

# Transformations of representation in constraint satisfaction



András Z. Salamon  
St Anne's College  
University of Oxford

A thesis submitted for the degree of  
*Doctor of Philosophy*  
Trinity 2013

# Transformations of representation in constraint satisfaction

András Z. Salamon

St Anne's College  
University of Oxford

*A thesis submitted for the degree of  
Doctor of Philosophy*

Trinity 2013

## Abstract

In this thesis I study constraint satisfaction problems or CSPs. These require determining whether values can be assigned to variables so that all constraints are satisfied. An important challenge is to identify tractable CSPs which can be solved efficiently.

CSP instances have usually been grouped together by restricting either the allowed combinations of values, or the way the variables are allowed to interact. Such restrictions sometimes yield tractable CSPs. A weakness of this method is that it cannot explain why all-different constraints form a tractable CSP. In this common type of constraint, all variables must be assigned values that are different from each other. New techniques are therefore needed to explain why such CSPs can be solved efficiently.

My main contribution is an investigation of such hybrid CSPs which cannot be defined with either one of these kinds of restrictions. The main technique I use is a transformation of a CSP instance to the microstructure representation. This represents an instance as a collection of sets, and a solution of the instance corresponds to an independent set in the clause structure.

For the common case where all constraints involve only two variables, I show how the microstructure can be used to define CSPs that are tractable because their clause structures fall within classes of graphs for which an independent set of specified size can be found efficiently. Such tractable hereditary classes are defined by using the technique of excluded induced subgraphs, such as classes of graphs that contain neither odd cycles with five or more vertices, nor their complements. I also develop finer grained techniques, by allowing vertices of the microstructure representation to be assigned colours, and the variables to be ordered. I show that these techniques define a new tractable CSP that forbids an ordered vertex-coloured subgraph in the microstructure representation.

---

## Statement of Originality

Some of the results in Chapter 5 were published in collaboration with my supervisor Peter Jeavons; the chapter itself is my own work and extends the previously published results.

- [141] A. Z. Salamon and P. G. Jeavons. Perfect constraints are tractable. In P. J. Stuckey, editor, *CP 2008: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, 524–528. Springer-Verlag, 2008. [doi:10.1007/978-3-540-85958-1\\_35](https://doi.org/10.1007/978-3-540-85958-1_35).

Chapter 6 includes some results that also appeared in two publications produced in collaboration with Martin Cooper and Peter Jeavons. The exposition of these results in Chapter 6 is my own work.

- [41] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Hybrid tractable CSPs which generalize tree structure. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *ECAI 2008, Proceedings of the 18th European Conference on Artificial Intelligence, July 21–25, Patras, Greece*, Frontiers in Artificial Intelligence and Applications **178**, 530–534. IOS Press, 2008. (Best paper award) [doi:10.3233/978-1-58603-891-5-530](https://doi.org/10.3233/978-1-58603-891-5-530).
- [42] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination. *Artificial Intelligence*, **174**(9–10), 570–584, June 2010. [doi:10.1016/j.artint.2010.03.002](https://doi.org/10.1016/j.artint.2010.03.002).

With these exceptions, this thesis is wholly my own work.

## Acknowledgements

Many people helped while I worked on this thesis. I thank everyone who contributed.

I explicitly thank the following individuals. Martin Green and Karen Petrie provided valuable folklore about constraint satisfaction when I was starting out. Paul Dorbec investigated different types of graph products with me. Stephan Kreutzer corrected some of my early misconceptions about product graphs and provided several useful nudges along the way. Sebastian Ordyniak discussed several graph problems and life in general. Rod Burstall pointed out an important early paper, Don Sannella provided a draft version of his book, and Christian Kissig invited me to speak in Leicester; each inspired me to consider this work in a broader theoretical context. Justyna Petke took time to discuss many of the gory details of satisfiability and constraint solving. Nina Alphey provided perspective and ongoing encouragement. Grant Passmore shared his boundless enthusiasm for mathematics. Dave Cohen provided much useful commentary, words of encouragement, and criticism at several important stages. Georg Gottlob provided key insights on the early stages of my thesis, and was highly supportive in the later stages. Chris Jefferson supplied many insightful comments, as well as reminding me of practical applications. Martin Brain has been a thoughtful correspondent over several years, invited me to speak in Bath, and helped me to discover the connections between the various communities investigating constraint satisfaction. Standa Živný endured my noisy typing for several years, was a voice of reason in discussions about theoretical computer science, and set an example of consistency and time management. Evgenij Thorstensen provided kind encouragement during the writing phase. Martin Cooper has been an incredibly energetic collaborator, and elegantly demonstrated, time and again, how to grow large trees from small acorns. My father Tamás István Salamon (1939–2009) encouraged me to persist with this work even shortly before his death.

Most importantly, I thank Peter Jeavons, who convinced me to work (and keep working) on microstructures, kept pointing out instances where I had brushed aside interesting questions with just a few words, showed by example how to turn good ideas into great papers, and who has always been right. He has been a most supportive supervisor.

Finally, I thank my partner Vashti Galpin, without whom none of this would have happened. She helped to hold things together even when real life threatened to completely derail this thesis. We even wrote some papers together, hopefully there will be more!

