathbb{S}$.*

Proof. The interaction region for a given variable v in U should contain the set of variables whose pairing with v has the property that both members of all $\mathbb{S}$-equivalent-related pairs have the same τ relation. Trivially, the interaction region for v should contain v, and this case is covered by the initialisation process on lines 4 and 5. The domain of SEquivalent contains all binary combinations of the variables in U, so the loops on line 6 and 8 ensure that the τ equality check on line 9 is performed for all $\mathbb{S}$-equivalent pairs of all possible pairs in U. The set unions on lines 12 and 13 ensure that the resultant relationship between the two variables in a pair is recorded against both variables. □

For a relational structure $\mathbb{S} = \langle U, R_1 \ldots R_m \rangle$ to be permitted by a CSP it must be true that whenever the scopes of two constraints are in the same R_i the constraints have the same relation. For the theoretical model of a CSP, where all relations are listed as allowed tuples over the product of a universal domain, it is clear when two relations are the same. However, in the mixed representation the constraint relations may be expressed as either allowed or disallowed tuples over the product of the individual domains of the variables in the scope. It is no longer clear when two constraint relations are the same in the mixed representation because the information about the restriction imposed by a constraint is now distributed between its relation and the domains of the variables in its scope. We call two constraint relations in the mixed representation *equivalent* if they are the same when represented in the positive representation. Our conversion method requires that we are able to identify which constraints in the input CSP have equivalent relations without converting their representations, so here we present the constraint and CSP data structures, and the algorithms by which this can be achieved for the three types of comparison:

positive to positive, positive to negative, and negative to negative. Note that these algorithms will only work when comparing constraints that are over variables whose domains list all values which may be assigned to that variable (i.e. have type '−'). This is true for all constraints in the input CSP, but may not be the case for constraints in intermediate CSP structures in our method.

A constraint relation in the positive representation lists all of the allowed assignments over the scope variables, and the domains of these variables must at least contain any values in the assignments. Having any extra values in the domains does not change the set of allowed assignments, so the domains do not need to be considered. As such, comparing the relations of two positive constraints is straightforward: They must have the same arity, and contain the same assignments (tuples). This method is given in Algorithm 6 ('Compare Positive Constraint Relations').

Compare Positive Constraint Relations

Algorithm 6: Compare Positive Constraint Relations

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
input : c_a - A constraint in C in the positive representation
input : c_b - A constraint in C in the positive representation
output: Boolean - TRUE if c_a and c_b have equivalent relations, else FALSE

```
1 begin
2     if |σ(c_a)| ≠ |σ(c_b)| then
3         return FALSE
4     if |ρ(c_a)| ≠ |ρ(c_b)| then
5         return FALSE
6     foreach t ∈ ρ(c_a) do
7         if t ∉ ρ(c_b) then
8             return FALSE
9     return TRUE
```

Data Structure 5.3.25. : Constraint

Let $c = \langle \sigma, \rho \rangle$ be a constraint. The scope, σ, is stored as an array of r variable names so, by Definition 4.0.4, is of size $O(r)$. Each tuple in the relation, ρ, is stored as an r-ary array of values such that the value in each position corresponds to the variable in the same position in the scope array. By Definition 4.0.4, the size of each tuple array is $O(r)$. The relation is stored as an array of t tuple arrays so, by Definition 4.0.4, the size of the relation array is $O(t.r)$. A single register is used to store the representation type of the constraint, so the size of the combined constraint structure is $O(r + t.r + 1)$, so $O(t.r)$. ⌟

Data Structure 5.3.26. : CSP

Let $\langle V, D, C \rangle$ be a CSP instance. The variables, V, are stored as an array of n variable names so, by Definition 4.0.4, is of size $O(n)$. The domain, D, is stored as an associative array that maps from the n variable names to an array of at most k domain values, plus a single register to denote the type of the domain. By Definition 4.0.4, the size of each value array is $O(k)$ so, by Definition 4.0.15, the size of the associative array is $O(n + k.n)$, so $O(k.n)$. The constraints, C, are stored as an array of c constraint structures as described in Data Structure 5.3.25 so, by Definition 4.0.4, the size of the constraint array is $O(t.r.c)$. The size of the combined CSP structure is therefore $O(n + k.n + t.r.c)$, so $O(k.n + t.r.c)$. ⌟

Algorithm Analysis 5.3.27. : Compare Positive Constraint Relations (Algorithm 6)
By Data Structure 5.3.25 for a constraint, each scope on line 2 is stored as an array containing at most r variable names so by Definition 4.0.4 for an array, determining the size of each scope takes O (1) time. By Definition 4.0.1 for a RAM, comparing two integers takes $O(1)$ time.

Again by Data Structure 5.3.25 for a constraint, each relation on line 4 is stored as an array containing at most t tuples of at most r values so by Definition 4.0.4 for an array, determining the size of each relation takes O (1) time. Comparing two integers takes $O(1)$ time.

The loop on line 6 will execute at most t times. By Proposition 4.0.11 checking membership of a tuple of size at most $O(r)$ in a relation array containing at most t tuples requires $O(r.t)$ time.

We can now summarise the time complexity analysis for Algorithm 6 (Compare Positive Constraint Relations):

$$\begin{aligned} & O(t(r.t)) \\ = \quad & O\left(t^2.r\right) \end{aligned}$$

⌟

In a negatively represented constraint, the Cartesian product of the domains defines the set of possible assignments from which the listed disallowed assignments are removed. It is still straightforward to compare two constraint relations when one is in the positive representation and the other is in the negative representation because the positively represented relation provides the complete list of allowed assignments. Whether the Cartesian product of the domains in the negatively represented constraint could support all of the allowed assignments in the positively represented relation can be determined by checking whether the values in the unary projections of the positively represented relation are present in the appropriate domains of the negatively represented constraint. The number of assignments that the negatively represented relation would

express if it were in the positive representation can be determined by subtracting the number of disallowed assignments from the number that would be allowed by the Cartesian product of the domains. If the number of allowed assignments minus the number of disallowed assignments equals the number of allowed assignments in the positively represented relation, then we simply need to check that none of the positively allowed assignments is in the set of disallowed assignments. This method is given in Algorithm 7 ('Compare Positive and Negative Constraint Relations').

Compare Positive and Negative Constraint Relations

Algorithm 7: Compare Positive and Negative Constraint Relations

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
input : c_a - A constraint in C in the positive representation
input : c_b - A constraint in C in the negative representation
output: Boolean - TRUE if c_a and c_b have equivalent relations, else FALSE

```
1  begin
2      if |σ(c_a)| ≠ |σ(c_b)| then
3          return FALSE
4      foreach v ∈ σ(c_a) do          /* Check all values occurring in the Positive */
5          i ← position of v in σ(c_a)   /* representation are in the domains of the */
6          v' ← σ(c_b)[i]                /* variables in the Negative representation */
7          foreach t ∈ ρ(c_a) do
8              if Π_v t ∉ D(v') then
9                  return FALSE
10     count ← 1
11     foreach v ∈ σ(c_b) do
12         count ← count ×|D(v)|
13     if count −|ρ(c_b)| ≠ |ρ(c_a)| then
14         return FALSE
15     foreach t ∈ ρ(c_a) do
16         if t ∈ ρ(c_b) then
17             return FALSE
18     return TRUE
```

Algorithm Analysis 5.3.28. : Compare Positive and Negative Constraint Relations (Algorithm 7)

By Data Structure 5.3.25 for a constraint, each scope on line 2 is stored as an array containing at most r variable names so by Definition 4.0.4 for an array, determining the size of each scope

takes O (1) time. By Definition 4.0.1 for a RAM, comparing two integers takes $O(1)$ time.

The loop on line 4 will execute at most r times. By Proposition 4.0.11 finding the index of an element in a scope array on line 5 requires $O(r)$ time. By Proposition 4.0.6 addressing a variable in a scope array by index on line 6 is $O(1)$ time. The loop on line 7 will execute at most t times. The position of v is already known (i), so by Proposition 4.0.10 projecting the tuple on line 8 requires $O(1)$ time. By Data Structure 5.3.26, a domain is an associative array mapping from variables to arrays of values, so by Proposition 4.0.17 finding $D(v')$ requires $O(n)$ time, and by Proposition 4.0.11 checking membership of a value then takes $O(k)$ time.

Assigning a value to a counter on line 10 is $O(1)$. The loop on line 11 will execute at most r times. Looking up the domain of v on line 12 requires $O(n)$ time, and determining the size of the value array is $O(1)$ time. By Data Structure 5.3.26, constraint relations are stored as arrays, so determining the size of each relation on line 13 is $O(1)$ time, and the integer arithmetic required to evaluate the condition requires $O(1)$ time. The loop on line 15 will execute at most t times. A tuple contains at most r values, and a relation contains at most t tuples, so by Proposition 4.0.11 determining membership on line 16 requires $O(r.t)$ time.

We can now summarise the time complexity analysis for Algorithm 7 (Compare Positive and Negative Constraint Relations):

$$\begin{aligned} & O\left(r\left(r + t\left(n + k\right)\right) + r\left(n\right) + t\left(r.t\right)\right) \\ = \; & O\left(r\left(r + t.n + t.k\right) + r.n + t^2.r\right) \\ = \; & O\left(r^2 + t.n.r + t.k.r + t^2.r\right) \end{aligned}$$

⌟

Comparing the relations of two negatively represented constraints is more complicated because the Cartesian product of the domains for each constraint may be different, and we do not have a positively represented constraint that provides the set of allowed assignments. However, it is still possible to determine whether two negatively represented constraints have equivalent relations by performing two symmetrical checks. Firstly, if a disallowed assignment in one relation is a member of the Cartesian product of the domains of the other, then the other relation must also have this disallowed assignment. This check is given in Algorithm 8 ('Check Disallowed Assignments'). Secondly, if for some position in the scope the domain of the variable in one constraint contains a value which is not in the domain of the other corresponding variable, then that relation must contain all extensions to the variable being assigned that value as disallowed assignments because they are implicitly disallowed by not being in the Cartesian product of allowed assignments in the other constraint. This check is given in Algorithm 9 ('Check Extra Domain Values'), and the full comparison method is given in Algorithm 10 ('Compare Negative Constraint Relations').

Check Disallowed Assignments

Algorithm 8: Check Disallowed Assignments

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
input : c_a - A constraint in C in the negative representation
input : c_b - A constraint in C in the negative representation
output: Boolean - TRUE if c_a and c_b meet condition one, else FALSE

```
1  begin
2      foreach t ∈ ρ(c_a) do
3          inProduct ← TRUE
4          foreach v ∈ σ(c_a) do
5              i ← position of v in σ(c_a)
6              v' ← σ(c_b)[i]
7              if Π_v t ∉ D(v') then
8                  inProduct ← FALSE
9          if inProduct = TRUE then
10             if t ∉ ρ(c_b) then
11                 return FALSE
12     return TRUE
```

Algorithm Analysis 5.3.29. : Check Disallowed Assignments (Algorithm 8)

The loop on line 2 will execute at most t times. Assigning a Boolean value to a variable is $O(1)$ time. The loop on line 4 will execute at most r times. By Proposition 4.0.11 finding the index of an element in a scope array on line 5 requires $O(r)$ time. By Proposition 4.0.6 addressing a variable in a scope array by index on line 6 is $O(1)$ time. The position of v is already known (i), so by Proposition 4.0.10 projecting the tuple on line 7 requires $O(1)$ time. By Data Structure 5.3.26, a domain is an associative array mapping from variables to arrays of values, so by Proposition 4.0.17 finding $D(v')$ requires $O(n)$ time, and by Proposition 4.0.11 checking membership of a value then takes $O(k)$ time. Assigning the Boolean value on line 8 is $O(1)$ time. Evaluating the condition of a Boolean variable on line 9 requires $O(1)$ time. A tuple contains at most r values, and a relation contains at most t tuples, so by Proposition 4.0.11 determining membership on line 10 requires $O(r.t)$ time.

We can now summarise the time complexity analysis for Algorithm 8 (Check Disallowed Assignments):

$$\begin{aligned} & O\left(t\left(r\left(r+n+k\right)+r.t\right)\right) \\ = \quad & O\left(t\left(r^2+r.n+r.k+r.t\right)\right) \\ = \quad & O\left(t.r^2+t.r.n+t.r.k+r.t^2\right) \end{aligned}$$

⌟

Check Extra Domain Values

Algorithm 9: Check Extra Domain Values

input : A CSP $\langle V, D, C\rangle$ in the mixed representation
input : c_a - A constraint in C in the negative representation
input : c_b - A constraint in C in the negative representation
output: Boolean - TRUE if c_a and c_b meet condition two, else FALSE

```
1  begin
2      foreach v ∈ σ(c_a) do
3          foreach d ∈ D(v) do
4              i ← position of v in σ(c_a)
5              v' ← σ(c_b)[i]
6              if d ∉ D(v') then        /* Check v assigned d can never be allowed */
7                  expected ← 1
8                  foreach v'' ∈ σ(c_a) do
9                      if v ≠ v'' then
10                         expected ← expected ×|D(v'')|
11                 actual ← 0
12                 foreach t ∈ ρ(c_a) do
13                     if Π_v t = d then
14                         actual ← actual +1
15                 if actual ≠ expected then
16                     return FALSE
17     return TRUE
```

Algorithm Analysis 5.3.30. : Check Extra Domain Values (Algorithm 9)

The loop on line 2 will execute at most r times. By Data Structure 5.3.26, a domain is an associative array mapping from variables to arrays of values, so by Proposition 4.0.17 finding $D(v)$ requires $O(n)$ time. For each iteration of the loop on line 2, the loop on line 3 will execute at most k times. By Proposition 4.0.11 finding the index of an element in a scope array on line 4

requires $O(r)$ time. By Proposition 4.0.6 addressing a variable in a scope array by index on line 5 is $O(1)$ time. Again by Proposition 4.0.17, finding $D(v')$ on line 6 requires $O(n)$ time, and by Proposition 4.0.11 checking membership is $O(k)$ time. By Definition 4.0.1 for a RAM, Setting the integer variable on line 7 is $O(1)$ time.

The loop on line 8 will execute at most r times. By Proposition 4.0.3 comparing two variable names is $O(1)$ time. By Proposition 4.0.17, finding $D(v'')$ on line 10 requires $O(n)$ time. By Definition 4.0.4, determining the size of the value array is $O(1)$ time, and performing the integer arithmetic to update the value of 'expected' is $O(1)$ time.

Setting the integer variable on line 11 is $O(1)$ time. The loop on line 12 will execute at most t times. The position of v is already known (i), so by Proposition 4.0.10 projecting the tuple on line 13 requires $O(1)$ time, and comparing the result to the domain value requires $O(1)$ time. The condition on line 15 can be evaluated using integer arithmetic, so is $O(1)$ time.

We can now summarise the time complexity analysis for Algorithm 9 (Check Extra Domain Values):

$$\begin{aligned} & O\left(r\left(n + k\left(r + n + k + r.n + t\right)\right)\right) \\ = \; & O\left(r\left(n + k.r + k.n + k^2 + k.r.n + k.t\right)\right) \\ = \; & O\left(n.r + k.r^2 + k.n.r + k^2.r + k.r^2.n + k.t.r\right) \\ = \; & O\left(k.n.r + k^2.r + k.r^2.n + k.t.r\right) \end{aligned}$$

⌟

Compare Negative Constraint Relations

Algorithm 10: Compare Negative Constraint Relations

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
input : c_a - A constraint in C in the negative representation
input : c_b - A constraint in C in the negative representation
output: Boolean - TRUE if c_a and c_b have equivalent relations, else FALSE

```
1 begin
2     if |σ(c_a)| ≠ |σ(c_b)| then
3         return FALSE
4     result ← Check Disallowed Assignments ( c_a, c_b )
5     result ← result ∧ Check Disallowed Assignments ( c_b, c_a )
6     result ← result ∧ Check Extra Domain Values ( c_a, c_b )
7     result ← result ∧ Check Extra Domain Values ( c_b, c_a )
8     return result
```

Algorithm Analysis 5.3.31. : Compare Negative Constraint Relations (Algorithm 10) By Data Structure 5.3.25 for a constraint, each scope on line 2 is stored as an array containing at most r variable names so by Definition 4.0.4 for an array, determining the size of each scope takes O (1) time. By Definition 4.0.1 for a RAM, comparing two integers takes $O(1)$ time.

On lines 4 and 5 we call Algorithm 8 (Check Disallowed Assignments) whose time complexity is shown to be $O\left(t.r^2 + t.r.n + t.r.k + r.t^2\right)$ in Algorithm Analysis 5.3.29.

On lines 6 and 7 we call Algorithm 9 (Check Extra Domain Values) whose time complexity is shown to be $O\left(k.n.r + k^2.r + k.r^2.n + k.t.r\right)$ in Algorithm Analysis 5.3.30.

By Definition 4.0.1 for a RAM, assignments to a Boolean variable and evaluation of logical operators can be performed in $O(1)$ time.

We can now summarise the time complexity analysis for Algorithm 10 (Compare Negative Constraint Relations):

$$\begin{aligned} & O\left(2.\left(t.r^2 + t.r.n + t.r.k + r.t^2\right) + 2.\left(k.n.r + k^2.r + k.r^2.n + k.t.r\right)\right) \\ = \quad & O\left(t.r^2 + t.r.n + t.r.k + r.t^2 + k.n.r + k^2.r + k.r^2.n + k.t.r\right) \\ = \quad & O\left(t.r^2 + t.r.n + t.r.k + r.t^2 + k.n.r + k^2.r + k.r^2.n\right) \end{aligned}$$

⌟

Now that we are able to identify when two constraints have equivalent relations, we can perform our first pre-processing step on the input CSP, which is to look for any negatively represented constraints whose relations are equivalent to that of some positively represented constraint. If we find such a negatively represented constraint, then we can replace its relation with a copy of the positive equivalent. In the input CSP all domains are of type '$-$', so there is no need to change the domains of the variables in the scope of the constraint whose relation is changed as to have equivalent relations the domains must already contain all values occurring in the set of allowed assignments. We perform this action using Algorithm 11 ('Replace Equivalent Negative Constraints').

Replace Equivalent Negative Constraints

Algorithm 11: Replace Equivalent Negative Constraints

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
output: A CSP $\langle V, D, C \rangle$ in the mixed representation

```
1 begin
2     foreach c ∈ C do
3         if constraint type of c is positive then
4             foreach c′ ∈ C do
5                 if constraint type of c′ is negative then
6                     if Compare Positive and Negative Constraint Relations (c, c′) then
7                         ρ(c′) ← ρ(c)
8                         set constraint type of c′ to positive
9     return ⟨V, D, C⟩
```

Algorithm Analysis 5.3.32. : Replace Equivalent Negative Constraints (Algorithm 11) The loop on line 2 will execute at most c times. By Data Structure 5.3.25 the type of a constraint is stored in a single register, so determining the type of a constraint on line 3 is $O(1)$ time. For each iteration of the loop on line 2, the loop on line 4 will execute at most c times. Again, the type of a constraint can be determined on line 5 in $O(1)$ time.

On line 6 we call Algorithm 7 (Compare Positive and Negative Constraint Relations) whose time complexity is shown to be $O\left(r^2 + t.n.r + t.k.r + t^2.r\right)$ in Algorithm Analysis 5.3.28. As by Data Structure 5.3.25 the relation is an array in which the tuples are arrays ordered with respect to a separate scope array, the relation for c on line 7 may be replaced by copying the relation for c' directly. By Proposition 4.0.2, reading and writing a relation Data Structure is $O(t.r)$. Setting the type of the constraint on line 8 requires writing to a single register, so is $O(1)$ time.

We can now summarise the time complexity analysis for Algorithm 11 (Replace Equivalent Negative Constraints):

$$\begin{aligned} & O\left(c\left(c\left(r^2 + t.n.r + t.k.r + t^2.r + t.r\right)\right)\right) \\ = \; & O\left(c^2\left(r^2 + t.n.r + t.k.r + t^2.r\right)\right) \\ = \; & O\left(c^2.r^2 + c^2.t.n.r + c^2.t.k.r + c^2.t^2.r\right) \end{aligned}$$

⌟

Lemma 5.3.33. *Let P be a CSP in the mixed representation. Applying Algorithm 11 ('Replace Equivalent Negative Constraints') to P does not change the solutions to P and does not change the set of relational structures permitted by P.*

Proof. Algorithm 11 only replaces constraint relations when they are equivalent, i.e. permit the same assignments, so does not change the set of solutions or permitted relational structures. □

In a later algorithm (Restore Removed Constraints) we will need to know which negatively represented constraints that we have not been able to directly convert after merging must have the same relation in order to permit the desired relational structure. As such, the second pre-processing step is to build a partition for the negatively represented constraints in the input CSP. This is only an approximation of how the negatively represented constraints are partitioned to permit the input relational structure because the parts we create are coarse, and so may have constraints grouped together which should not be. However, it does guarantee that any two constraints which must be in the same part, are in the same part. If Algorithm 11 ('Replace Equivalent Negative Constraints') has been run on the CSP being considered, then the partition also has the property that every part contains either negatively represented or positively represented constraints, but not both.

Data Structure 5.3.34. : Constraint Partition

Let $P = \langle V, D, C \rangle$ be a CSP, and let $C_0 \cap \ldots \cap C_x$ be a partition of the constraints in C. The partition is stored as an associative array mapping each constraint c in C to an array containing the set of constraints which are in the same part as c. By Data Structure 5.3.25, a constraint is of size $O(t.r)$. The key array may contain at most c constraints, so is of size $O(c.t.r)$. Each value in the value array is an array containing at most c constraints, so is also of size $O(c.t.r)$. The value array therefore has size $O\left(c^2.t.r\right)$, and the size of the combined constraint partition structure is $O\left(c.t.r + c^2.t.r\right)$, so $O\left(c^2.t.r\right)$. ⌟

Create Approximate Partition

Algorithm 12: Create Approximate Partition

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
output: Partition - a function from each c in C to a set of constraints

```
1  begin
2      Partition ← ∅
3      foreach c ∈ C do
4          Partition(c) ← ∅
5          if constraint type of c is negative then
6              foreach c′ ∈ C do
7                  if constraint type of c′ is negative then
8                      if Compare Negative Constraint Relations(c, c′) = TRUE then
9                          Partition(c) ← Partition(c) ∪ {c′}
10         else
11             foreach c′ ∈ C do
12                 if constraint type of c′ is positive then
13                     if Compare Positive Constraint Relations(c, c′) = TRUE then
14                         Partition(c) ← Partition(c) ∪ {c′}
15     return Partition
```

Algorithm Analysis 5.3.35. : Create Approximate Partition (Algorithm 12)
By Data Structure 5.3.34, Partition is an associative array that maps from at most c constraints to arrays containing at most c constraints. By Proposition 4.0.16, constructing an empty associative array on line 2 requires $O(1)$ time.

The loop on line 3 will execute at most c times. For each execution of the associative array insertion statement on line 4, the insertion key will be unique and the value is an empty array of size at most $O(c.t.r)$. The key is a constraint of size $O(t.r)$, so by Proposition 4.0.19, each insertion into Partition is $O(t.r)$ time. By Data Structure 5.3.25 the type of a constraint is stored in a single register, so determining the type of a constraint on line 5 is $O(1)$ time.

The loop on line 6 will execute at most c times. Again, determining the type of a constraint on line 7 is $O(1)$ time. On line 7 we call Algorithm 10 (Compare Negative Constraint Relations) whose time complexity is shown to be $O\left(t.r^2 + t.r.n + t.r.k + r.t^2 + k.n.r + k^2.r + k.r^2.n\right)$ in Algorithm Analysis 5.3.31. The comparison of the result to a Boolean value is $O(1)$ time. The Partition associative array has at most c keys which are constraints of size $O(t.r)$, so by Propo-

sition 4.0.17 addressing the corresponding value array is $O(c.t.r)$ time. c' can not already exist in the value array for c, so the union operation on line 9 can be performed as an insert which by Proposition 4.0.8 is $O(t.r)$ time.

The loop on line 11 will execute at most c times. Again, determining the type of a constraint on line 12 is $O(1)$ time. On line 13 we call Algorithm 6 (Compare Positive Constraint Relations) whose time complexity is shown to be $O\left(t^2.r\right)$ in Algorithm Analysis 5.3.27. The comparison of the result to a Boolean value is $O(1)$ time. The Partition associative array has at most c keys which are constraints of size $O(t.r)$, so by Proposition 4.0.17 addressing the corresponding value array is $O(c.t.r)$ time. c' can not already exist in the value array for c, so the union operation on line 14 can be performed as an insert which by Proposition 4.0.8 is $O(t.r)$ time.

We can now summarise the time complexity analysis for Algorithm 12 (Create Approximate Partition):

$$\begin{aligned} & O\left(c\left(t.r + c\left(t.r^2 + t.r.n + t.r.k + r.t^2 + k.n.r + k^2.r + k.r^2.n + t.r\right) + c\left(t^2.r + t.r\right)\right)\right) \\ = \; & O\left(c\left(t.r + c.t.r^2 + c.t.r.n + c.t.r.k + c.r.t^2 + c.k.n.r + c.k^2.r + c.k.r^2.n + c.t.r + c.t^2.r\right)\right) \\ = \; & O\left(c^2.t.r^2 + c^2.t.r.n + c^2.t.r.k + c^2.r.t^2 + c^2.k.n.r + c^2.k^2.r + c^2.k.r^2.n\right) \end{aligned}$$

⌟

Lemma 5.3.36. *Let $P = \langle V, D, C\rangle$ be a CSP given in the mixed representation, and let $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ be a relational structure permitted by P. Given P as input, Algorithm 12 ('Create Approximate Partition') constructs a partition of C such that for each $c_1, c_2 \in C$, if $\sigma(c_1)$ and $\sigma(c_2)$ are both in R_i, for some i, then they are both in the same part in the partition.*

Proof. By Definition 3.4.1, two constraint scopes may only appear in the same relation of a relational structure if they have the same constraint relation. By construction in Algorithm 12, two constraints are only placed into the same part if their constraint relations are equivalent. □

Merge

Given the input CSP P and permitted relational structure $\mathbb{S}$, Algorithm 13 ('Merge') combines the variables of P such that the variables in each interaction region of $\mathbb{S}$ are replaced with a single variable. The constraint relations are combined accordingly by being replaced with their projections onto the subsets of their scopes which are combined to form a region.

The domain for each new region variable cannot be constructed from the Cartesian product of the domains of the original variables which have been combined to form the region because the class containing the original CSP may have unbounded arity, and so an unbounded number of

variables in any given region. Instead, the domain of each region is constructed as the union of the projections of all the merged constraints onto that region. If the region is in the scope of a positively represented constraint, then this projection will contain all the allowed assignments, and so the domain is of type '$-$'. However, if the region is only in the scope of negatively represented constraints, then the set of values which may occur in a solution may not be in the union of the projections. This can be checked by evaluating whether the domain of the region variable contains the same number of values as would have been generated by taking the Cartesian product of the domains of the variables combined to form the region. If all values have been generated by the projections, then the domain is of type '$-$'. However, if this is not the case, then all the missing values are equivalent in so much as there is no constraint which disallows them, and so a partial assignment to all other region variables can always be extended to any one of these values on this region variable, so the domain is of type '$+$'.

Data Structure 5.3.37. : Merged Constraint

Let $c = \langle \sigma, \rho \rangle$ be a constraint, and let $c' = \langle \sigma', \rho' \rangle$ be a merged instance of c. The merged scope, σ', contains the regions on σ, of which there are at most w by Definition 5.3.7. Each region may contain at most r variable names, so by Definition 4.0.4 the size of the array representing σ' is $O(r.w)$. Each tuple in ρ' is formed from the projections of a tuple in ρ onto the regions in σ', so there are at most w projections, each onto at most r variables. By Definition 4.0.4, the array representing each tuple in ρ' will be of size $O(w.r)$. There are at most t tuples in ρ, and there will be the same number in ρ', so by Definition 4.0.4 the size of the array representing ρ' is $O(w.r.t)$. A single register is used to store the representation type of the constraint, so the size of the combined merged constraint structure is therefore $O(r.w + w.r.t + 1)$, so $O(w.r.t)$. ⌟

A constraint and the corresponding merged constraint will contain the same number of variable and value symbols. However, the distribution of these symbols between regions may vary, so the fixed size array structures used in this analysis must allow space for all possible combinations. For example, there will still be a total of r variable names in each structure, but an allowance is made for up to r variable names to be listed in each of the w regions. A more efficient structure could be employed, such as one using delimiters, but would only provide a linear improvement in space.

Algorithm 13: Merge

input : A CSP $\langle V, D, C \rangle$ in the Mixed representation
input : A relational structure $\mathbb{S}$
output: A CSP $\langle V', D', C' \rangle$ - the merged version of $\langle V, D, C \rangle$

```
1  begin
2      Regions ← Generate Interaction Regions (𝕊)
3      V' ← ∅
4      D' ← ∅
5      C' ← ∅
6      foreach v ∈ V do
7          region ← Regions (v) , regionDomain ← ∅
8          set domain type of regionDomain to 'undefined'
9          foreach c ∈ C do                          /* Form the merged domains */
10             if region ⊆ σ (c) then
11                 foreach f ∈ ρ (c) do
12                     regionDomain ← regionDomain ∪ {f_|region}
13                 if representation of c is positive then
14                     set domain type of regionDomain to '−'
15         if domain type of regionDomain is 'undefined' then
16             set domain type of regionDomain to '−'
17             count ← 1          /* If the region contains negative constraints */
18             foreach v ∈ region do          /* determine domain type by counting */
19                 count ← count × |D (v)|
20             if count > |regionDomain| then
21                 set domain type of regionDomain to '+'
22         V' ← V' ∪ {region}
23         D' ← D' ∪ {region ↦ regionDomain}
24     foreach c ∈ C do                              /* Merge the constraints */
25         σ' ← ∅
26         ρ' ← ∅
27         foreach region ∈ Keys (D') do
28             if region ⊆ σ (c) then
29                 σ' ← σ' ∪ {region}
30         foreach f ∈ ρ (c) do
31             ρ' ← ρ' ∪ {⋃_{w∈σ'} {w ↦ f_|w}}
32         C' ← C' ∪ {⟨σ', ρ'⟩}
33     return ⟨V', D', C'⟩
```

Data Structure 5.3.38. : Merged CSP

Let $P = \langle V, D, C \rangle$ be a CSP instance, and let $P' = \langle V', D', C' \rangle$ be a merged instance of P. The variables, V', are now regions which contain at most r variable names, and there may be at most n regions. By Definition 4.0.4, the array representing V' is of size $O(r.n)$. The domain D' is an associative array which contains mappings from the region variables in V' to the projections of the constraint relations in C onto those regions, plus a single register to denote the type of the domain. These projections may each be as large as the projection onto the full scope for c constraints, i.e. may contain the original relations of at most c constraints, in each of which there are at most t tuples of arity r so, by Definition 4.0.4, the size of each value array is $O(r.t.c)$. By Definition 4.0.15, the size of the associative array is $O(r.n + r.t.c.n)$, so $O(r.t.c.n)$. The constraints, C', are stored as an array of c merged constraint structures as described in Data Structure 5.3.37 so, by Definition 4.0.4, the size of this array is $O(w.r.t.c)$. The size of the combined merged CSP structure is therefore $O(r.n + r.t.c.n + w.r.t.c)$, so $O(r.t.c.n + w.r.t.c)$. ⌟

Algorithm Analysis 5.3.39. : Merge (Algorithm 13)

On line 2 we call Algorithm 5 (Generate Interaction Regions) whose time complexity is shown to be $O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r\right)$ in Algorithm Analysis 5.3.23.

The data structures that will contain the merged CSP are initialised on lines 3 to 5. By Data Structure 5.3.37 for a merged constraint and Data Structure 5.3.38 for a merged CSP, the maximum size of each array is known for a given CSP instance. By Proposition 4.0.5, construction of each empty array is $O(1)$ time.