---

The quotes preceding each chapter are by philosopher and logician C. S. Peirce, from the 1892 essay *The Critic of Arguments* [127], and by A. B. Kempe [100] from the 1886 essay *A Memoir on the Theory of Mathematical Form*. The section numbers for the Peirce quotes are those used in the Collected Papers. Peirce worked on the foundations of relation algebra, and in this essay he discusses how he changed his views about the expressive power of binary relations. This reassessment occurred after reading Kempe’s paper, which outlines how binary relations can be used to express any concept. While Peirce conceded in his essay that binary relations are expressive enough to capture arbitrary relationships, he also forcefully defended the use of higher arity relations as more natural.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	A high level tour . . . . .	1
	Structural restrictions . . . . .	4
	Language restrictions . . . . .	4
1.2	Document roadmap . . . . .	5
1.3	Contributions . . . . .	6
	Hereditary microstructure vs. propagation . . . . .	6
	Microstructure as product . . . . .	7
	Direct encoding as SAT . . . . .	7
	Perfect microstructure unifies results . . . . .	7
	Explanation of tractability of all-different . . . . .	7
	Microstructure as vertex-coloured structure . . . . .	7
	Broken-triangle new hybrid class . . . . .	8
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Constraint satisfaction problems . . . . .	9
2.2	Foundations . . . . .	10
	2.2.1 Computational complexity . . . . .	11
	2.2.2 Standard definitions . . . . .	12
	2.2.3 Relational structures, graphs, and hypergraphs . . . . .	14
	2.2.4 General foundations . . . . .	18
2.3	Examples of constraint satisfaction problems . . . . .	18
2.4	Classic representations . . . . .	23
	2.4.1 Variable-value representation . . . . .	23
	2.4.2 Partial assignments and solutions . . . . .	24
	2.4.3 Domains . . . . .	26
	2.4.4 First-order logic representation . . . . .	27
	2.4.5 Homomorphism representation . . . . .	28
2.5	Historical perspective . . . . .	30
2.6	Chapter summary . . . . .	32

<b>3</b>	<b>Concepts</b>	<b>33</b>
3.1	Properties of constraints . . . . .	33
3.1.1	Infrastructure and equivalence . . . . .	33
3.1.2	Combining constraints . . . . .	35
3.1.3	More about domains . . . . .	37
3.1.4	Different notions of instance equivalence . . . . .	39
	Nogood equivalence . . . . .	39
	Mutual reducibility . . . . .	40
	Same-solution equivalence . . . . .	40
3.2	Tractable CSPs . . . . .	41
3.2.1	CSPs from restrictions . . . . .	41
3.2.2	Reasons for tractability: structure . . . . .	43
3.2.3	Reasons for tractability: language . . . . .	44
3.2.4	Reasons for tractability: hybrid . . . . .	46
3.3	Transformations between representations . . . . .	47
3.3.1	Variable-value to first-order logic . . . . .	48
3.3.2	Variable-value to homomorphism . . . . .	48
3.3.3	First-order logic to variable-value . . . . .	49
3.3.4	First-order logic to homomorphism . . . . .	49
3.3.5	Homomorphism to variable-value . . . . .	49
3.3.6	Homomorphism to first-order logic . . . . .	50
3.4	Chapter summary . . . . .	50
<b>4</b>	<b>Microstructure representation</b>	<b>51</b>
4.1	Building a representation from literals . . . . .	52
4.1.1	Microstructure . . . . .	52
4.1.2	Clause structure . . . . .	53
4.1.3	Microstructure representation . . . . .	54
4.1.4	Graphical notation . . . . .	56
4.2	Solutions in the microstructure representation . . . . .	58
4.3	Examples . . . . .	61
4.4	Properties of the microstructure representation . . . . .	64
4.4.1	Normalisation . . . . .	64
4.4.2	Microstructure representation of combined constraints . . . . .	65
4.4.3	Clause structure as direct encoding . . . . .	67
4.4.4	Product form . . . . .	69
4.5	Context and summary . . . . .	73
<b>5</b>	<b>Hereditary classes of binary microstructures</b>	<b>74</b>
5.1	Transforming to the binary microstructure . . . . .	75
5.1.1	Global constraints . . . . .	76

5.1.2	Exact 2-section . . . . .	77
5.1.3	Hidden variable transformation . . . . .	83
5.1.4	Dual and hidden transformation . . . . .	84
5.1.5	Reflections on transforming to binary . . . . .	84
5.2	Hereditary classes . . . . .	85
5.2.1	Forbidden substructures . . . . .	85
5.2.2	IS-easy and IS-hard classes of graphs . . . . .	87
5.2.3	Hereditary classes, forbidden substructures, and domain reduction . . . . .	88
5.2.4	Small forbidden structures . . . . .	90
5.3	Perfect microstructure . . . . .	92
5.3.1	Classes related to trees . . . . .	94
5.3.2	Classes related to chordal graphs . . . . .	96
5.3.3	Classes related to gridline graphs . . . . .	97
5.3.4	Summary of relationships . . . . .	99
5.4	Summary and contribution . . . . .	100
<b>6</b>	<b>Variable-coloured binary microstructures</b>	<b>102</b>
6.1	Vertex-coloured structures and logic . . . . .	103
6.2	Transformations . . . . .	104
6.3	Rainbow independent sets . . . . .	107
6.4	Variable ordering . . . . .	109
6.5	Reductions . . . . .	110
6.6	Forbidden vc-graphs . . . . .	111
6.7	Broken-triangle property . . . . .	114
6.8	Summary and contribution . . . . .	120
<b>7</b>	<b>Complete representation</b>	<b>121</b>
7.1	Complete structures . . . . .	121
7.2	Decompositions . . . . .	122
7.3	Summary and contribution . . . . .	125
<b>8</b>	<b>Conclusions</b>	<b>126</b>
8.1	Main contributions . . . . .	126
8.2	Further work . . . . .	127
8.2.1	Precise complexity of instance equivalences . . . . .	127
8.2.2	Products have large width . . . . .	127
8.2.3	Applying constraints techniques when microstructure is implicit . . . . .	128
8.2.4	Refining hereditary clause structures . . . . .	128
8.2.5	SAT and the clause structure . . . . .	128
8.2.6	Hyperresolution and hereditary classes . . . . .	129
8.2.7	Classes beyond hereditary . . . . .	129

8.2.8	Incidence graph of microstructure . . . . .	130
8.2.9	Binarizing instances . . . . .	130
8.2.10	Booleanizing instances . . . . .	130
8.2.11	IS-easy classes of hypergraphs . . . . .	131
8.2.12	Generalized constraint satisfaction . . . . .	131
8.2.13	Microstructure oracles . . . . .	131
8.2.14	Property testing . . . . .	132
8.2.15	Extensions to soft constraints . . . . .	132
8.2.16	Forbidden subgraphs of tree-structured CSP instances . . . . .	132
8.2.17	Canonical constraint relations for non-binary CSPs . . . . .	132
8.2.18	Complete representation, for analysis of width measures . . . . .	133
8.2.19	Linear Space Hypothesis and the complete representation . . . . .	133
<b>References</b>		<b>135</b>
<b>Index of terms</b>		<b>148</b>



# List of Tables

4.1	The structures associated with the family of microstructure representations.	56
5.1	$G$ -free graph classes for $G$ with up to 5 vertices. Key: <i>IS-hard</i> , IS-easy.	93
6.1	$G$ -free clause structure = $\mathcal{F}$ -free vertex-coloured clause structure (with clique-completions assumed)	106
6.2	Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 1, 2, or 3 vertices.	112
6.3	Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 4 vertices and up to 3 colours.	113
6.4	Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 4 vertices and 4 colours.	114

*The process consists, psychologically, in catching one of the transient elements of thought upon the wing and converting it into one of the resting places of the mind. The difference between setting down spots in a diagram to represent recognised objects, and making new spots for the creations of logical thought, is huge.*

—C. S. Peirce, *The Critic of Arguments*, 1892. §424.

# 1

## Introduction

Every instance of a constraint satisfaction problem must be represented in some way. In this thesis I explain why the choice of representation is important, and how the process of transforming between different representations provides insight into whether particular constraint satisfaction problems are difficult or easy. In this chapter I summarize the main contributions of this thesis and provide an overview of the document. I start with a high level overview.

### 1.1 A high level tour

We are surrounded by problems of choice that need solutions. Some are diversions that we can choose to ignore, like puzzles, crosswords, and games. Other problems we must all solve: how and what food to obtain, how and when to prepare it, and how and with whom to eat it. In between these extremes lie problems such as when to schedule examinations, which landing bays to assign to aeroplanes arriving at an airport, and what price to set for a product to maximize profit.

Some problems of choice obviously have many possible solutions: for instance, deciding when to eat food one has already bought, choosing when to concede a point in a negotiation, or deciding what price to ask for a product for which there is predictable demand. For a problem posed in this way, we do not want just any solution, but the best solution possible, or at least a solution that is better than some benchmark. Solving such problems requires choosing the best configuration from a large set of possible solutions, or finding an acceptable one. The set of solutions may even be infinite. For other kinds of problems it is not obvious whether a solution exists at all.

**Example 1.1** (Timetabling). If a student needs to write both English and Mathematics examinations, then these cannot take place at the same time. In such cases, it may be possible to move one of the examinations to a different time. However, too few time slots may have been set aside for every examination to take place without conflicts, no matter how they are shuffled about.

**Example 1.2** (15-puzzle). The sliding block 15-puzzle requires finding a sequence of sliding moves to rearrange blocks to reach the target arrangement in the figure. Sam Loyd’s variant of the 15-puzzle, with the blocks numbered 14 and 15 interchanged, notoriously has no solution: it is not possible to obtain the target position from Loyd’s position via sliding moves.

1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8
9	10	11	12	9	10	11	12
13	14	15		13	15	14	

Target position

Loyd’s position

In 1879, Johnson and Story showed that there are two classes of board arrangements for the sliding block 15-puzzle, that every arrangement can be obtained from every other arrangement in the same class, but that it is not possible to obtain an arrangement in one class from an arrangement in the other class [98]. Around this time, the well-known creator of recreational puzzles Samuel Loyd offered a \$1,000 prize to anyone who could obtain a specific configuration in one class from a specific configuration in the other [105]. It is not clear whether Loyd was aware of the published proof or knew that the problem could not be solved.

**Example 1.3** (PLANAR GRAPH  $k$ -COLOURING). Any map can be coloured with four colours so that no two adjacent regions have the same colour [69]. This colouring can be found efficiently [136]. In contrast, we know of no efficient method which can tell whether three colours are enough for a given map, and which works for every kind of map [61].

**Example 1.4** (Injectivity). Many problems in computing fundamentally involve finding a function mapping integers to integers with some specified properties. Often there is an additional requirement that the desired function never maps two different values to the same value.

In this thesis I deal with the kinds of problems illustrated in Examples 1.1 to 1.4, called combinatorial decision problems. Combinatorial problems describe a collection of objects, possibly infinite, but where the objects are discrete. The task in combinatorial decision problems is then to determine whether or not there is an object in the collection that satisfies all the requirements; this is called a solution. In contrast with decision problems, optimisation problems are those for which solutions usually do exist, but some good solution is required.

Great effort is regularly expended for scientific, engineering, political, economic, and social science applications to construct special-purpose computer programs to solve decision problems. Yet it is common to phrase such problems in a way similar to how puzzles such

as Sudoku are presented: as a set of simple rules that any solution must obey. Sudoku puzzles require every row, column, and block to be filled in to contain each of the digits from 1 to 9, such that the numbers already in the grid are respected.

*Constraint programming* is a framework for expressing and answering combinatorial problems. Instead of specifying the steps required to construct a solution, problems are posed by describing the requirements (called constraints) that any solution must satisfy. Over the last four decades constraint programming has developed a large body of techniques to solve problems, and to produce solutions as efficiently as possible. These techniques are embedded in general-purpose constraint solvers, which allow problems to be described simply by specifying the constraints. The solver then deals with the process of finding a solution automatically. Constraint programming shifts the challenge of solving problems away from constructing special-purpose computer programs, to clearly expressing the constraints that solutions should satisfy, and how those constraints must interact.