The loop on line 6 will execute at most n times. For each iteration of the loop on line 6, the assignment to *region* on line 7 requires a lookup by variable name in the associative array of Regions. By Proposition 4.0.17, this is $O(n)$ time. By Definition 5.3.38 the domain for a variable in a merged CSP is of size $O(r.t.c)$. By Proposition 4.0.5, construction of regionDomain on line 7 is $O(1)$ time. By Definition 5.3.38 the type of a domain is stored in a single register, so by Proposition 4.0.2 setting the type of the domain is $O(1)$ time.

For each iteration of the loop on line 6, the loop on line 9 will execute at most c times. Each region contains at most n variable names, and each scope contains at most r variable names, so by Proposition 4.0.11 evaluating the subset condition on line 10 by checking for membership of each variable is $O(r.n)$.

On lines 11 and 12 the projection of each tuple in the relation of c onto the region is described using functional notation. For the array based constraint structure defined in Data Structure 5.3.25

the positions of the values in each tuple corresponding to the variables in the region can be determined from the positions of the variables in the scope. By Proposition 4.0.11, finding the position of a variable name in the scope is $O(r)$. This is done for each variable name in the region, so building the list of at most r positions is $O(r^2)$ time. By construction, the variable names in the region will be in the same order as they occur in the scope so, by Proposition 4.0.10, each tuple array can be projected onto these positions in $O(r)$ time. By Proposition 4.0.13 performing the set union to ensure uniqueness when adding a projected tuple to regionDomain (which can contain at most $c.t$ members) is $O(c.t.r^2)$ time. There are at most t tuples, so performing all operations equivalent to lines 11 and 12 is $O(r^2) + O(c.t^2.r^2)$, so $O(c.t^2.r^2)$ time.

By Data Structure 5.3.25 for a constraint, the type of a constraint is stored in a single register, so by Proposition 4.0.2, the condition on line 13 is $O(1)$ time. Similarly, by Data Structure 5.3.26, the type of a domain is stored in a single register, so by Proposition 4.0.2, the assignment on line 14 is $O(1)$ time.

By Data Structure 5.3.25, the type of a domain is stored in a single register, so by Proposition 4.0.2, the condition on line 15 is $O(1)$ time and the assignment on line 16 are both $O(1)$ time. The assignment on line 17 is writing to a single register, so by Proposition 4.0.2 is $O(1)$ time.

The loop on line 18 will execute at most r times as the maximum size of any region is r. By Definition 4.0.4, determining the size of the array representing $D(v)$ is $O(1)$ time. By Proposition 4.0.2, both reading and writing the value of 'count' is $O(1)$ time. By Definition 4.0.1, performing multiplication is $O(1)$ time, so the assignment on line 19 is $O(1)$ time.

By Definition 4.0.4, determining the size of the array representing regionDomain is $O(1)$ time, and by Proposition 4.0.2 reading the value of 'count' is $O(1)$ time, so the condition on line 20 can be evaluated in $O(1)$ time. By Data Structure 5.3.38, the type of a domain is stored in a single register, so by Proposition 4.0.2 the assignment on line 21 is $O(1)$ time.

By Data Structure 5.3.38, V' may contain at most n regions, each of size at most r so, by Proposition 4.0.13, performing the set union on line 22 requires $O(n.r^2)$ time.

On line 23 the domain for the region is recorded in D'. By Data Structure 5.3.38, D' is represented as an associative array which may contain at most n keys of size at most r each mapping to an array if size $O(r.t.c)$. As the region of each variable is processed by the loop on line 6, and variables may share the same region, the region key may already exist in the keys of D'. By Proposition 4.0.11 determining whether the key is already in the array is $O(r.n)$, and if it is not then by Proposition 4.0.18 inserting the key region to regionDomain mapping is

$O(r + c.t.r)$ time. Note that by construction, whenever the same region is processed multiple times, its regionDomain will always be the same. An auxiliary structure could be maintained to ensure that we only process each unique region once, but the cost of this will be equivalent to that of checking membership of the key array of D' and in the worst case each region will be unique.

The loop on line 24 will execute at most c times. Data Structure 5.3.37 states that σ' and ρ' are of size $O(r.w)$ and $O(w.r.t)$ respectively, so by Proposition 4.0.5 initialising them to empty sets on lines 25 and 26 is $O(1)$ time.

By Data Structure 5.3.38, D' is an associative array whose key array contains at most n elements of size at most r, so the loop on line 27 will execute at most n times. The condition on line 27 is checking if the region array is a subset of a scope array. Both a region and a scope may contain at most r variable names. By Proposition 4.0.11 determining whether a variable name is in the scope array requires $O(r)$ time, so checking whether each variable name in a region is in a scope requires $O(r^2)$ time. At most r regions could satisfy the condition on line 28, so by Proposition 4.0.13, the set union on line 29 can be performed in $O(r^2)$ time.

On lines 30 and 31 the merging of each tuple in $\rho(c)$ to correspond to the merged scope is described using functional notation. For the array based constraint and merged constraint structures in Data Structure 5.3.25 and Data Structure 5.3.37 respectively, each merged tuple is constructed by concatenating the projections of an original tuple onto each set of region variables. The positions of the values in each original tuple corresponding to the variables in a region can be determined from the positions of the variables in the original scope. By Proposition 4.0.11 finding the position of a variable name in the original scope requires $O(r)$ time. This is done for each variable name in a region, so building the list of at most r positions for each region in σ' is $O(r^2)$ time. By construction, the variable names in the region will be in the same order as they occur in the scope so, by Proposition 4.0.10, each tuple array can be projected onto these positions in $O(r)$ time. By Definition 5.3.7 there may be at most w regions in σ', so this must be performed for at most w region variables, and for at most t tuples. So, performing the relation merging operation on lines 30 and 31 is $O(r^2.r.w.t)$, so $O(r^3.w.t)$ time.

By Data Structure 5.3.37, the size of a merged constraint is $O(w.r.t)$ By Proposition 4.0.13, performing the set union operation on line 32 is $O\left(c.(w.r.t)^2\right)$, so $O(c.w^2.r^2.t^2)$ time.

We can now summarise the time complexity analysis for Algorithm 13 (Merge):

$$
\begin{aligned}
& O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r\right) + \\
& O\left(n\left(n + c\left(r.n + c.t^2.r^2\right) + r + n.r^2 + r.n + (r + c.t.r)\right) + \right. \\
& \left. c\left(n\left(r^2 + r^2\right).r^3.w.t + c.w^2.r^2.t^2\right)\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r + \right. \\
& \left. n\left(n + c.r.n + c^2.t^2.r^2 + n.r^2 + c.t.r\right) + c\left(n.r^5.w.t + c.w^2.r^2.t^2\right)\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r + \right. \\
& \left. n^2 + c.r.n^2 + c^2.t^2.r^2.n + n.r^2 + c.t.r.n + c.n.r^5.w.t + c^2.w^2.r^2.t^2\right) \\
= \; & O\left(m.r^2.c^2.n^2 + n^6 + m^2.c^2.r^3 + n^4.c^2.m^2.r + \right. \\
& \left. c^2.t^2.r^2.n + c.n.r^5.w.t + c^2.w^2.r^2.t^2\right)
\end{aligned}
$$

⌟

Lemma 5.3.40. *Let $P = \langle V, D, C\rangle$ be a CSP, and let $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ be a relational structure permitted by P. Given P and $\mathbb{S}$ as input, Algorithm 13 ('Merge') outputs the merged CSP $P' = \langle V', D', C'\rangle$ such that, P' permits the relational structure* $\mathrm{Mrg}(\mathbb{S}) = \langle U', R'_1, \ldots, R'_m\rangle$.

Proof. By Definition 3.4.1, for P' to permit the relational structure $\mathrm{Mrg}(\mathbb{S})$ we must have that $V' = U'$ and there is a partition $C' = C'_1 \cup \ldots \cup C'_m$ where for each C_i every constraint in C_i has the same relation and $R'_i = \{\sigma \mid \langle \sigma, \rho\rangle \in C'_i\}$.

By Definition 5.3.8, the universe of $\mathrm{Mrg}(\mathbb{S})$ is the set of regions. Each variable in the original CSP is mapped to the region which contains it in 'Regions'. Each original variable is considered by the loop on line 6, and its corresponding region is added to the set of variables of the merged CSP, V', on line 22 such that V' is the set of regions. Hence, $V' = U'$.

By construction on lines 24 to 32, each constraint $c \in C$ is merged with respect to the regions to produce $c' \in C'$ such that if $\sigma(c) \in R_i$ then $\sigma(c') \in R'_i$. Hence, P' permits the relational structure $\mathrm{Mrg}(\mathbb{S})$. □

Lemma 5.3.41. *Let $P = \langle V, D, C\rangle$ be a CSP, and let $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ be a relational structure permitted by P. Given P and $\mathbb{S}$ as input, Algorithm 13 ('Merge') outputs the merged CSP $P' = \langle V', D', C'\rangle$ such that, any solution to P' can be transformed to a solution to P, and if there is no solution to P', then there is no solution to P.*

Proof. By construction on lines 24 to 32, each tuple in the relation of a constraint is merged with respect to each region of the variables in the scope such that the set of mappings from the variables in the region to values is replaced with a single mapping from the region to that set of mappings. A constraint can therefore be unmerged in polynomial time by replacing the single

mapping with the set of mappings it maps to. As such, each constraint in C is equivalent to the merged constraint it corresponds to in C' as there is a polynomial time transformation between assignments.

By Definition 5.3.6, variables can only be members of the same region of $\mathbb{S}$ if they never appear separately in any scope of a CSP that permits $\mathbb{S}$ and are only members of a single region. So, by construction of P', variables are merged consistently between constraints.

Joining all constraints in C gives the set of solutions to P, and joining all constraints in C' gives the set of solutions to P'. As all constraints are individually equivalent with their merged counterparts, and variables merged are consistent between constraints, there is a polynomial transformation between the solutions of P and P' and they are equivalent. □

Create Improved Merged Partition

Once the original CSP has been merged by Algorithm 13, the constraint partition constructed by Algorithm 12 needs to be reevaluated. The first reason for this is that the constraints need to be replaced with the merged versions. This is straightforward as there is a one-to-one mapping between original and merged constraints, and our algorithm assumes that the constraints are kept in the same order. More importantly, merging of the CSP with respect to the interaction regions of the relational structure provides more information about the actual partitioning of the constraints. As such, we can now identify the parts, or sets of constraints which much share the same relation in order to permit the desired relational structure, with greater accuracy. This reevaluation of the partition is performed by Algorithm 14 ('Create Improved Merged Partition').

Data Structure 5.3.42. : Merged Constraint Partition

Let $P = \langle V, D, C \rangle$ be a merged CSP, and let $C_0 \cap \ldots \cap C_x$ be a partition of the merged constraints in C. The partition is stored as an associative array mapping each merged constraint c in C to an array containing the set of merged constraints which are in the same part as c. By Data Structure 5.3.37, a merged constraint is of size $O(w.r.t)$. The key array may contain at most c merged constraints, so is of size $O(c.w.r.t)$. Each value in the value array is an array containing at most c merged constraints, so is also of size $O(c.w.r.t)$. The value array therefore has size $O\left(c^2.w.r.t\right)$, and the size of the combined constraint partition structure is $O\left(c.w.r.t + c^2.w.r.t\right)$, so $O\left(c^2.w.r.t\right)$. ⌟

Algorithm 14: Create Improved Merged Partition

input : A CSP $\langle V, D, C \rangle$ in the mixed representation
input : A CSP $\langle V', D', C' \rangle$ - the merged version of $\langle V, D, C \rangle$
input : Partition - a function from each c in C to a set of constraints
output: MergedPartition - a function from each c' in C' to a set of constraints

```
1  begin
2      MergedPartition ← ∅
3      foreach c ∈ C do
4          i ← position of c in C
5          mrgc ← C'[i]
6          MergedPartition(mrgc) ← ∅
7          foreach c' ∈ Partition(c) do
8              j ← position of c' in C
9              mrgc' ← C'[j]
10             compatible ← |σ(mrgc)| = |σ(mrgc')|
11             if compatible = TRUE then                    /* If the scope sizes match */
12                 foreach q = 0...|σ(mrgc)| − 1 do         /* check the merged positions */
13                     if |σ(mrgc)[q]| ≠ |σ(mrgc')[q]| then
14                         compatible ← FALSE
15                     if compatible = TRUE then
16                         foreach y = 0...|σ(mrgc)[q]| − 1 do
17                             foreach z = 0...|σ(mrgc')[q]| − 1 do
18                                 if position of σ(mrgc)[q][y] in σ(c) ≠ position of
                                   σ(mrgc')[q][z] in σ(c') then
19                                     compatible ← FALSE
20             if compatible = TRUE then
21                 MergedPartition(mrgc) ← MergedPartition(mrgc) ∪ {mrgc'}
22     return MergedPartition
```

Algorithm Analysis 5.3.43. : Create Improved Merged Partition (Algorithm 14)

By Data Structure 5.3.42, MergedPartition is an associative array that maps from at most c merged constraints to arrays containing at most c merged constraints. By Proposition 4.0.16, constructing an empty associative array on line 2 requires $O(1)$ time.

The loop on line 3 will execute at most c times. By Data Structure 5.3.26, C is an array of c constraints of size at most $O(t.r)$. By Proposition 4.0.11 finding the index of an element in C on line 4 requires $O(c.t.r)$ time. By Data Structure 5.3.38, C' is an array of c merged constraints of size at most $O(w.r.t)$, so by Proposition 4.0.6 addressing a merged constraint in C' by index on line 5 is $O(1)$ time. For each execution of the associative array insertion statement on line 6, the

insertion key will be unique and the value is an empty array of size at most $O(c.w.r.t)$. The key is a constraint of size $O(w.r.t)$, so by Proposition 4.0.19, each insertion into Partition is $O(w.r.t)$ time.

The keys of Partition are constraints, so by Proposition 4.0.17, addressing a value array by key in Partition on line 7 is $O(c.t.r)$. The loop on line 7 will execute at most c times. By Proposition 4.0.11 finding the index of an element in C on line 8 requires $O(c.t.r)$ time. By Proposition 4.0.6 addressing a merged constraint in C' by index on line 9 is $O(1)$ time. The scopes of the merged constraints on line 10 are stored as arrays, so by Definition 4.0.4 determining their size is $O(1)$ time, and assigning a Boolean value to 'compatible' is $O(1)$ time.

Evaluating the Boolean value on line 11 is $O(1)$ time. The loop on line 12 will execute at most w times. By Definition 4.0.4 both addressing by index and determining the size of an array requires $O(1)$ time, so evaluating the condition on line 13 is $O(1)$ time. Setting the Boolean value on line 14 is $O(1)$ time.

Evaluating the Boolean value on line 15 is $O(1)$ time. The loop on line 16 will execute at most r times. The loop on line 17 will execute at most r times. By Proposition 4.0.6 addressing an array on line 18 is $O(1)$ time. By Proposition 4.0.11 finding the index of a variable in the scope of a constraint is $O(r)$ time. Comparing the indices on line 18 requires $O(1)$ time. Setting the Boolean value on line 19 is $O(1)$ time.

Evaluating the Boolean value on line 20 is $O(1)$ time. The addressing into MergedPosition on line 21 is always addressing the value array at the last inserted key at this point in the algorithm, so by Proposition 4.0.6 the value array can always be addressed at an index corresponding to the current size of the array in $O(1)$ time. The union operation on line 21 will always be adding a value which cannot already be in the set, so can be performed as an insertion which by Proposition 4.0.8 requires $O(w.r.t)$ time.

We can now summarise the time complexity analysis for Algorithm 14 (Create Improved Merged Partition):

$$\begin{aligned} & O\left(c\left(c.t.r + w.r.t + c.t.r + c\left(c.t.r + w.r.r.2r + w.r.t\right)\right)\right) \\ = \; & O\left(c\left(c^2.t.r + c.w.r^3 + c.w.r.t\right)\right) \\ = \; & O\left(c^3.t.r + c^2.w.r^3 + c^2.w.r.t\right) \end{aligned}$$

⌟

Lemma 5.3.44. *Let $P = \langle V, D, C\rangle$ be a CSP in the mixed representation, let $\mathbb{S} = \langle U, R_1, \ldots, R_m\rangle$ be a relational structure permitted by P, and let Partition be a partition of $C'' \subseteq C$ constructed by Algorithm 12 ('Create Approximate Partition') for P. Let $P' = \langle V', D', C'\rangle$ be a merged CSP in the mixed representation, and let $\mathrm{Mrg}(\mathbb{S}) = \langle U', R'_1, \ldots, R'_m\rangle$ be a relational structure permitted*

by P. Given P, P' and Partition as input, Algorithm 14 ('Create Improved Merged Partition') constructs a partition of C' such that for each $c'_1, c'_2 \in C'$, if $\sigma(c'_1)$ and $\sigma(c'_2)$ are both in R'_i, for some i, then they are both in the same part in the partition.

Proof. By Definition 5.3.8, the merged structure of a relational structure $\mathbb{S}$ is constructed such that each $R'_i \in \text{Mrg}(\mathbb{S})$ is the merged version of R_i. As such, two merged scopes can only be in the same R'_i if their corresponding unmerged scopes were in the same R_i. By Lemma 5.3.36, Partition is constructed such that for each $c_1, c_2 \in C''$, c_1 and c_2 are in the same part if $\sigma(c_1)$ and $\sigma(c_2)$ are both in some R_i.

To satisfy $\text{Mrg}(\mathbb{S})$, the merged scopes in P' can only not be in the same partitions as their corresponding unmerged scopes in P if the construction of the merged structure $\text{Mrg}(\mathbb{S})$ has merged different positions of the original scopes. Algorithm 14 enforces this by checking that the scopes contain the same number of merged regions on line 10, that the sizes of the regions are the same on line 13, and that the positions of variables in the original scopes which are now in the regions are the same on line 18. □

Convert to Positive

We now employ Algorithm 15 ('Convert to Positive') as a first pass in generating a positively represented version of the merged CSP. This algorithm inverts the relations for any negatively represented constraints for which there is enough domain knowledge to generate the Cartesian product of possible allowed assignments, and removes those constrains for which there is not enough information. As the constraints removed are those which contain a variable in their scope which is only involved in negatively represented constraints, and for which the lack of domain information indicates must contain some value for which every solution to the remaining variables must extend (i.e. has a '+' domain), these constraints impose no restriction on the remainder of their scope that is not already expressed by a constraint which can be converted (or is already in the positive representation). This means that the set of solutions to the variables whose domains are of type '$-$' is not affected by the removal of these constraints.

The domains of type '+' are then converted to the special type '$*$' which contains a single special value that is not restricted by any of the constraints, and can therefore always be assigned.

Algorithm 15: Convert to Positive

input : A CSP $\langle V, D, C \rangle$ - a merged CSP in the Mixed representation
output: A CSP $\langle V', D', C' \rangle$ - a merged CSP in the Positive representation
output: RemovedConstraints - a set of Negatively represented constraints

```
1  begin
2      V' ← V
3      D' ← D
4      RemovedConstraints ← ∅
5      foreach v ∈ V' do
6          if domain type of D'(v) is '+' then          /* Can only be involved in */
7              foreach c ∈ C do                          /* negative constraints */
8                  if v ∈ σ(c) then        /* and these can't be directly converted */
9                      RemovedConstraints ← RemovedConstraints ∪ {c}
10                     C ← C \ {c}
11             set domain type of D'(v) to '∗'
12     C' ← ∅
13     foreach c ∈ C do
14         if constraint type of c is 'Negative' then          /* Invert the relation */
15             a ← |σ(c)|
16             newRelation ← {⟨f_1, ..., f_a⟩ | f_i ∈ D'(σ(c)[i]) for all 1 ≤ i ≤ a}
17             foreach tuple ∈ ρ(c) do
18                 newRelation ← newRelation \ {tuple}
19             c' ← ⟨σ(c), newRelation⟩
20             set constraint type of c' to 'Positive'
21             C' ← C' ∪ {c'}
22         else
23             C' ← C' ∪ {c}                                  /* Already positive */
24     return ⟨V', D', C'⟩
25     return RemovedConstraints
```

When converting a mixed instance to the positive representation, more space may be required to store the tuples than is available in the merged constraint data structure (Data Structure 5.3.37) as the Cartesian product of the inferred domains may contain more than t tuples. As such, we now introduce the Maximal Merged data structures.

Data Structure 5.3.45. : Maximal Merged Constraint

Let $c = \langle \sigma, \rho \rangle$ be a constraint, and let $c' = \langle \sigma', \rho' \rangle$ be a merged instance of c. The maximal merged constraint structure is the same as the merged constraint structure (Data Structure 5.3.37) except that it allows the relation ρ' to contain the maximum possible number of tuples, that is the number of tuples in the Cartesian product of the domains of the variables in σ'. As per Data Structures 5.3.37 and 5.3.38, the size of the scope array is $O(r.w)$, and each domain in a merged CSP may contain at most $t.c$ elements of size $O(r)$. By Definition 5.3.7, the arity of a merged constraint is at most the interaction width, w, so a relation array formed by the Cartesian product of the domains may contain at most $(t.c)^w$ tuple arrays, each of which contains at most w elements of size $O(r)$, so is of size $O(t^w.c^w.r.w)$. The combined size of the maximal merged constraint structure is therefore $O(r.w + t^w.c^w.r.w)$, so $O(t^w.c^w.r.w)$. ⌟

Data Structure 5.3.46. : Maximal Merged CSP

Let $P = \langle V, D, C \rangle$ be a CSP instance, and let $P' = \langle V', D', C' \rangle$ be a merged instance of P. The maximal merged CSP structure is the same as the merged CSP structure (Data Structure 5.3.38) except that the constraint array contains maximal merged constraints. As such, the size of V' is $O(r.n)$, and the size of D' is $O(r.n + r.t.c.n)$, so $O(r.t.c.n)$. C' is now stored as an array of at most c merged constraint structures as described in Data Structure 5.3.45, so by Definition 4.0.4 is of size $O(c(t^w.c^w.r.w))$, so $O\left(t^w.c^{w+1}.r.w\right)$. The size of the combined maximal merged CSP structure is therefore $O\left(r.n + r.t.c.n + t^w.c^{w+1}.r.w\right)$, so $O\left(r.t.c.n + t^w.c^{w+1}.r.w\right)$. ⌟

Algorithm Analysis 5.3.47. : Convert to Positive (Algorithm 15)

The input CSP, $\langle V, D, C \rangle$, is not required after execution of this algorithm and will be modified. Setting V' and D' on line 2 and 3 requires setting a single register for each to the base address of V and D respectively, so by Proposition 4.0.2 requires $O(1)$ time. RemovedConstraints on line 4 will hold any constraints removed from C, so is an array of size $O(w.r.t.c)$. By Proposition 4.0.5, construction of RemovedConstraints in its initial empty state requires $O(1)$ time.

By Data Structure 5.3.38, there are at most n variables (regions) in V', so the loop on line 5 will execute at most n times. By Proposition 4.0.17 finding $D'(v)$ in the associative array representing

the domain requires $O\left(r.n^2\right)$ time, and by Proposition 4.0.2 reading the single register required for the condition on line 6 is then $O\left(1\right)$ time.

The loop on line 7 will execute at most c times. On line 8, v is a region so has size $O\left(r\right)$, and $\sigma\left(c\right)$ may contain at most w regions. By Proposition 4.0.11, determining whether v is a member of $\sigma\left(c\right)$ requires $O\left(r.w\right)$ time. Each constraint can only be removed from C at most once, so the set union on line 9 to add the constraint to RemovedConstraints can be performed by simply inserting the constraint, which by Proposition 4.0.8 is $O\left(w.r.t\right)$ time. By Proposition 4.0.14, the set difference operation on line 10 is $O\left(1.c.\left(w.r.t\right)^2\right)$, so $O\left(c.w^2.r^2.t^2\right)$ time.

By Proposition 4.0.17 finding $D'\left(v\right)$ in the associative array representing the domain requires $O\left(r.n^2\right)$ time, and by Proposition 4.0.2 setting the single register for the domain type on line 11 is $O\left(1\right)$ time.

Any negatively represented merged constraints remaining in C will now be inverted to form their positive equivalents. An inverted constraint may contain at most all of the tuples produced from the Cartesian product of the domains of the variables in its scope, which may be more than t. As such, $\langle V', D', C'\rangle$ is required to be implemented as a maximal merged CSP (Data Structure 5.3.46. V' and D' remain unchanged, but C' is now of size $O\left(t^w.c^{w+1}.r.w\right)$. By Proposition 4.0.5, constructing the empty array for C' on line 12 is $O\left(1\right)$ time.

The loop on line 13 will execute at most c times. Determining the constraint type of c on line 14 requires reading a single register which by Proposition 4.0.2 is $O\left(1\right)$ time. By Definition 4.0.4 the number of elements in an array is stored in a register of the array so, by Proposition 4.0.2, determining the number of elements in a scope on line 15 is $O\left(1\right)$ time.

By Definition 5.3.7, the arity of a merged constraint is at most w, so newRelation can be constructed on line 16 by constructing a series of at most w intermediate relations, the first of which contains the set of unary tuples corresponding to each domain value in the domain of the first scope member. Each subsequent relation then contains the set of tuples formed by extending each tuple in its predecessor to each domain value in the domain of the next scope member. The final relation constructed in this manner will be newRelation. Each domain may contain at most $t.c$ elements of size at most r so, in general, the construction of each intermediate relation requires reading $\left(t.c\right)^{w-1}$ tuples of size $O\left(w.r\right)$ and then writing $\left(t.c\right)^{w}$ tuples of size $O\left(w.r\right)$. By Proposition 4.0.2, each construction in general is $O\left(t^{w-1}.c^{w-1}.w.r\right) + O\left(t^w.c^w.w.r\right)$, so $O\left(t^w.c^w.w.r\right)$ time. This is performed at most w times, so requires $O\left(t^{w+1}.c^{w+1}.w^2.r\right)$ time in total.

The loop on line 17 will execute at most t times. By Proposition 4.0.14, the set difference

operation on line 18 requires $O\left(1.\left(t.c\right)^{w}.\left(w.r\right)^{2}\right)$, so $O\left(t^{w}.c^{w}.w^{2}.r^{2}\right)$ time.

On line 19, c' is constructed by copying the scope of c and newRelation to form a new constraint structure as per Data Structure 5.3.45. By Proposition 4.0.2, both reading and writing of the scope and relation requires time $O\left(r.w\right)$ and $O\left(t^{w}.c^{w}.r.w\right)$ respectively, so $O\left(t^{w}.c^{w}.r.w\right)$ time overall. By Proposition 4.0.2 setting the single register for the constraint type on line 27 is $O\left(1\right)$ time. As each constraint in C is processed only once by the loop on line 13, the set union on line 21 can be performed by simply inserting c' in to C, which by Proposition 4.0.8 requires time $O\left(t^{w}.c^{w}.r.w\right)$.

On line 23 a merged constraint (Data Structure 5.3.37) is added to a set of maximal merged constraints (Data Structure 5.3.45). As the maximal merges constraint structure is larger, c will fit in the space reserved for a each constraint in C'. Also, as the only difference between the two structures is the maximum number of tuples in the relation array, c can be converted to the correct type simply by updating the maximum size register on the array once it is in place, which by Proposition 4.0.2 is $O\left(1\right)$ time. As with the set union on line 28, each constraint in C is only processed once, so c can simply be inserted into C', which by Proposition 4.0.8 requires $O\left(w.r.t\right)$ time.

We can now summarise the time complexity analysis for Algorithm 15 (Convert to Positive):

$$\begin{aligned}
& O\left(n\left(r.n^{2}+c\left(r.w+w.r.t+c.w^{2}.r^{2}.t^{2}\right)+r.n^{2}\right)+\right.\\
& \left.c\left(t^{w+1}.c^{w+1}.w^{2}.r+t\left(t^{w}.c^{w}.w^{2}.r^{2}\right)+t^{w}.c^{w}.r.w+t^{w}.c^{w}.r.w+w.r.t\right)\right)\\
= \;& O\left(n\left(r.n^{2}+c\left(w.r.t+c.w^{2}.r^{2}.t^{2}\right)\right)+c\left(t^{w+1}.c^{w+1}.w^{2}.r+t\left(t^{w}.c^{w}.w^{2}.r^{2}\right)\right)\right)\\
= \;& O\left(n\left(r.n^{2}+c.w^{2}.r^{2}.t^{2}\right)+c\left(t^{w+1}.c^{w+1}.w^{2}.r+t^{w+1}.c^{w}.w^{2}.r^{2}\right)\right)\\
= \;& O\left(r.n^{3}+n.c.w^{2}.r^{2}.t^{2}+t^{w+1}.c^{w+1}.w^{2}.r+t^{w+1}.c^{w}.w^{2}.r\right)
\end{aligned}$$

⌟

Lemma 5.3.48. *Let $P=\langle V,D,C\rangle$ be a merged CSP as constructed by Algorithm 13 ('Merge'). Given P as input, Algorithm 15 ('Convert to Positive') outputs the CSP $P'=\langle V',D',C'\rangle$ such that any assignment to $V'\setminus\{v'\mid D'\left(v'\right)$ is of type '$*$'$\}$ that is a partial assignment to P' is also a partial assignment to P.*

Proof. Let v be a variable in V such that $D\left(v\right)$ is of type '+', and let c be a constraint such that $v\subseteq\sigma\left(c\right)$. c must be in the negative representation by construction of P, and must allow all assignments to $\sigma\left(c\right)\setminus\{v\}$ as if it did not then there must be a disallowed assignment listed in the relation for each value in $D\left(v\right)$ which could be in a solution, and hence $D\left(v\right)$ would be of type '−'. As such, each negative constraint with a variable in its scope that has a domain of type '+' imposes no restrictions on the allowed assignments to any subset of its scope. $V'=V$ as Algorithm 13 does not alter the set of variables. All of the variables with domains of type '+'

are changed to type '$*$' on line 11 of Algorithm 15, so removing any such constraint cannot affect partial assignments over $V' \setminus \{v' \mid D'(v') \text{ is of type '} * \text{'}\}$. □

Lemma 5.3.49. *Let $P = \langle V, D, C \rangle$ be a merged CSP as constructed by Algorithm 13 ('Merge'), and let* $\mathrm{Mrg}(\mathbb{S})$ *be a relational structure permitted by P. Given P as input, Algorithm 15 ('Convert to Positive') outputs the CSP $P' = \langle V', D', C' \rangle$ and set of removed constraints C^- such that the CSP $\langle V', D', C' \cup C^- \rangle$ permits* $\mathrm{Mrg}(\mathbb{S})$.

Proof. $V' = V$ as Algorithm 15 does not alter the set of variables. $C' \cup C^- = C$ as for each $c \in C$ Algorithm 15 either removes c from C and places it into C^-, changes the relation to an equivalent positive relation, or does not alter c. As such, any relational structure permitted by P is permitted by $\langle V', D', C' \cup C^- \rangle$. □

Restore Removed Constraints

Algorithm 15 ('Convert to Positive') has generated a positively represented CSP whose solutions can be transformed into solutions to the original unmerged instance. However, by removing constraints it no longer permits the same class of relational structures, so no longer permits the merged structure that we require of it in order to appeal to Grohe's Theorem 3.4.9 in our final proof. To correct this, we run Algorithm 16 which uses the information we constructed earlier in the merged partition to synthesise constraint relations for the removed constraints which will preserve solutions and also meet the conditions necessary for the positive instance to permit our desired relational structure.

The domains of all the variables involved in the removed constraints have domains which are either of type '$-$' or '$*$', so we can build a Cartesian product from these domains. Replacing each removed constraint with 'anything goes' over its scope would not affect the solutions, but is unlikely to meet the requirements needed to permit the desired relational structure as either not all constraints in the same part of the partition have been removed, or some removed constraints in the same part have different domain types for some position in their scopes. As such, Algorithm 16 collates information about each constraint in the same part in order to generate a single relation which satisfies all required restrictions. This is possible because the construction of the merged structure ensures that all regions impose the same restriction.