*Constraint satisfaction* is the mathematical study of the combinatorial problems that underlie constraint programming. Constraint satisfaction considers infinite classes of instances called constraint satisfaction problems or CSPs. For each instance of a problem, the task is then to decide whether the instance has a solution or not. In general, constraint satisfaction problems are believed to be difficult to decide, in the sense that no efficient algorithms are known that can decide any CSP instance. If an optimisation problem can be solved, then it is possible to pose and solve an associated decision problem, by setting a desirable level of goodness, and asking if there exists a solution that meets that level. If the underlying decision problem is difficult, then the corresponding optimisation problem will also be.

Efficient algorithms do exist for instances with special features, and brute force methods can decide any CSP instance but require a long time to do so. One of the important questions in constraint satisfaction is:

for which CSPs are there methods to decide every problem instance efficiently?

The distinction between *tractable* problems, for which efficient methods exist, and problems for which no such methods are known, is discussed further in Chapter 2.

When the constraints are provided by listing allowed combinations of values, then the constraint satisfaction problem is known to be NP-complete in general. For more compact representations of constraints, the decision problem can be even more difficult.

In contrast, some CSPs are straightforward to solve. Sometimes all instances have properties that can be used to construct an efficient decision procedure. The following two example CSPs are both straightforward.

**Example 1.5** (VICTORIAN LETTERS). Given is an alphabet with  $k$  letters, in ascending order (for English,  $k = 26$ ). Call a letter *Victorian* if the way it is usually depicted contains a straight line, and two straight lines in a letter only ever meet at right angles. (Letters **BDEFGHIJLPRTU** are Victorian in the alphabet used by English.) The problem is to interleave the two alphabetically increasing sequences of Victorian and non-Victorian

letters, so that Victorian letters are never neighbours. There are two solutions for English:

**BADCEKFMGNHOIQJSLVPW~~R~~XTYUZ** and

**AB~~C~~DKEMFNGOHQISJVLWPXRY~~T~~ZU**.

If the number of Victorian letters in the alphabet exceeds  $(k + 1)/2$  then there are no solutions; exactly  $(k + 1)/2$  Victorian letters guarantees a unique solution (this can only happen when  $k$  is odd); and at most  $k/2$  Victorian letters gives rise to at least two solutions.

**Example 1.6** (DIRECTED GRAPH ACYCLICITY). An instance is a system of strict inequalities with  $n$  variables. Each inequality involves two variables, so is of the form  $X < Y$ . Is it possible to assign positive integers  $1, 2, \dots, n$  to the  $n$  variables so that all inequalities are satisfied? This problem has no solution when there is some cycle of inequalities involving at least two variables, of the form  $X_1 < X_2, X_2 < X_3, \dots, X_{k-1} < X_k, X_k < X_1$ . Every other instance has at least one solution. If every pair of variables appears in an inequality, then the instance has at most one solution.

If the instances of a constraint satisfaction problem are restricted in some way, the problem often becomes easier. Two main kinds of restriction have been studied.

**Structural restrictions** One approach is to restrict the *structure*, the way that the constraint variables interact. If the interactions can be rearranged so that they are close to being hierarchical or tree-like, then solutions can be computed efficiently using a divide-and-conquer approach. Note that in Example 1.5 the requirement that the sequences of letters be alphabetically increasing is not essential in order to obtain a solution, as the sequences do not interact. Example 1.5 then has this kind of restricted structure, as the only interaction between letters is that in no adjacent pair of letters may both letters be Victorian.

**Language restrictions** A second approach is to limit the *language*, the types of constraints that can be used to express a problem instance. When the language allowed for expressing constraints is highly structured, it becomes possible to exploit this structure to compute solutions efficiently. Example 1.6 has a restricted language, as it can be expressed using a single kind of constraint: strict inequality between integers.

Both structural and language restrictions have been investigated. I discuss this background further in Chapter 2. The techniques for purely structural and purely language restrictions are highly nontrivial and have been intensively studied for over a decade. Currently it appears that essentially all structural and language restrictions have been found which relate to tractable problems of practical interest. It is believed that the other cases cannot be solved efficiently [22, 74]. Attempts to unify the language restrictions that lead to tractable classes is an area of active investigation in universal algebra. This involves links to open questions about algebras with cyclic terms [10]. The frontier of structural restrictions that guarantee tractability also involves links to open questions related to fixed-parameter tractability [118].

However, constraint satisfaction problems often cannot be expressed using purely structural or purely language restrictions. One of the most common constraints requires the solution to be injective as in Example 1.4, or equivalently, to consist of values that are all different from each other. Such *all-different* constraints are the basic building blocks of scheduling problems, of puzzles such as Sudoku, and for expressing many kinds of mathematical objects. The all-different constraint is known to be tractable; yet it cannot be expressed purely in terms of a language restriction, as its constraint language can express graph colouring problems, and such problems are not known to be tractable. The all-different constraint also cannot be expressed purely in terms of a structural restriction, as it allows potentially every variable in the instance to interact with every other. The special interaction between structure and language in the all-different constraint is what makes this constraint tractable. To study the all-different constraint therefore requires restrictions on both structure and language, acting together.

In this thesis I pursue a systematic study of CSPs which are defined in terms of some combination of structure and language. I focus on those CSPs where restricting just the structure cannot describe the class of instances, and nor can restricting just the language. Such CSPs require *hybrid descriptions*, combining restrictions on structure as well as language. I consider CSPs with constraints that may involve arbitrarily many variables, with no restriction on the number of constraints allowed in a problem instance, and where the structure is not necessarily close to being hierarchical.