Algorithm 16: Restore Removed Constraints

input : A CSP $\langle V, D, C\rangle$ - a merged CSP in the positive representation
input : RemovedConstraints - a set of merged constraints in the negative representation
input : MergedPartition - a function from each c' in C' to a set of constraints
output: A CSP $\langle V, D, C\rangle$ - a merged CSP in the positive representation

```
1  begin
2      foreach c ∈ RemovedConstraints do
3          foreach c' ∈ MergedPartition(c) do          /* Check whether some part */
4              foreach i = 0 ... |σ(c)| − 1 do             /* has the full domain */
5                  if domain type of D(σ(c')[i]) is '−' then
6                      D(σ(c)[i]) ← D(σ(c')[i])
7          a ← |σ(c)|
8          newRelation ← {⟨f_1, ..., f_a⟩ | f_i ∈ D'(σ(c)[i]) for all 1 ≤ i ≤ a}
9          hasStar ← FALSE
10         foreach v ∈ σ(c) do
11             if domain type of D(v) is '∗' then
12                 hasStar ← TRUE
13         if hasStar = FALSE then  /* The constraint can be directly converted */
14             foreach t ∈ ρ(c) do
15                 newRelation ← newRelation \ {t}
16         c ← ⟨σ(c), newRelation⟩
17         set constraint type of c to positive
18         C = C ∪ {c}
19     return ⟨V, D, C⟩
```

Algorithm Analysis 5.3.50. : Restore Removed Constraints (Algorithm 16)

The loop on line 2 will execute at most c times. By Data Structure 5.3.42, MergedPartition is an associative array mapping from at most c merged constraints to arrays each containing at most c merged constraints. By Data Structure 5.3.37, a merged constraint is of size $O(w.r.t)$, so by Proposition 4.0.17 addressing the corresponding value array for a given key on line 3 is $O(c.w.r.t)$ time. The loop on line 3 will execute at most c times, and the loop on line 4 will execute at most w times.

The input CSP $\langle V, D, C\rangle$ is implemented as a maximal merged CSP (Data Structure 5.3.46, so D is an associative array mapping at most n region keys of size $O(r)$ to values of size $O(r.t.c.n)$. By Proposition 4.0.6, addressing an array by index is $O(1)$, so by Proposition 4.0.17 finding a domain in the associative array representing the domain on line 5 requires $O\left(n.r^2\right)$ time, and by Proposition 4.0.2 reading the single register required for the condition on line 5 is then $O\left(1\right)$ time. Similarly, finding a domain in the associative array representing the domain on line 6requires

$O\left(n.r^2\right)$ time, and by Proposition 4.0.2 replacing the domain values requires $O\left(r.t.c\right)$ time.

By Definition 4.0.4 the number of elements in an array is stored in a register of the array so, by Proposition 4.0.2, determining the number of elements in a scope on line 7 is $O\left(1\right)$ time. By Definition 5.3.7, the arity of a merged constraint is at most w, so newRelation can be constructed on line 8 by constructing a series of at most w intermediate relations, the first of which contains the set of unary tuples corresponding to each domain value in the domain of the first scope member. Each subsequent relation then contains the set of tuples formed by extending each tuple in its predecessor to each domain value in the domain of the next scope member. The final relation constructed in this manner will be newRelation. Each domain may contain at most $t.c$ elements of size at most r so, in general, the construction of each intermediate relation requires reading $\left(t.c\right)^{w-1}$ tuples of size $O\left(w.r\right)$ and then writing $\left(t.c\right)^{w}$ tuples of size $O\left(w.r\right)$. By Proposition 4.0.2, each construction in general is $O\left(t^{w-1}.c^{w-1}.w.r\right) + O\left(t^{w}.c^{w}.w.r\right)$, so $O\left(t^{w}.c^{w}.w.r\right)$ time. This is performed at most w times, so requires $O\left(t^{w+1}.c^{w+1}.w^2.r\right)$ time in total.

Setting the Boolean value on line 9 is $O(1)$ time. The loop on line 10 will execute at most w times. By Proposition 4.0.17 finding a domain in the associative array representing the domain on line 11 requires $O\left(n.r^2\right)$ time, and by Proposition 4.0.2 reading the single register required for the condition on line 11 is then $O\left(1\right)$ time. Setting the Boolean value on line 12 is $O(1)$ time.

Evaluating the Boolean value on line 13 is $O(1)$ time. The loop on line 14 will execute at most t times. By Proposition 4.0.14, the set difference operation on line 15 requires $O\left(1.\left(t.c\right)^{w}.\left(w.r\right)^{2}\right)$, so $O\left(t^{w}.c^{w}.w^2.r^2\right)$ time.

On line 16, c is constructed by copying the scope of c and newRelation to form a new constraint structure as per Data Structure 5.3.45. By Proposition 4.0.2, both reading and writing of the scope and relation requires time $O\left(r.w\right)$ and $O\left(t^{w}.c^{w}.r.w\right)$ respectively, so $O\left(t^{w}.c^{w}.r.w\right)$ time overall. By Proposition 4.0.2 setting the single register for the constraint type on line 17 is $O\left(1\right)$ time. As each constraint in C is processed only once by the loop on line 2, the set union on line 18can be performed by simply inserting c in to C, which by Proposition 4.0.8 requires time $O\left(t^{w}.c^{w}.r.w\right)$.

We can now summarise the time complexity analysis for Algorithm 16 (Restore Removed Constraints):

$$\begin{aligned}
& O\left(c\left(c.w.r.t + c\left(w\left(n.r^2 + n.r^2 + r.t.c\right)\right) + t^{w+1}.c^{w+1}.w^2.r + w\left(n.r^2\right)\right.\right. \\
& \left.\left. + t\left(t^w.c^w.w^2.r^2\right) + t^w.c^w.r.w + t^w.c^w.r.w\right)\right) \\
= \; & O\left(c\left(c.w.r.t + c\left(w.n.r^2 + w.r.t.c\right) + t^{w+1}.c^{w+1}.w^2.r + w.n.r^2\right.\right. \\
& \left.\left. + t^{w+1}.c^w.w^2.r^2 + t^w.c^w.r.w\right)\right) \\
= \; & O\left(c\left(c.w.n.r^2 + w.r.t.c^2 + t^{w+1}.c^{w+1}.w^2.r + t^{w+1}.c^w.w^2.r^2\right)\right) \\
= \; & O\left(c^2.w.n.r^2 + w.r.t.c^3 + t^{w+1}.c^{w+2}.w^2.r + t^{w+1}.c^{w+1}.w^2.r^2\right)
\end{aligned}$$

⌟

Lemma 5.3.51. *Let P be a CSP in the mixed representation that contains no negatively represented constraints for which there is a positive constraint with the equivalent relation also in P, and let Partition be the partition generated for P by Algorithm 12 ('Create Approximate Partition'). Let P' be the merged CSP generated from P by Algorithm 13 ('Merge'), and MergedPartition be the partition generated for P' by Algorithm 14 ('Create Improved Merged Partition'). Let $P'' = \langle V'', D'', C'' \rangle$ be a merged CSP in the positive representation and C^- be a set of negatively represented constraints as constructed for P'' by Algorithm 15 ('Convert to Positive'). Let* $\mathrm{Mrg}(\mathbb{S})$ *be a relational structure permitted by $\langle V'', D'', C'' \cup C^- \rangle$. Given P'', C^- and MergedPartition as inputs, Algorithm 16 ('Restore Removed Constraints') returns a CSP $P'' = \langle V'', D'', C^+ \rangle$ such that:*

1. *P'' is in the positive representation*
2. *P'' permits* $\mathrm{Mrg}(\mathbb{S})$
3. *any assignment to $V'' \setminus \{v'' \mid D''(v'') \text{ is of type '} * \text{'}\}$ that is a partial assignment to P'' is also a partial assignment to P'.*

Proof. P'' will be in the positive representation as all constraints in C^- are converted to the positive representation by Algorithm 16 ('Restore Removed Constraints') before being added to C^+.

By construction in Algorithm 12 ('Create Approximate Partition'), Partition contains all negatively represented constraints in P. So, by construction in Algorithm 14 ('Create Improved Merged Partition'), MergedPartition contains all negatively represented constraints in P'. Algorithm 15 does not add negatively represented constraints, so MergedPartition contains all constraints which are in C^-.

By Lemma 5.3.49 and Lemma 5.3.44, P'' will permit Mrg($\mathbb{S}$) if each negative constraint in C^- is converted to positive in such a way as its relation is equivalent to all other constraints in the same part of MergedPartition. This is required to satisfy item 2.

By the argument in the proof of Lemma 5.3.48, replacing each constraint in C^- with the positively represented constraint whose allowed tuples are the Cartesian product of the domains of the variables in the scope will satisfy item 3, but may violate the partition condition required for item 2.

To meet the conditions required for both items 1 and 2 to be satisfied, the positive constraints to replace those in C^- must be synthesised such that they they have the equivalent relation as any other constraint in the same part of MergedPartition, and only place a restriction on any subset of their scope where the correct restriction is known.

Algorithm 16 achieves this in the following way for each $c^- \in C^-$:

- If any other constraints in the same part of MergedPartition have a domain of type '$-$' at the same position that c^- has a domain of type '$*$', then this '$*$' domain can be replaced with the '$-$'. The variable with the '$-$' domain must have been in a positively represented constraint and so this domain lists all of the allowed values that were considered equivalent but not listed in the '$+$' domain that was converted to a '$*$' by Algorithm 15 ('Convert to Positive'). This is performed on lines 3 to 6.

- The relation of c^- is now replaced with the Cartesian product of its domains. If its scope no longer contains any variables whose domains are of type '$*$', then the disallowed assignments must be removed from the new product relation so as to provide the correct restrictions to the values for the variables which previously has a domain of type '$*$'. The constraints in the same part of MergedPartition all had the same set of disallowed assignments by construction of MergedPartition, so this is consistent for all constraints in the same part. This is performed on lines 7 to 16.

□

Unmerge Solution

The positive instance we have constructed can now be solved. By Grohe's result 3.4.9, finding a solution is tractable for the class of CSPs whose arity is bounded, if the core of the relational structure they permit has bounded treewidth. This is true for any instance constructed using our method if the original instance is in the class of CSPs with bounded interaction width.

Algorithm 17: Unmerge Solution

input : A CSP $\langle V, D, C\rangle$ in the Mixed representation
input : A CSP $\langle V', D', C'\rangle$ - a merged CSP in the Positive representation
input : C'' - a set of merged constraints in the Negative representation
input : S' - A solution to $\langle V', D', C'\rangle$
output: S - a solution to $\langle V, D, C\rangle$

```
1  begin
2      S ← ∅
3      foreach v′ ∈ V′ do
4          if domain type of D′(v′) is not '∗' then
5              S ← S ∪ S′(v′)
6          else
7              disallowedAssignments ← ∅
8              foreach c″ ∈ C″ do
9                  disallowedAssignments ← disallowedAssignments ∪ Π_{v′} ρ(c″)
10             Stack ← [∅]
11             found ← false
12             repeat                                  /* Generate and test assignments */
13                 currentAssignment ← pop from Stack
14                 if |currentAssignment| = |v′| then
15                     if currentAssignment ∉ disallowedAssignments then
16                         found ← true
17                 else
18                     v ← v′[ |currentAssignment| ]
19                     foreach k ∈ D(v) do
20                         push currentAssignment ∪ {v ↦ k} to Stack
21             until found = true
22             S ← S ∪ currentAssignment
23     return S
```

Data Structure 5.3.52. : Assignment

Let $P = \langle V, D, C\rangle$ be a CSP instance. An assignment to $X \subseteq V$ is stored as two arrays. The first array contains the at most n variable names in X, and the second contains at most n values such that for each position, $i = 0 \ldots |X| - 1$, the value in position i is in the domain of the variable in position i of the variable array. By Definition 4.0.4, the size of each array is $O(n)$. ⌟

Data Structure 5.3.53. : Merged Assignment Let $P = \langle V, D, C\rangle$ be a merged CSP instance. Similar to Data Structure 5.3.52, a merged assignment to $X \subseteq V$ is stored as two arrays. The first array contains the at most n region variables in X, which by Data Structure 5.3.38 are

of size $O(r)$, so by Definition 4.0.4 the array is of size $O(r.n)$. The second array contains the at most n values such that for each position, i, the value in position i is in the domain of the region variable in position i of the variable array. Each domain value is of size $O(r)$, so the value array is of size $O(r.n)$. ⌟

Algorithm Analysis 5.3.54. : Unmerge (Algorithm 17)

By Data Structure 5.3.52, the size of a solution assignment to an unmerged CSP is $O(n)$, so by Proposition 4.0.5 constructing the variable and value arrays for S on line 2 is $O(1)$ time.

There may be as many variables in a merged CSP as in the original instance, so the loop on line 3 will execute at most n times. By Proposition 4.0.17, finding the base address of a domain in the maximal merged CSP (Data Structure 5.3.46) for a given merged variable requires $O\left(r.n^2\right)$ time, and by Proposition 4.0.2, reading the single register for the domain type on line 4 is $O(1)$ time.

In Data Structure 5.3.53, each element in the value array of a merged solution assignment is an assignment for the variables of the region in the corresponding variable array. By Proposition 4.0.11, finding the index of v' in the variable array of S' requires $O\left(r.n^2\right)$ time, and by Proposition 4.0.6 addressing the element in the value array by this index is $O(1)$ time. By Definition 5.3.7, each original variable may only occur in a single region, so the at most r variables (and corresponding at most r values) in $S'(v')$ may simply be inserted into S. By Proposition 4.0.2, reading each variable and value is $O(1)$ time, and by Proposition 4.0.8, inserting each variable and value is also $O(1)$ time. The operation on line 5 will therefore require $O\left(r.n^2(r+r)\right)$, so $O\left(r^2.n^2\right)$ time.

disallowedAssignments is used to hold the unary projections of the, at most c, constraints in C'' onto v'. By Data Structure 5.3.37 for a merged constraint, each unary projection contains at most t tuples of size r, so disallowedAssignments is of size $O(c.t.r)$, and by Proposition 4.0.5 construction of disallowedAssignments on line 7 is $O(1)$ time. The loop on line 8 will execute at most c times. By Proposition 4.0.11, finding the index of v' in the scope array of c'' requires $O\left(r^2.w\right)$ time. By Proposition 4.0.10, projecting each tuple of $\rho(c'')$ onto this index position is $O(r)$ time, so projecting $\rho(c'')$ onto this position is $O(t.r)$ time. By Proposition 4.0.13, performing the set union between this projection and disallowedAssignments requires $O\left(t.(c.t).r^2\right)$, so $O\left(c.t^2.r^2\right)$ time. The operation on line 9 will therefore require $O\left(r^2.w + t.r + c.t^2.r^2\right)$, so $O\left(r^2.w + c.t^2.r^2\right)$ time.

Insertions are always made into the first available position in an array, as per Definition 4.0.4,

so a stack may be implemented using an array such that *push* is equivalent to insertion and *pop* is equivalent to removing the last populated element. The stack on line 10 is used to build candidate assignments to the variables of the region v', so during execution may contain a sequence of $\sum_{i=1}^{r} k.i$ tuples each of size at most r, so is of size $O\left(k.r^2\right)$ and by Proposition 4.0.5 requires $O\left(1\right)$ time to construct. By Proposition 4.0.2, setting the Boolean flag on line 11 is $O\left(1\right)$ time.

During execution of the loop on line 12, the algorithm is either in a state where it is constructing assignments by pushing elements onto the stack, or it is tearing down assignments from the stack back to a point where it can resume construction. If constructing assignments, it may need at most r iterations of the loop to build assignments to the full set of region variables in v', and if tearing down assignments it may need at most $k.r$ iterations of the loop to reach a state where it can restart construction. In the worst case, all assignments in disallowedAssignments will be constructed before finding a satisfying assignment, so at most $c.t + 1$ assignments. Therefore, the loop on line 12 will execute at most $\left(c.t + 1\right)\left(r + k.r\right)$ times, so $O\left(c.t.k.r\right)$ times.

By Definition 4.0.4 for an array, determining the index of the last element is $O\left(1\right)$ time. By Proposition 4.0.2 reading and then storing the last element from Stack on line 13 is $O\left(r\right)$ time, and by Proposition 4.0.9 deleting it from the array is $O\left(1\right)$ time. By Definition 4.0.4, the current number of elements is stored in a register of an array, so the comparison of the number of elements on line 14 is $O\left(1\right)$ time.

By Proposition 4.0.11 determining whether currentAssignment is a member of disallowedAssignments requires $O\left(c.t.r^2\right)$ time. By Proposition 4.0.2, setting the Boolean flag on line 16 is $O\left(1\right)$ time.

By Definition 4.0.4, the current number of elements in an array is stored in a single register of an array, size of current on line 18 is $O\left(1\right)$. The size of current is then used as the index to address a single element in v', which by Proposition 4.0.6 is $O\left(1\right)$ time. By Proposition 4.0.2 reading an then storing the single element from v' (which is a single variable name from V) is $O\left(1\right)$ time.

By Proposition 4.0.17, finding $D\left(v\right)$ in the non-merged CSP structure (Data Structure 5.3.26) requires $O\left(n^2\right)$ time. The loop on line 19 will then execute at most k times.

By Proposition 4.0.2, making a clone of currentAssignment to be pushed onto the stack is $O\left(r\right)$ time. The mapping from variable to value is implicit as we maintain the variable ordering in currentAssignment, and a mapping from v cannot already exist in currentAssignment at this point, so the union on line 20 may simply be performed by inserting k into the clone of currentAssignment, which by Proposition 4.0.8 is $O\left(1\right)$ time. By Proposition 4.0.8, pushing the clone of currentAssignment onto the stack is $O\left(r\right)$ time. Therefore, the operation on line 20 requires

$O(r + r)$, so $O(r)$ time.

The condition on line 21 requires reading the single register for the found flag and the single register storing the current size of Stack, which by Proposition 4.0.2 are both $O(1)$ time. Evaluating the OR condition then requires integer arithmetic, which by Definition 4.0.1 for a RAM is $O(1)$.

Data Structure 5.3.52 for an assignment consists of a variable array and a corresponding value array. On line 22, the variables for which currentAssignment has been constructed cannot already be in S, so the set union operation can be performed by simply inserting each original variable in the region variable v' into the variable array of S, and each value in the currentAssignment into the value array of S. By Proposition 4.0.8, each insertion is $O(1)$ time, so the operation on line 22 requires $O(r + r)$, so $O(r)$ time.

We can now summarise the time complexity analysis for Algorithm 17 (Unmerge):

$$\begin{aligned} & O\left(n\left(r.n^2 + r^2.n^2 + c\left(r^2.w + c.t^2.r^2\right) + c.t.k.r\left(r + c.t.r^2 + n^2 + k.r\right) + r\right)\right) \\ = \; & O\left(n\left(r^2.n^2 + c.r^2.w + c^2.t^2.r^2 + c^2.t^2.k.r^3 + c.t.k.r.n^2 + c.t.k^2.r^2\right)\right) \\ = \; & O\left(r^2.n^3 + c.r^2.w.n + c^2.t^2.r^2.n + c^2.t^2.k.r^3.n + c.t.k.r.n^3 + c.t.k^2.r^2.n\right) \\ = \; & O\left(c.r^2.w.n + c^2.t^2.k.r^3.n + c.t.k.r.n^3 + c.t.k^2.r^2.n\right) \end{aligned}$$

⌟

Lemma 5.3.55. *Let $P = \langle V, D, C\rangle$ be a CSP in the mixed representation, and $\mathbb{S}$ be a relational structure permitted by P. Let $P' = \langle V', D', C'\rangle$ be a CSP in the positive representation that permits* Mrg($\mathbb{S}$) *constructed by Algorithm 16 ('Restore Removed Constraints'), and let C^- be the set of negatively represented constraints generated by Algorithm 15 ('Convert to Positive'). Given P, P', C^-, and S' a solution to P', Algorithm 17 ('Unmerge Solution') generates a solution to P.*

Proof. Algorithm 15 sets the domain of a region variable, $v' \in V'$ to '$*$' if it only occurs in the scope of negatively represented constraints, and the relations of those constraints do not contain all values that may be assigned to v' in a solution to P'. This infers that there is at least one value assignable to the v' which any partial assignment to $V' \setminus \{v'\}$ can extend to in order to form a solution to P'. These values are generated by Algorithm 17 on lines 7 to 21 by building each possible region value from the domains of the original variables from P in the region and testing whether this value is allowed by the removed constraints. The values from the original domains used to build the region value are then the mappings to the original variables in the solution to P and are added on line 22.

For the variables in S'' whose domains are not of type '$*$', each value in S'' maps directly to a value that is itself a set of mappings from original variables in P to values. □

Solve Mixed CSP

Any solution found to the positive instance we have constructed will have the special star value assigned to those variables with a domain type of '*'. We can convert these to actual values for which the solution is still valid by generating candidate values for the region variable from the domains of the original instance variables that were merged to form the region. We can then test these values against the negatively represented constraints that were removed to check if they are valid (i.e. not disallowed by the removed constraints). We will only have to generate and test at most the number of disallowed assignments plus one before we find a valid value.

The values assigned to the merged variables can then be directly transformed into assignments to original variables by unmerging them such that the full assignment is a solution to the original instance.

Algorithm 18: Solve Mixed CSP

input : A CSP $\langle V, D, C\rangle$ in the Mixed representation
input : $\mathbb{S}$ - a relational structure permitted by $\langle V, D, C\rangle$
output: S - a solution to CSP $\langle V, D, C\rangle$

```
1  begin
2      ⟨V, D, C⟩ ← Replace Equivalent Negative Constraints (⟨V, D, C⟩)
3      Partition ← Create Approximate Partition (⟨V, D, C⟩)
4      ⟨V′, D′, C′⟩ ← Merge (⟨V, D, C⟩, 𝕊)
5      MergedPartition ←
6                  Create Improved Merged Partition (⟨V, D, C⟩, ⟨V′, D′, C′⟩, Partition)
7      ⟨V″, D″, C″⟩, Removed ← Convert to Positive (⟨V′, D′, C′⟩)
8      ⟨V″, D″, C″⟩ ←
9                  Restore Removed Constraints (⟨V″, D″, C″⟩, Removed, MergedPartition)
10     S″ ← SOLVE (⟨V″, D″, C″⟩)
11     S ← Unmerge (⟨V, D, C⟩, ⟨V″, D″, C″⟩, Removed, S″)
12     return S
```

Proposition 5.3.56. *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be any recursively enumerable class of relational structures with bounded interaction width, and let $\mathcal{C}$ be the class of CSPs which permit a structure in $\mathcal{H}$. The class* Mixed($\mathcal{C}$) *is tractable if* $\mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H}))) < \infty$.

Proof. Let $\mathcal{C}'$ be the class of CSPs which permit a structure in $\mathrm{Mrg}(\mathcal{H})$. By Definition 5.3.8, the class $\mathrm{Mrg}(\mathcal{H})$ has bounded arity as the class $\mathcal{H}$ has bounded interaction width. By Grohe's result, Theorem 3.4.9, the class Positive($\mathcal{C}'$) is tractable if $\mathrm{tw}(\mathrm{Core}(\mathrm{Mrg}(\mathcal{H}))) < \infty$.

Let P be any instance in Mixed($\mathcal{C}$), and $\mathbb{S}$ be a structure in $\mathcal{H}$ permitted by P. We will show that P can be reduced to a derived instance P' in Positive($\mathcal{C}'$) in polynomial time with respect to the size of P, and that a solution to P can then be constructed from any solution to P' in polynomial time with respect to the size of P by executing Algorithm 18 ('Solve Mixed CSP').

The size of $\mathbb{S} = \langle U, R_1, \ldots, R_m \rangle$ is always smaller than the size of $P = \langle V, D, C \rangle$ as by Definition 3.4.1: $U = V$, the union of the R_i relations can contain no more scopes than there are constraints in C, and there can be no more non-empty R_i's than there are constraints in C. As such, any algorithm which is polynomial with respect to the size of S is also polynomial with respect to the size of P.

By Lemma 5.3.33, Algorithm 11 ('Replace Equivalent Negative Constraints') does not change the solutions of P, and does not change the relational structures permitted by P. Constraints in Mixed($\mathcal{C}$) may be represented using either positive or negative relations, and P still permits $\mathbb{S}$, so P is still in Mixed($\mathcal{C}$). By Algorithm Analysis 5.3.32, Algorithm 11 runs in polynomial time with respect to the size of P.

Algorithm 12 ('Create Approximate Partition') does not modify P and by Algorithm Analysis 5.3.35 runs in polynomial time with respect to the size of P.

Algorithm 13 ('Merge') constructs the derived CSP, $\text{Mrg}(P) = \langle V', D', C' \rangle$. By Lemma 5.3.40, $\text{Mrg}(P)$ is in Mixed($\mathcal{C}'$), and by Lemma 5.3.41, any solution to $\text{Mrg}(P)$ can be transformed to a solution to P, and if there is no solution to $\text{Mrg}(P)$, then there is no solution to P. By Algorithm Analysis 5.3.39, Algorithm 13 runs in polynomial time with respect to the size of P.

Algorithm 14 ('Create Improved Merged Partition') does not modify $\text{Mrg}(P)$ and by Algorithm Analysis 5.3.43 runs in polynomial time with respect to the size of P.

Algorithm 15 ('Convert to Positive') constructs the derived positively represented instance, $\text{Positive}(\text{Mrg}(P)) = \langle V'', D'', C'' \rangle$, and a set of negatively represented constraints, C^-. By Lemma 5.3.48, the set of partial assignments to $V'' \setminus \{v \mid D(v) \text{ is of type } `*'\}$ is the same in $\text{Mrg}(P)$ and $\text{Positive}(\text{Mrg}(P))$, and each partial assignment extends to a solution in both. By Lemma 5.3.49, the CSP $\langle V'', D'', C'' \cup C^- \rangle$ permits the relational structure $\text{Mrg}(\mathbb{S})$. By Algorithm Analysis 5.3.47, Algorithm 15 runs in polynomial time with respect to the size of P.

Algorithm 16 ('Restore Removed Constraints') constructs P' from $\text{Positive}(\text{Mrg}(P))$ and C^-. By Lemma 5.3.51, P' is in Positive($\mathcal{C}'$), and the set of partial assignments on the variables $V'' \setminus \{v \mid D(v) \text{ is of type } `*'\}$ is the same in $\text{Mrg}(P)$ and P', and each partial assignment extends to a solution in both. By Algorithm Analysis 5.3.50, Algorithm 16 runs in polynomial time with respect to the size of P.

P' is in Positive($\mathcal{C}'$) and may be solved using any appropriate algorithm as per Grohe's result 3.4.9. If there is no solution to P', then there is no solution to P.

By Lemma 5.3.55, Algorithm 17 ('Unmerge Solution') transforms a solution to P' into a solution to P. By Algorithm Analysis 5.3.54, Algorithm 17 runs in polynomial time with respect to the size of P. □

We may now restate from page 68, and provide a proof for, the main theorem of this chapter.

Theorem 5.3.9. *Assuming that W[1] is not FPT. Let $\mathcal{H}$ be any recursively enumerable class of relational structures with bounded interaction width, and let $\mathcal{C}$ be the class of CSPs which permit a structure in $\mathcal{H}$. The class* Mixed($\mathcal{C}$) *is tractable if and only if* $\text{tw}(\text{Core}(\text{Mrg}(\mathcal{H}))) < \infty$.

Proof. By Proposition 5.3.11, Mixed($\mathcal{C}$) is *intractable* if $\text{tw}(\text{Core}(\text{Mrg}(\mathcal{H}))) = \infty$. By Proposition 5.3.56, Mixed($\mathcal{C}$) is *tractable* if $\text{tw}(\text{Core}(\text{Mrg}(\mathcal{H}))) < \infty$. Therefore, Mixed($\mathcal{C}$) is tractable if and only if $\text{tw}(\text{Core}(\text{Mrg}(\mathcal{H}))) < \infty$. □

5.4 Application to Hypergraphs

A hypergraph is equivalent to a relational structure whose relations each contain only a single tuple, that is, each hyperedge forms the sole tuple in its own relation. The interaction width of a hypergraph is therfore the interaction width of this equivalent relational structure. Recall Example 5.3.1 which demonstrates that there are less tractable classes defined for hypergraphs than relational structures. This simplified view of interaction width allows us to generate novel structurally tractable classes for the mixed representation by adding the additional requirement of bounded interaction width to current structurally tractable classes for the positive representation. These classes are simple applications of Theorem 5.3.9.

If we were to construct the dual hypergraph H', of a hypergraph H, the multiset of edges in H' may contain duplicates as several edges in H may share the same vertex.

Definition 5.4.1. *The **dual hypergraph** [Dec03] of a hypergraph $H = \langle V, E \rangle$, is the hypergraph $H' = \langle E, Z \rangle$ such that, for each v in V, there exists an edge in the multiset Z which is itself the set of edges in E that contain v. That is $\forall v \in V, \{e \in E \mid v \in e\} \in Z$.*

It follows from the definition that an interaction region corresponds to each set of edges in the dual which are over the same set of dual vertices. So, the interaction width is the maximum number of dual edges which any of the dual vertices is contained in[2]. This does not help with identifying regions as regions are not always merged to a single dual vertex. However, constructing the dual hypergraph of H' (that is, the dual of the dual), results in a hypergraph which is the equivalent of the original hypergraph H, but with merged vertices forming a single 'dual-dual' vertex.

Definition 5.4.2. *For a hypergraph, $\langle V, E \rangle$, we define the **restriction** to a set of vertices, $V' \subseteq V$, to be the induced hypergraph $\langle V', E' \rangle$ where $E' = \{e \cap V' \mid e \in E\}$.*

Theorem 5.4.3. *Let $H = \langle V, E \rangle$ be a hypergraph and let Ξ be an acyclic guarded decomposition of H of width k. Let $V' \subseteq V$ and H' be the restriction of H to V'. There exists an acyclic guarded decomposition, Ξ', of width at most k for H'*

We are also able to find the acyclic guarded decomposition for H' from the acyclic guarded decomposition for H.

Proof. We are required to show that for any $V' \subseteq V$ we can construct a guarded decomposition, Ξ', which is acyclic and which has width at most k.

Construct Ξ' such that for every guarded block $\langle \lambda, \chi \rangle \in \Xi$ there is a guarded block $\langle \lambda', \chi' \rangle \in \Xi'$ such that $\lambda' = \{e \cap V' \mid e \in \lambda\}$ and $\chi' = \chi \cap V'$.

For Ξ' to be a guarded decomposition of H', it needs to be a complete guarded cover of H'. Ξ is a complete guarded cover of H, so for every edge $e \in E$, there exists $\langle \lambda, \chi \rangle \in \Xi$ with $e \in \lambda$ and $e \subseteq \chi$. By our construction, we may remove vertices from a hyperedge e to give e', and by doing so we also remove the same vertices from the edges in λ and from χ so that $e' \in \lambda'$ and $e' \subseteq \chi'$. So, Ξ' is a complete guarded cover and therefore is a guarded decomposition of H'.

As Ξ is acyclic, its blocks form an acyclic set of hyperedges, A, over V. If we let A' be the set of hyperedges formed by removing the vertices not in V' from A, then it is clear that if performing

[2]The interaction width is similar to the maximum valency of any dual vertex, except that interaction width counts unary dual hyperedges as well as outward connections from the dual vertices.

Graham's Algorithm (Algorithm 1) on A results in an empty set (indicating that A is acyclic), then performing Graham's Algorithm on A' will also result in an empty set. As our construction is such that the blocks of Ξ' form A', Ξ' is acyclic.

Finally, we show Ξ' has width at most k. Ξ has a width of k, so its largest guard has k edges in it. Our construction does not add edges, (although edges may be removed or merged if all of the vertices in the edge are removed,) so by construction Ξ' must have width at most k. □

In particular, this theorem allows us to restrict a hypergraph to a set of vertices such that there is exactly one vertex in each region.