In the setting of hybrid descriptions, most existing results on when a CSP can be decided efficiently cannot be applied directly. I therefore propose a basic framework on how to study CSPs with hybrid descriptions, and demonstrate some of the first results in this area. The unifying thread of my work is transforming the representation of a CSP, while ensuring that the new representation falls into a class that is better understood. In this way, many existing results can be unified, and some new results can be derived, to obtain useful information about classes of CSPs with hybrid descriptions.

## 1.2 Document roadmap

I first review the terminology of constraint satisfaction, the background of my research question, and the historical context of previous work in the area, in Chapter 2.

I then discuss fundamental concepts, such as the infrastructure of a CSP instance, how unary constraints can be combined with other constraints to simulate the effect of keeping different domains for each variable, and different notions of equivalence between CSP instances. Following this review of foundations, I review in more detail the main restrictions of structure and language that have previously been used to define tractable classes of CSP instances. Chapter 3 concludes with a discussion of how the three standard representations of CSP instances can be transformed to each other.

It is possible to consider in a single structure all the partial assignments allowed by

the constraints, or the partial assignments disallowed by the constraints. This leads to the microstructure representation, introduced in Chapter 4. This representation considers the structure and the language in a unified structure. This attribute of the microstructure representation turns out to be helpful in the analysis of problems that have efficient algorithms for hybrid reasons.

In Chapter 5 I consider the microstructure of binary constraint satisfaction problems. The discussion culminates in a detailed presentation of the result that many tractable constraint satisfaction problems have perfect microstructures, including some hybrid CSPs.

Classes of problems can also be defined using a richer notion of structure. Each vertex of a structure can be assigned a colour, and an order may also be associated with vertices or colours. Microstructures of CSPs can be regarded as vertex-coloured structures, with each colour representing a distinct variable. This notion leads to classes of vertex-coloured structures that are defined as those not containing some set of forbidden vertex-coloured structures. This yields new tractable classes of CSPs. Chapter 6 deals with vertex-coloured structures and how these can be applied to describe classes of CSPs, including a new hybrid tractable class.

In Chapter 7 I briefly examine a different representation of CSP instances that is intended to highlight limitations of some existing approaches to defining tractable classes by restricting structure of instances in a class.

Chapter 8 concludes this document with a review of some open questions and suggestions for future work.

### 1.3 Contributions

I now briefly describe the main contributions of this thesis. The descriptions use standard terms from the theory of constraint satisfaction, and some terminology in this section will only be defined in later chapters.

In this thesis I make extensive use of the *microstructure representation* of a CSP instance, which will be formally defined in Definition 4.2. This contains in a single structure all allowed or disallowed combinations of assignments of values to variables as specified in the instance. The *clause structure* or *microstructure complement* is a form of the microstructure representation that contains all partial assignments explicitly disallowed by the constraints specified in the instance. Collecting all the information in the instance in a single structure turns out to be helpful in the analysis of problems with hybrid descriptions.

**Hereditary microstructure vs. propagation** Propagation is the central mechanism of constraint programming [4, 14]. I show in Chapter 5 how domain reduction during propagation motivates the notion of hereditary CSPs. These are CSPs where any induced substructure of the microstructure of an instance is the microstructure of some other instance in the class. These CSPs correspond naturally to the partial progress of a constraint solver during search. Moreover, hereditary CSPs can be defined by a set of

forbidden substructures in the microstructure. This may then lead to an efficient algorithm to decide such CSPs, for instance when the set of forbidden substructures is finite, or as in Chapter 5, when this set is infinite but has nice properties.

**Microstructure as product** In Chapter 4 and Chapter 5, I show that the microstructure and the clause structure can be seen as two kinds of products of relational structures. A product form of the microstructure was previously known for the special case of graph homomorphisms [83]. I extend this product representation to general relational structure homomorphisms, and show how to characterize the microstructure and clause structure of CSP instances as products.

**Direct encoding as SAT** By interpreting the vertices of the clause structure as literals, each edge in the clause structure can be seen as a clause in the direct encoding of a CSP instance to SAT. (SAT is the Boolean satisfiability problem, which requires deciding if there is an assignment of true and false values to the variables of a Boolean formula presented in conjunctive normal form, which makes the formula true.) There is therefore a direct correspondence between results about the clause structure and the behaviour of the direct encoding of a CSP instance to SAT. In Chapter 4 I discuss how structural results about the clause structure provide insight into the structure of many SAT instances.

**Perfect microstructure unifies results** I show in Chapter 5 that several classes of CSPs for which efficient algorithms were previously known, can be unified as having microstructures that are perfect graphs. This provides a surprising way to unify several quite disparate classes: instances whose structure forms a tree, CSPs which have chordal microstructures or clause structures, and all-different constraints together with domain reduction. Each such CSP instance can be solved by applying a polynomial-time algorithm to find maximum independent sets in their perfect clause structures [76]. I published this work with my supervisor Peter Jeavons at the main international constraints conference [141].

**Explanation of tractability of all-different** There exists an efficient algorithm for propagation of all-different constraints [134]. Yet, structural and language results considered separately cannot explain the tractability of the all-different constraint. As shown in Chapter 5, this constraint has a perfect microstructure, and this property is maintained by domain reduction. This provides a coherent hybrid reason for the tractability of all-different constraints.

**Microstructure as vertex-coloured structure** In Chapter 6 I show how the microstructure can be seen as a vertex-coloured structure, with colours corresponding to variables, and how forbidding variable-coloured substructures from being contained in the microstructure or the clause structure then captures several interesting classes of CSPs that are not captured without the use of such colourings.



**Broken-triangle new hybrid class** With collaborators Martin Cooper and Peter Jeavons, I introduced a new tractable hybrid class that generalizes tree-structured CSPs. This class is defined by means of forbidden induced substructures called *broken triangles*, relative to the existence of a suitable variable ordering. For this class, each instance can be preprocessed to efficiently find a variable ordering which guarantees backtrack-free search. This yields the first known class that is not tractable for either structural or language reasons but for which such an ordering exists. This work, discussed in Chapter 6, led to a paper at the main European conference on artificial intelligence, where the paper won a best paper award [41]. This has been developed further into a paper in the journal *Artificial Intelligence* [42].