Proposition 5.4.4. *The restriction of a hypergraph to a subset of its vertices does not increase interaction width.*

Proof. From Definition 5.3.7, the interaction width is the maximum number of regions associated with a hyperedge. Removing vertices from all hyperedges in which they belong cannot increase the number of regions associated with a hyperedge. □

As such, we find that we can restrict a hypergraph without modifying its structural width parameters, that is, both decomposition width and interaction width.

For the mixed representation, we may apply the transformation used by the proof of tractability in Theorem 5.3.9 to convert such instances to the positive representation (without changing the decomposition or interaction widths).

We can see that for any tractably identifiable structural decomposition, such as bounded width hypertrees [GLS99], we generate a new tractable class with respect to the mixed representation.

Corollary 5.4.5. *Let $\mathcal{D}$ be a tractably identifiable structural decomposition and $\mathcal{H}$ the family of hypergraphs of width k with respect to $\mathcal{D}$ and interaction width i. The class of CSPs whose structure is in $\mathcal{H}$ and represented with respect to the mixed representation is tractable.*

Corollary 5.4.5 defines structurally tractable classes for the mixed representation with respect to bounded interaction width. Theorem 5.3.9 states that any tractable class defined in these terms must have bounded treewidth of the cores of the merged structures. As such, for the mixed representation, any structural decomposition reduces to treewidth under bounded interaction width. It appears that the interactions between constraints may well be of some importance for sensible representations.

5.5 Tractable Classes of SAT

k-SAT is the subclass of SAT in which each clause contains at most k literals.

2-SAT is NL-complete [Pap94]. 3-SAT is known to be NP-compete [Kar72] and k-SAT can be reduced to 3-SAT, so k-SAT, $k \geq 3$ is in general NP-complete.

Example 5.5.1. A *Horn clause* is a clause which may contain no more than one positive literal.

HORN-SAT is the subclass of k-SAT in which all clauses are Horn clauses. HORN-SAT is interesting (and useful) as it is P-complete [Pap94]. Jeavons and Cooper show that HORN-SAT is equivalent to Max-Closed [JC95]. □

Each clause in a SAT instance only disallows a single assignment. There can be no polynomial-time conversion from SAT clauses to the positive representation as this would lead to a possible exponential blow-up in the size of an instance unless the arity is bound. That is, unless only a finite subset of the language is considered.

Szeider [Sze03] (Corollary 1) has developed a structural tractability result for SAT which is based on incidence width (the treewidth of the incidence graph). He has shown that any class of SAT instances with bounded incidence width is fixed parameter tractable.

We can show that, even just for SAT, this class is incomparable with bounded interaction width, so there are two distinct structural tractability results for SAT. However, ours has a natural extension to domains of larger size, so we hope may be applicable to other practical problems.

In the examples that follow, we consider families of hypergraphs and simplify the notion of an incidence graph by assuming a single constraint on each hyperedge so that we can define an incidence graph of a hypergraph rather than a CSP. This is a reasonable simplification as it does not increase the treewidth of the incidence graph and CSPs exist for each hypergraph under this assumption.

Consider the hypergraph $H_n = \langle V_n, E_n \rangle$, shown in Figure 5.6 (left), with vertices:

$$V_n = \{v_1, v_2, \ldots v_n, w_1, w_2, \ldots, w_n\}$$

and edges:

$$E_n = \{e = \{v_1, v_2, \ldots, v_n\}, f_1 = \{v_1, w_1\}, f_2 = \{v_2, w_2\}, \ldots, f_n = \{v_n, w_n\}\} \quad .$$

The interaction width of H_n is n, since the edge $e = \{v_1, v_2, \ldots, v_n\}$ has a separate interaction with each of the n edges f_1, f_2, …, f_n.

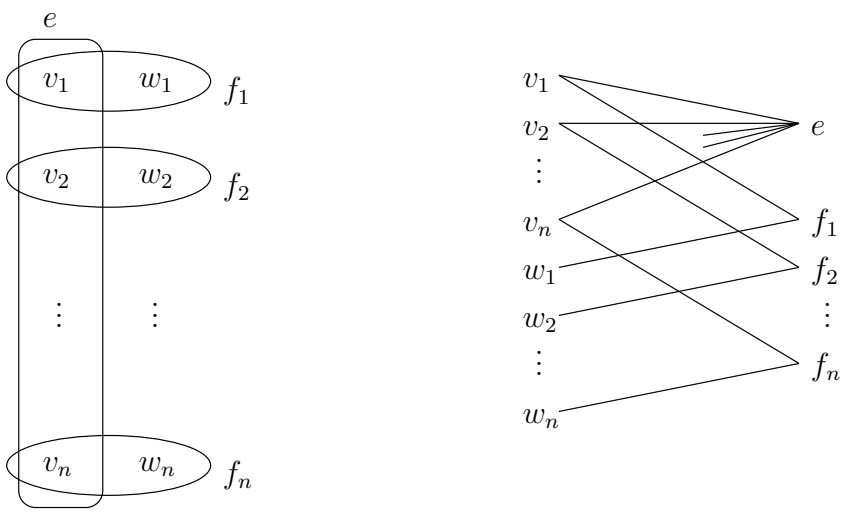


Figure 5.6: (left) The hypergraph H_n. (right) The incidence graph ${H_n}^*$ of H_n.

Consider instead the incidence graph ${H_n}^*$ of H_n, given in Figure 5.6 (right). It is straightforward to show that the treewidth of ${H_n}^*$ is 1. An ordering of the vertices of ${H_n}^*$ that witnesses this fact is:

$$[e, v_1, f_1, w_1, v_2, f_2, w_2, \ldots, v_n, f_n, w_n] \quad .$$

Let $\mathcal{H}$ be the family of hypergraphs $\{H_i \mid i \in \mathbb{N}\}$. The infinite class $\mathcal{H}$ shows that Szeider's result (bounded treewidth of the incidence graph) *dominates* bounded interaction width (in the sense given by Gottlob et al. [GLS00]) when considering only classes of SAT instances.

To show domination in the other direction we consider a slightly different hypergraph, $J_n = \langle V_n, F_n \rangle$, shown in Figure 5.7 (left), with vertices the same as H_n, that is:

$$V_n = \{v_1, v_2, \ldots v_n, w_1, w_2, \ldots, w_n\}$$

and edges:

$$F_n = \{e_i = \{v_1, v_2, \ldots, v_n, w_i\} \mid i = 1, 2, \ldots, n\} \quad .$$

The interaction width of J_n is 2, since for each i, the relational tuple e_i has only two interaction regions, $\{v_1, v_2, \ldots, v_n\}$ and $\{w_i\}$.

Consider instead the incidence graph ${J_n}^*$ of J_n, given in Figure 5.7 (right). The treewidth of ${J_n}^*$ is at least n, since the vertices v_1, v_2, ..., v_n, e_1, e_2, ..., e_n of ${J_n}^*$ form a complete bipartite subgraph on $\{v_1, v_2, \ldots, v_n\}$ and $\{e_1, e_2, \ldots, e_n\}$. An ordering of the vertices of ${J_n}^*$ that has

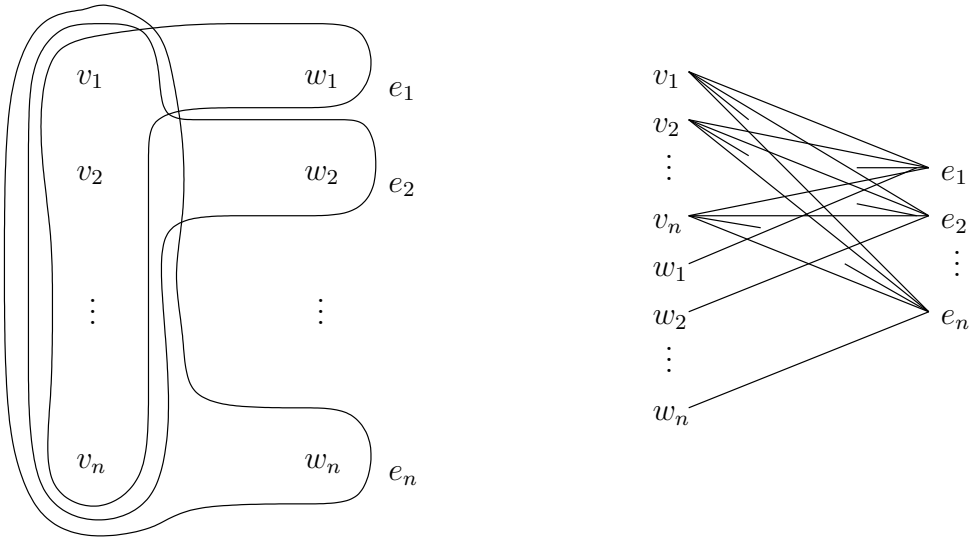


Figure 5.7: (left) The hypergraph J_n. (right) The incidence graph ${J_n}^*$ of J_n.

treewidth n is:

$$[v_1, v_2, \ldots, v_n, e_1, e_2, \ldots, e_n, w_1, w_2, \ldots, w_n] \quad .$$

Therefore, the treewidth of ${J_n}^*$ is exactly n.

Let $\mathcal{J}$ be the family of hypergraphs $\{J_i \mid i \in \mathbb{N}\}$. The infinite class $\mathcal{J}$ shows that bounded interaction width dominates bounded treewidth of the incidence graph when considering only classes of SAT instances.

In this section we have shown that bounded interaction width is incomparable to bounded treewidth of the incidence graph for SAT instances. Also, the two families $\mathcal{H}$ and $\mathcal{J}$ show that non-trivial examples of each tractable class exist. It is interesting to note that in both of these cases the (hyper)graphs are acyclic. For CSPs represented with respect to the positive representation we could state that both families are subclasses of the tractable acyclic class. However, we cannot apply the acyclic solution algorithm to SAT instances.

5.6 Place in the Succinctness Hierarchy

The main result of this section will be to show that the mixed representation is *strictly* more structurally tractable than the GDNF representation. We also show that the positive representation is strictly more structurally tractable than the mixed representation.

We do this by demonstrating the succinctness relation between these three representations and

appealing to Corollary 5.2.7. We will then exhibit appropriate classes of relational structures to distinguish their structural tractability.

Proposition 5.6.1. *The GDNF representation is as succinct as the mixed representation which is as succinct as the positive representation.*

Proof. By Definition 5.2.6, since any instance in the positive representation can be given in the mixed representation with only a linear size increase (by remaining in the positive representation and adding the necessary representation flag), it follows immediately that the mixed is as succinct as the positive.

Given a positively represented constraint we can use the straightforward construction of Chen and Grohe [CG06] which generates a product of unary sets for each allowed assignment. This may be done in linear time, and the resulting GDNF representation of the constraint is (approximately) the same size.

Given a negatively represented constraint we can use a result by Katsirelos and Walsh [KW07]. They show that any negatively represented constraint may be converted to an equivalent GDNF representation of the constraint, with a polynomial number of set products in polynomial time using a simple algorithm which descends a *decision tree* with a polynomial number of leaves.

A mixed representation of a constraint can either be a positive representation, or a negative representation, so we are done. □

Theorem 5.6.2. *The positive representation is strictly more structurally tractable than the mixed representation which is strictly more structurally tractable than the GDNF representation.*

Proof. By Proposition 5.6.1 and Corollary 5.2.7 we have that the positive representation is more structurally tractable than the mixed representation which is more structurally tractable than the GDNF representation.

We show strictness by exhibiting appropriate classes of relational structures.

We saw in Example 5.0.20 the class of acyclic structures is not tractable for the negative representation. This implies that it is not tractable for the mixed representation. This serves to show that the positive representation is strictly more structurally tractable than the mixed representation.

Consider the relational structure $\mathbb{H}_n = \langle V_n, E_1, E_2, \ldots, E_n \rangle$ where the universe is:

$$V_n = \{v_1, v_2, \ldots v_n, w_1^1, w_2^1, w_2^2, \ldots, w_n^1, \ldots, w_n^n\}$$

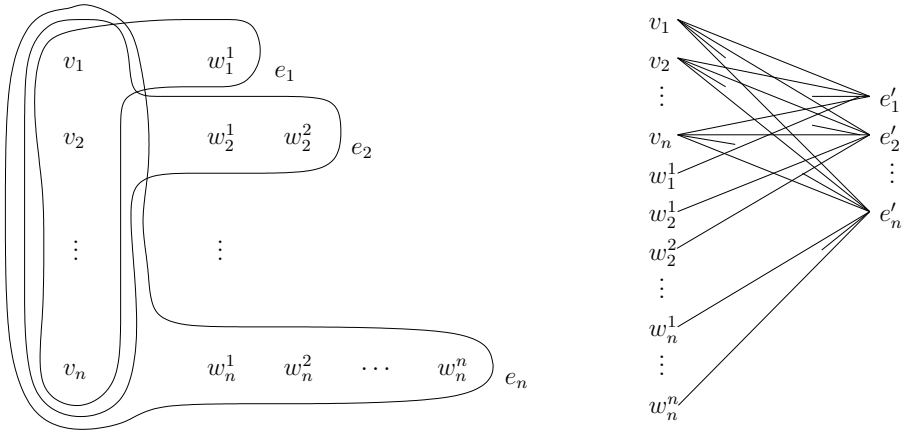


Figure 5.8: (left) $H(\mathbb{H}_n)$ (right) IncG($\mathbb{H}_n$)

and the relations are:

$$E_1 = \{e_1 = \langle v_1, v_2, \ldots, v_n, w_1^1 \rangle\}$$
$$E_2 = \{e_2 = \langle v_1, v_2, \ldots, v_n, w_2^1, w_2^2 \rangle\}$$
$$\vdots$$
$$E_n = \{e_n = \langle v_1, v_2, \ldots, v_n, w_n^1, \ldots, w_n^n \rangle\}$$

We depict the structural hypergraph, $H(\mathbb{H}_n)$, of $\mathbb{H}_n$ in Figure 5.8 (left).

The interaction width of $\mathbb{H}_n$ is 2, since for each i, the relational tuple e_i has only two interaction regions, $\{v_1, v_2, \ldots, v_n\}$ and $\{w_i^1, \ldots, w_i^i\}$.

What is more, it is straightforward to show that the merged structure of $\mathbb{H}_n$ is tree-structured, so the treewidth of the merged structure of $\mathbb{H}_n$ is 1.

By consideration of the arities of the relations we can see that each of these relational structures is a core. So, to determine the tractability of these structures for the GDNF representation, we have only to consider their incidence width.

The incidence graph, IncG($\mathbb{H}_n$), of $\mathbb{H}_n$ is the bipartite graph $\langle V_n', E_n' \rangle$ such that $V_n' = V_n \cup L(\mathbb{H}_n)$, where

$$L(\mathbb{H}_n) = \{e_1' = \langle 1, e_1 \rangle, e_2' = \langle 2, e_2 \rangle, \ldots, e_n' = \langle n, e_n \rangle\}, \text{ and}$$

$$
\begin{aligned}
E'_n = \{&\langle v_1, e'_1\rangle, \ldots, \langle v_n, e'_1\rangle, \langle w_1^1, e'_1\rangle\} \\
&\cup \{\langle v_1, e'_2\rangle, \ldots, \langle v_n, e'_2\rangle, \langle w_2^1, e'_2\rangle, \langle w_2^2, e'_2\rangle\} \\
&\vdots \\
&\cup \{\langle v_1, e'_n\rangle, \ldots, \langle v_n, e'_n\rangle, \langle w_n^1, e'_n\rangle, \ldots, \langle w_n^n, e'_n\rangle\} \ .
\end{aligned}
$$

We depict the incidence graph, IncG($\mathbb{H}_n$), of $\mathbb{H}_n$ in Figure 5.8 (right). Since the vertices $\{v_1, v_2, \ldots, v_n\}$ and $\{e'_1, e'_2, \ldots, e'_n\}$ of IncG($\mathbb{H}_n$) form a complete bipartite subgraph, the treewidth of IncG($\mathbb{H}_n$) is at least n.