*A relation is a fact about a number of things.*

*... In reality, every fact is a relation.*

—C. S. Peirce, *The Critic of Arguments*, 1892. §416.

# 2

## Background

In this chapter I review the background material underlying this thesis. After discussing some especially useful examples of constraints, I formally review the terminology and basic definitions used in constraint satisfaction, including three standard representations used in the literature. Most of the definitions in this chapter are standard. However, some concepts in Section 2.4.2 and Section 2.4.3 are novel. The final part of the chapter is an overview of the literature that forms the foundation for my contributions.

### 2.1 Constraint satisfaction problems

A constraint satisfaction problem **instance** informally consists of a set of *variables* to which we wish to assign *values*, so that a given set of *constraints* is satisfied. The constraints specify which combinations of values may be assigned to specific groups of variables, and which combinations may not be assigned. When more constraints are added to a problem instance, then the set of solutions generally becomes smaller (although it may also stay the same).

This informal definition of constraint satisfaction can be formalized in many different ways. I define and discuss three classic representations in Section 2.4. Intuitively, a constraint satisfaction problem instance asks the question of whether it is possible to map one object into another, while preserving combinatorial structure. Such a mapping is called an **assignment**, and associates a value with each variable. If an assignment preserves structure in the appropriate way, then it is called a **solution**.

A **CSP** is a collection of constraint satisfaction problem instances under study. In this thesis I am interested in the asymptotic worst-case behaviour of different CSPs. From

the point of view of computational complexity, any CSP that contains only finitely many instances can be solved in linear time by storing the list of accepted instances in a table, in which the input instance can be looked up in constant time. Similarly, a CSP where all but finitely many instances have solutions, can be solved by checking the finite table of exceptions. The vast constants that may be involved in such an analysis, and the possibly immense amount of computational effort required to construct such a table, are ignored for the purposes of asymptotic worst-case computational complexity, even if in practice they matter a great deal. In this thesis, a CSP will refer to an *infinite* collection of instances.

A collection of CSP instances forms a proper class. It is possible to replace “class” in the notion of a collection of CSP instances with “set” if a canonical representation for instances is specified. For instance, instead of allowing any variables and values to be used in a CSP instance, one may restrict attention to those CSP instances which use variable names as well as values from the set of positive integers. Such a CSP then forms a countable set.

To avoid having to deal with the distinction between sets and proper classes, I assume that every CSP is defined relative to a predefined set of variable names and a predefined set of domain values. The variable names can be some countably infinite set, such as the natural numbers or all finite strings over a fixed finite alphabet of letters. For practical purposes the set of values is also usually restricted to be at most countably infinite. In this thesis I work with some countable set of values, such as the natural numbers. Moreover, I require any CSP instance to use only finitely many variables, each taking only finitely many possible values. These are standard assumptions in the theory of constraint satisfaction.

By fixing a set of allowed variable names and a set of allowed domain values, every CSP will then form a set of instances, and the collection of all CSPs will also be a set. Having avoided proper classes in this way, I will therefore use the term **class** informally to denote a collection of objects, either of CSP instances forming a CSP, or a collection of CSPs sharing some common attribute of interest.

## 2.2 Foundations

I now recall some basic definitions from computational complexity and discrete mathematics.

To begin, the following conventions will be useful in many places in this document.

**Definition 2.1** (basic notation).

1. Let  $\mathbb{N}$  denote the set  $\{1, 2, 3, \dots\}$  of all positive integers.
2. For every positive integer  $n$ , let  $[n]$  denote the set of integers  $\{1, 2, \dots, n\}$ .
3. For a real number  $x$ , let  $\lceil x \rceil$  denote the least integer that is not less than  $x$ .
4. For a set  $X$ , let  $|X|$  denote the cardinality of  $X$ . When  $X$  is a finite set then  $|X|$  is simply the number of distinct elements in  $X$ .

5. For a positive integer  $k$ , let  $\binom{X}{k}$  denote the set of all  $k$ -element subsets of set  $X$ . When  $X$  is a finite set and  $0 \leq k \leq |X|$ , then

$$\left| \binom{X}{k} \right| = \binom{|X|}{k} = \frac{|X|!}{k!(|X| - k)!}.$$

┘

### 2.2.1 Computational complexity

For completeness I briefly review the standard terminology of computational complexity. To avoid digressing too far, I assume as given the notion of a Turing machine in both its deterministic and nondeterministic variants; further details can be found in any standard textbook [60, 125].

The set of all finite strings formed from symbols from an alphabet  $\Sigma$  is denoted by  $\Sigma^*$ .

A **problem** is a class  $L$  of problem instances. For every problem  $L$  there is a **decision problem** relative to a description  $L_D$  of the problem, where  $L \subseteq L_D$ . The decision problem requires deciding for any input instance  $\mathcal{P} \in L_D$  whether it is in  $L$  or not.

DECISION PROBLEM FOR  $L$  RELATIVE TO  $L_D$

Input: instance  $\mathcal{P} \in L_D$

Question: is  $\mathcal{P}$  in  $L$ ?

Commonly  $\Sigma = \{0, 1\}$  and the description is then a set  $L_D \subseteq \{0, 1\}^*$  of finite binary strings, representing instances that are regarded as syntactically valid. If the decision problem for  $L_D$  relative to  $\Sigma^*$  is easy compared to the decision problem for  $L$  relative to  $L_D$ , then the description  $L_D$  is often simply taken as  $\Sigma^*$ . A Turing machine  $M$  decides problem  $L$  with description  $L_D$  if  $L$  is the set of those strings  $x$  in  $L_D$  for which  $M$  halts in an accepting state when given  $x$  as input. It is also common to use descriptions  $L_D$  that restrict the possible set of instances in some convenient way, without necessarily being related to representation; in this terminology  $L$  is referred to as a **promise problem** conditioned on the input being from the set of allowed inputs  $L_D$  [66].

A function  $\phi: \mathbb{N} \rightarrow \mathbb{N}$  is **polynomially bounded** if there is a polynomial  $p(x)$  with integer coefficients such that for all  $n \in \mathbb{N}$ ,  $\phi(n) \leq p(n)$ . For a Turing machine  $M$  that always halts in a finite number of steps on any input, let  $T_M$  be the function associating with each  $n$  the maximum number of steps taken by  $M$  to halt on any input of length  $n$ . A Turing machine  $M$  is **polynomial-time bounded** if  $T_M$  is polynomially bounded.

A problem can be decided in **polynomial time**, or is **polynomial-time decidable**, if there is a polynomial-time bounded Turing machine that decides the problem.

A function  $\phi: \mathbb{N} \rightarrow \mathbb{N}$  is **logarithmically bounded** if there is a constant  $k \in \mathbb{N}$  such that for all  $n \in \mathbb{N}$ ,  $\phi(n) \leq k \log n$ . Let  $S_M$  be the function associating with each  $n$  the maximum number of worktape cells used by  $M$  when given input  $x$  of length  $n$  (the output

size is not part of this number). Turing machine  $M$  is **log-space bounded** if  $S_M$  is logarithmically bounded.

Note that a Turing machine that is polynomial-time bounded or log-space bounded always halts in a finite number of steps.

A **many-one reduction** from problem  $L_1$  to problem  $L_2$  is a function  $\alpha: \Sigma^* \rightarrow \Sigma^*$  such that  $\alpha(\mathcal{P}) \in L_2$  if, and only if,  $\mathcal{P} \in L_1$ . A Turing machine  $M$  computes a many-one reduction  $\alpha$  from  $L_1$  to  $L_2$  if  $M$  halts for all inputs  $\Sigma^*$ ; whenever  $M$  is given a string  $x \in L_1$  as input then it halts with  $\alpha(x)$  as output, and whenever  $M$  is given a string not in  $L_1$  as input then it halts with a string not in  $L_2$  as output. A many-one reduction  $\alpha$  is **polynomial-time** if there is a deterministic polynomial-time bounded Turing machine which computes  $\alpha$ , and **log-space** if there is a deterministic log-space bounded Turing machine which computes  $\alpha$ . Problems  $L_1$  and  $L_2$  are **log-space equivalent** if  $L_1$  is log-space many-one reducible to  $L_2$  and  $L_2$  is log-space many-one reducible to  $L_1$ .

**Definition 2.2** (basic complexity notions).

1. **P** is the class of decision problems which can be decided by a polynomial-time bounded deterministic Turing machine.
2. **NP** is the class of decision problems which can be decided by a polynomial-time bounded nondeterministic Turing machine.
3. A decision problem is **NP-complete** if it is in NP, and every problem in NP can be many-one reduced to it, using log-space reductions.  $\square$

In this thesis I model every CSP as a decision problem. The description  $L_D$  contains all syntactically valid CSP instances, and  $L$  is the class of all those instances which have at least one solution. I refer to  $L$  as the CSP.

It is sometimes convenient to refer to **solving** a problem rather than deciding it. For constraint satisfaction problems with a finite domain, finding a solution can be done in a standard way by calling the decision procedure a polynomial number of times [35, Theorem 9]. When dealing with CSPs that can be decided in polynomial time, I will therefore not belabour the distinction between deciding and finding a solution.

### 2.2.2 Standard definitions

I now recall some definitions that are generally useful.

**Definition 2.3** (tuples). An **ordered pair**  $(u, v)$  is the set  $\{\{u\}, \{u, v\}\}$ . A function can be regarded as a set of ordered pairs. For a function  $\sigma$ , let  $\sigma_i$  denote the ordered pair  $(i, \sigma(i)) \in \sigma$ .

Let  $I$  be a set with  $r$  elements. A **tuple of arity**  $r$  over sets  $\{X_i \mid i \in I\}$  indexed by  $I$  is a function  $\sigma: I \rightarrow \bigcup \{X_i \mid i \in I\}$  with  $\sigma(i) \in X_i$  for every  $i \in I$ . (Mention of the index set or the sets over which a tuple is defined may be omitted if these are clear from the

context.) If  $X_i = X$  for each  $i \in I$ , then a tuple over  $\{X_i \mid i \in I\}$  is referred to as a tuple over  $X$ . A tuple  $\sigma$  indexed by  $I = \{i_1, i_2, \dots, i_r\}$  is usually written as  $(\sigma_{i_1}, \sigma_{i_2}, \dots, \sigma_{i_r})$ ,  $\{\sigma_i \mid i \in I\}$ , or sometimes as  $(\sigma_j)_{j \in I}$ . When  $i \notin I$  and  $x \notin \{\sigma(j) \mid j \in J\}$ , then it is sometimes also convenient to write  $((\sigma_j)_{j \in I}, x)$  to denote the tuple  $\sigma \cup \{(i, x)\}$ . The arity of a tuple  $\sigma$  is denoted  $\rho(\sigma)$ , and  $\rho(\sigma) = |I|$  if  $\sigma$  is indexed by  $I$ . A tuple  $\sigma$  of arity 2 can be regarded as an ordered pair  $(\sigma_1, \sigma_2)$ , and vice versa.  $\square$

It is sometimes convenient to allow tuples of arity 0. This usage is common in database theory, where such arity 0 tuples denote constants. However, in keeping with common practice in the constraints literature, I will assume here that arity is always positive.