Let $\mathcal{H}$ be the class of relational structures $\{\mathbb{H}_i \mid i = 1, 2, \ldots\}$. The infinite class $\mathcal{H}$ has bounded interaction width together with bounded treewidth of the merged structures, but has unbounded treewidth of the incidence graphs. As such, the structural class of CSPs defined by $\mathcal{H}$ is not tractable with respect to the GDNF representation. However, it is tractable with respect to the mixed representation. □

In this section we have described a class of structures which is not tractable for the GDNF representation, yet is tractable for the mixed representation. This class shows that the mixed representation provides novel tractable structural classes. In particular, since the mixed representation naturally extends SAT, this provides a useful result, extending known structural tractability results for SAT, in particular Szeider's result shown earlier [Sze03].

5.7 Not a Dichotomy

It might be hoped that bounded interaction width would provide a dichotomy for tractable structural classes in the mixed representation. In this section we show that, unfortunately, this is not the case.

Consider the relational structure $\mathbb{J}_n = \langle V_n, F_n, G_1, \ldots, G_n\rangle$ where

$$
V_n = \{v_1, v_2, \ldots v_n, w_1^1, w_2^1, w_2^2, \ldots, w_n^1, \ldots, w_n^n\}
$$

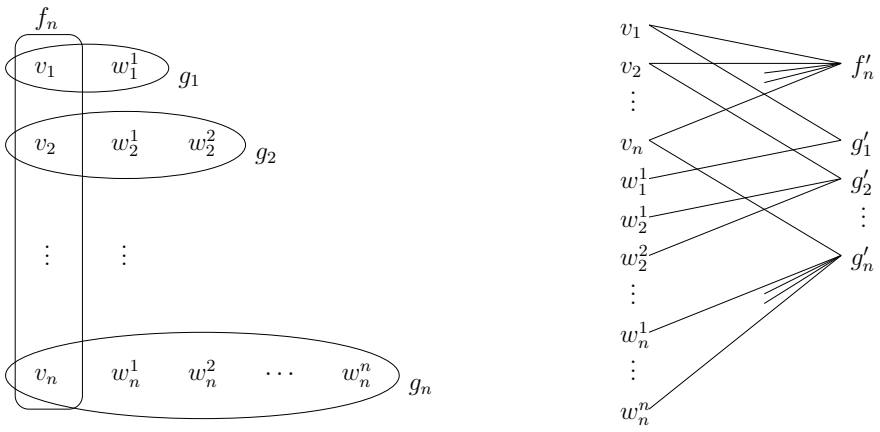


Figure 5.9: (left) $H(\mathbb{J}_n)$ (right) IncG($\mathbb{J}_n$).

and the relations are:

$$
\begin{aligned}
F_n &= \{f_n = \langle v_1, v_2, \ldots, v_n \rangle\} \\
G_1 &= \{g_1 = \langle v_1, w_1^1 \rangle\} \\
G_2 &= \{g_2 = \langle v_2, w_2^1, w_2^2 \rangle\} \\
&\vdots \\
G_n &= \{g_n = \langle v_n, w_n^1, \ldots, w_n^n \rangle\}
\end{aligned}
$$

We depict the structural hypergraph, $H(\mathbb{J}_n)$, of $\mathbb{J}_n$ in Figure 5.9 (left).

The interaction width of $\mathbb{J}_n$ is n, since the relational tuple f_n has a separate interaction with each of the n hyperedges $g_1, g_2, \ldots, g_n$.

Consideration of the arities of the relations is enough to show that these relational structures are cores, so to determine if this class of structures is tractable for the GDNF representation we need only consider their incidence width.

Consider the incidence graph, IncG($\mathbb{J}_n$), of $\mathbb{J}_n$ which is the bipartite graph $\langle W, F'_n \rangle$ such that $W = V_n \cup L(\mathbb{J}_n)$, where

$$L(\mathbb{J}_n) = \{f'_n = \langle 1, f_n \rangle, g'_1 = \langle 2, g_1 \rangle, g'_2 = \langle 3, g_2 \rangle, \ldots, g'_n = \langle n+1, g_n \rangle\}, \text{ and}$$

$$
\begin{aligned}
F'_n = \{&\langle v_1, f'_n\rangle, \ldots, \langle v_n, f'_n\rangle\} \\
&\cup \{\langle v_1, g'_1\rangle, \langle w_1^1, g'_1\rangle\} \\
&\cup \{\langle v_2, g'_2\rangle, \langle w_2^1, g'_2\rangle, \langle w_2^2, g'_2\rangle\} \\
&\vdots \\
&\cup \{\langle v_n, g'_n\rangle, \langle w_n^1, g'_n\rangle \ldots, \langle w_n^n, g'_n\rangle\} \ .
\end{aligned}
$$

We depict the incidence graph, IncG($\mathbb{J}_n$), of $\mathbb{J}_n$ in Figure 5.9 (right). It is straightforward to show that the treewidth of IncG($\mathbb{J}_n$) is 1. An ordering of the vertices of IncG($\mathbb{J}_n$) that witnesses this fact is

$$[f'_n, v_1, g'_1, w_1^1, v_2, g'_2, w_2^1, w_2^2, \ldots, v_n, g'_n, w_n^1, \ldots, w_n^n] \ .$$

Let $\mathcal{J}$ be the class of relational structures $\{\mathbb{J}_i \mid i = 1, 2, \ldots\}$. The infinite class $\mathcal{J}$ has unbounded interaction width, and hence unbounded treewidth of the cores of the merged structures, but has bounded treewidth of the incidence graphs. Hence, we cannot say using our results for the mixed representation (by interaction width) that this class of relational structures defines a tractable structural class with respect to the mixed representation. However, it is tractable for the GDNF representation and thus, by reduction, is tractable for the mixed representation. This demonstrates that interaction width alone is not sufficient to provide a dichotomy for the tractable structural classes for the mixed representation.

Chapter 6

Conclusions and Further Work

The work in this thesis was motivated by the apparent disconnect between the constraint representations used for theoretical tractability results, and those used by constraint practitioners. We have seen that acyclicity is fundamental to structural tractability results, but that it is not, in general, a sufficient condition for tractability for more succinct representations.

In order to improve our understanding and ability to address this issue, we have considered the conditions necessary to apply existing structural tractability results to more succinct representations. In particular, we have considered the mixed representation as it allows us to succinctly specify SAT instances.

We showed that existing structural tractability results are applicable in the case where instances in a more succinct representation can be converted in polynomial time into equivalent instances in the positive representation, in such a way as to preserve existing structural width properties. We determined that the difficulty of performing such a conversion was related to the number and nature of the distinct interactions a constraint is involved in, and so developed the notion of *interaction width* as a measurement of this property. We were then able to develop the algorithms necessary to perform this polynomial time conversion for classes of bounded interaction width by merging the variables involved in each interaction. As we had defined interaction width for relational structures, we could then appeal to a result by Grohe [Gro07] to give a dichotomy for the mixed representation: that the class of CSPs permitting a relational structure with bounded interaction width are tractable if and only if the treewidth of the core of their merged structure is bounded.

Chen and Grohe [CG06] had previously shown that there is a simple width based characterisation of structurally tractable classes for the GDNF representation which provides a complete

dichotomy: classes permitting relational structures whose cores do not have bounded incidence width are W[1] hard. Szeider [Sze03] has also developed a structural tractability result for SAT: that any class of SAT instances with bounded incidence width is fixed parameter tractable. We have shown that the tractable classes we have defined for the mixed representation, and therefore SAT, are novel by demonstrating that they are incomparable to both of these existing results.

Observe that the examples of Sections 5.6 and 5.7 are acyclic. We know that acyclicity is not enough to guarantee tractability for succinct representations. In these cases it is the additional restriction imposed by, respectively, *interaction width together with the treewidth of the core of the merged structure* and *incidence width* that gives tractability for the appropriate succinct representations.

The results presented in this thesis provide a significant contribution to our understanding of the conditions for tractability under succinct representations. Understanding why tractability results may be different for a class of CSPs under different representations moves us closer to being able to decouple the reasons for tractability from the effects of representation. We have shown, at least for the representations considered here, that the number and manner of interactions between constraints is an important property that can provide a necessary condition for tractability.

Further Work

For the GDNF representation, the tractability characterisation given by Chen and Grohe [CG06] is a complete dichotomy. However, we do not yet have a dichotomy for structural classes of mixed representations of CSPs, except in the special case of bounded interaction width.

We have shown that the hierarchy of structural tractability is strict for positive, mixed and GDNF representations, and have used this characterisation to construct novel structurally tractable classes for SAT. Unfortunately, we cannot yet show whether or not the negative representation is more structurally tractable than the mixed representation, which would complete the hierarchy.

It would also be interesting to evaluate the effects of applying a minimum description length principle to the mixed representation. That is, only allowing constraints to be represented using the smaller of either the positive or negative representations. However, this may be considered an unusual restriction as it is unlikely to be enforced in practice.

In this thesis we have been concerned with a single succinct representation, mixed, and previous work has also been been restricted to single succinct representations considered in isolation. This is because we rely on knowledge of the succinct representation in order to perform the conversion

to the positive representation. As such, these results have shown that structural restrictions can lead to the applicability of existing tractability results for single succinct representations. As the mixed representation is the combination of the positive and negative representations, it may be that there are larger combinations of different representations for which tractable classes may be defined under some structural restriction. It may be considered that the results follow from the combination of a structural and a representational restriction, and it would be interesting further work to investigate whether there are properties of more general representational restrictions which allow similar results. For instance, the solvers commonly used in practice are constraint propagators whose input languages consist of global constraints. The nature of global constraints is such that they do not follow such a consistently defined single representation, and it is an interesting question as to what restrictions to a language of global constraints may enable structural tractability results to be applied.

The algorithm analysis framework described in Chapter 4 was invaluable for maintaining the consistency of the complexity analyses in Chapter 5. The construction of this framework was necessary as no suitable existing catalogue of data structures and operational complexities was found to be available. The expansion of this framework beyond the structures and operations necessary to analyse the algorithms contained in this thesis may result in a useful resource.

Bibliography

[AGG05] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree-width and related hypergraph invariants. In *2005 European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05)*, 2005.

[BBJ07] G.S. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and Logic*. Cambridge University Press, 2007.

[BCDP07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, 2007.

[BCSvB01] Adam Beacham, Xinguang Chen, Jonathan Sillito, and Peter van Beek. Constraint programming lessons learned from crossword puzzles. In *In Proceedings of the 14th Canadian Conference on Artificial Intelligence*, pages 78–87, 2001.

[BFMY83] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, 1983.

[BGKK10] Andrei A. Bulatov, Martin Grohe, Phokion G. Kolaitis, and Andrei Krokhin. 09441 abstracts collection – the constraint satisfaction problem: Complexity and approximability. In Andrei A. Bulatov, Martin Grohe, Phokion G. Kolaitis, and Andrei Krokhin, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, number 09441 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[BR97] C. Bessière and J-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[CG06] Hubie Chen and Martin Grohe. Constraint satisfaction with succinctly specified relations. In Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints*, number 06401 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[CGH09] David A. Cohen, Martin J. Green, and Chris Houghton. Constraint representations and structural tractability. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2009.

[CJG05] David A. Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *IJCAI*, pages 72–77, 2005.

[CJG08] David Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74:721–743, 2008. Earlier, uncorrected, version appears in Proceedings of IJCAI'05, pp. 72–77.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Coh04] David A. Cohen. Tractable decision for a constraint language implies tractable search. *Constraints*, 9(3):219–229, 2004.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CRR72] Stephen A. Cook, Robert, and A. Reckhow. Journal of computer and system sciences 7, 354–375 (1973) time bounded random access machines, 1972.

[CV12] Nadia Creignou and Heribert Vollmer. Parameterized complexity of weighted satisfiability problems. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 341–354. Springer, 2012.

[Dec92] Rina Dechter. Constraint Networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.

[Dec03] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.

[dK92] Johan de Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 780–785. AAAI Press, 1992.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DP89] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[EFW+02] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In *In: CP 02 Principles and Practice of Constraint Programming*, pages 525–540. Springer, 2002.

[End10] Herbert B. Enderton. *Computability Theory: An Introduction to Recursion Theory.* Academic Press, 1st edition, 2010.

[Fer09] Maribel Fernndez. *Models of Computation: An Introduction to Computability Theory.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[FJHM05] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *IJCAI*, pages 109–116, 2005.

[FM01] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 77–92, London, UK, 2001. Springer-Verlag.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, 1978.

[Fre82] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[Gas79] John Gary Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, 1979.

[GB65] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, October 1965.

[GGM+05] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, pages 1–15, 2005.

[GJ79] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA., 1979.

[GJC94] Marc Gyssens, Peter Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[GLS99] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394–399. Morgan Kaufmann, 1999.

[GLS00] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[GM99] Martin Grohe and Julian Mariño. Definability and descriptive complexity on databases of bounded tree-width. *Lecture Notes in Computer Science*, 1540:70–82, 1999.

[GM06] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 289–298, New York, NY, USA, 2006. ACM.

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6), 2009.

[Gra79] M H. Graham. On the universal relation. Technical report, University of Toronto, Toronto, Ontario, Canada, 1979.

[Gro07] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1, 2007.

[GSMT98] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *AIPS*, pages 208–213, 1998.

[GW94] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109, 1994.

[GW02] Ian P. Gent and Toby Walsh. Satisfiability in the year 2000. *J. Autom. Reasoning*, 28(2):99, 2002.

[HCG06] Chris Houghton, David A. Cohen, and Martin J. Green. The effect of constraint representation on structural tractability. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204, pages 726–730, 2006.

[HE79] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proc. of the 6th IJCAI*, pages 356–364, Tokio, Japan, 1979.

[HS02] Warwick Harvey and Peter Stuckey. Improving linear constraint propagation by changing constraint representation, 2002.

[HW84] G.H. Hardy and E.M. Wright. *An introduction to the theory of numbers*. Oxford science publications. Clarendon Press, 1984.

[JC95] P.G. Jeavons and M.C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2):327–339, 1995.

[JCG96] P.G. Jeavons, D.A. Cohen, and M. Gyssens. A test for tractability. In *Proceedings 2nd International Conference on Constraint Programming—CP'96 (Boston, August 1996)*, volume 1118 of *Lecture Notes in Computer Science*, pages 267–281. Springer-Verlag, 1996.

[JCG97] Peter Jeavons, David Cohen, and Marc Gyssens. Closure properties of constraints. *Journal of the ACM*, 44:527–548, 1997.

[JJNV89] P. Janssen, P. Jegou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint satisfaction problems: achieving pair-wise consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *AAAI*, pages 390–396, 2005.

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[KW07] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 379–393. Springer-Verlag, 2007.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, November 1993.

[RDP90] F. Rossi, V. Dahr, and C. Petrie. On the equivalence of constraint satisfaction problems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI90)*, August 1990. Also MCC Technical Report *ACT-AI-222-89*.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

[Sze03] Stefan Szeider. On fixed-parameter tractable parameterizations of SAT. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May*

5-8, 2003 Selected Revised Papers, volume 2919 of *Lecture Notes in Computer Science*, pages 188–202, 2003.

[TCC+91] William M. Taylor, Peter Cheeseman, Peter Cheeseman, Bob Kanefsky, and Bob Kanefsky. Where the really hard problems are. In *J. Mylopoulos and R. Reiter (Eds.), Proceedings of 12th International Joint Conference on AI (IJCAI-91),Volume 1*, pages 331–337. Morgan Kauffman, 1991.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42), 1936.

[US94] Toms E. Uribe and Mark E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *Proc. of the 1st International Conference on Constraints in Computaional Logics*, pages 34–49. Springer-Verlag, 1994.

[vHK06] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 6. Elsevier, 2006.

[vN93] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993.

[WGS03] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

[YAAP03] Jun Yuan, Ken Albin, Adnan Aziz, and Carl Pixley. Constraint synthesis for environment modeling in functional verification. In *DAC*, pages 296–299. ACM, 2003.

[ZS94] Hantao Zhang and Mark E. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, Artificia Intelligence Center, SRI International, Menlo, 1994.