**Definition 2.4** (Cartesian product). The **Cartesian product** of the tuple of sets  $(X_{i_1}, X_{i_2}, \dots, X_{i_r})$  is the set  $\prod_{j=1}^r X_{i_j}$ , usually written  $X_{i_1} \times X_{i_2} \times \dots \times X_{i_r}$ , of all tuples of arity  $r$  over sets  $\{X_i \mid i \in I\}$  indexed by  $I = \{i_j \mid j \in [r]\}$ . The Cartesian product of  $r$  copies of the set  $X$ , indexed by  $[r]$ , is denoted  $X^r$ .  $\square$

A relation collects together tuples.

**Definition 2.5** (relations).

1. A **relation** over sets  $(X_{i_1}, X_{i_2}, \dots, X_{i_r})$  of arity  $r$  is a subset of the **complete relation**  $\prod_{j=1}^r X_{i_j}$ . A **proper relation** is a relation that is not the complete relation. A relation over  $X$  of arity  $r$  is a subset of  $X^r$ . The arity of a relation  $R$  is denoted  $\rho(R)$ .
2. If  $\rho(R) = 1$  then  $R$  is said to be **unary**, and if  $\rho(R) = 2$  then it is said to be **binary**.
3. If  $R$  is a binary relation, then  $xRy$  denotes  $(x, y) \in R$ .
4. A binary relation  $R$  **relates**  $x$  to  $y$  if  $xRy$ , and  $x$  and  $y$  are said to be **related** by  $R$  if either  $xRy$  or  $yRx$ .
5. The **inverse** of a binary relation  $R$  is the relation  $R^{-1} = \{(x, y) \mid yRx\}$ .
6. The **image** of a binary relation  $R$  under a set  $A$  is  $R(A) = \{y \mid \exists x \in A \ xRy\}$ .
7. A binary relation  $R$  over  $X$  is **symmetric** if  $xRy$  implies  $yRx$  for every  $x, y \in X$ ; **antisymmetric** if  $xRy$  and  $yRx$  implies  $x = y$  for every  $x, y \in X$ ; **transitive** if  $xRy$  and  $yRz$  implies  $xRz$  for every  $x, y, z \in X$ ; **reflexive** if  $(x, x) \in R$  for every  $x \in X$ ; and **irreflexive** if  $(x, x) \notin R$  for any  $x \in X$ . A **preorder** is a binary relation that is transitive and reflexive. A **partial order** is a preorder that is also antisymmetric. A **total order** is a partial order for which whenever  $x$  and  $y$  are distinct elements of  $X$ , then either  $xRy$  or  $yRx$ . A **strict total order** is a transitive and irreflexive relation for which whenever  $x$  and  $y$  are distinct elements of  $X$ , then either  $xRy$  or  $yRx$ .

8. An **equivalence relation** is a binary relation  $\equiv$  that is reflexive, symmetric, and transitive. An **equivalence class** with respect to an equivalence relation  $\equiv$  is a class  $\mathcal{C}$  such that every member of  $\mathcal{C}$  is related by  $\equiv$  to every other member.
9. For binary relations  $R$  and  $R'$ , if  $R \subseteq R'$  then  $R$  is said to be **finer** than  $R$ , and  $R'$  is said to be **coarser** than  $R$ .

Note that a function  $f: X \rightarrow Y$  can be regarded as a relation over  $(X, Y)$  with the additional property that if  $(x, y) \in f$  and  $(x, z) \in f$ , then  $y = z$ .

**Definition 2.6** (basic notation, part 2).

1. A **strictly totally ordered set** is a set  $X$  that has an associated strict total order over  $X$ . If  $\leq$  is a total order over  $X$ , then  $<$  denotes the strict total order obtained by removing all pairs  $(x, x)$  from the relation  $\leq$ , for every  $x \in X$ .
2. If  $X$  is a strictly totally ordered set, then there is a bijection between  $\binom{X}{k}$  and the set of **increasing**  $k$ -element tuples over  $X$ , which are those tuples  $(x_1, x_2, \dots, x_k)$  for which  $x_1 < x_2 < \dots < x_k$ . I will therefore use the same notation  $\binom{X}{k}$  for the set of increasing  $k$ -element tuples of  $X$ .

**Definition 2.7** (projection). The **projection** of tuple  $\sigma$  over  $(X_{i_1}, X_{i_2}, \dots, X_{i_r})$  to  $J \subseteq \{i_1, i_2, \dots, i_r\}$  is the tuple  $\sigma \cap \bigcup \{\{j\} \times X_j \mid j \in J\}$ , written as  $\sigma_J$ . The projection of relation  $R$  over  $(X_{i_1}, X_{i_2}, \dots, X_{i_r})$  to  $J \subseteq \{i_1, i_2, \dots, i_r\}$  is the relation  $R_J$  of projections of all tuples in  $R$  to  $J$ .

Note that  $\sigma_{\{j\}} = \{\sigma_j\}$ .

### 2.2.3 Relational structures, graphs, and hypergraphs

Relational structures are fundamental building blocks for my work.

**Definition 2.8** (relational structure). A **relational structure**  $S$  is a pair  $(V(S), (Q_i)_{i \in I})$ , where  $V(S)$  is the set of **vertices** of  $S$ , and the set  $I$  is used to index the relations  $Q_i$  over  $V(S)$  in the structure. If  $V(S)$  is a finite set then  $S$  is **finite**. The **language** of  $S$  is the set of relations  $\Gamma(S) = \{Q_i \mid i \in I\}$  that occur in  $S$ . —

**Definition 2.9** (signature). The **signature** of a relational structure  $S = (V(S), (Q_i)_{i \in I})$  is the tuple  $(\rho(Q_i))_{i \in I}$  of arities of its relations, and its **arity** is  $\rho(S) = \max\{\rho(Q_i) \mid i \in I\}$ .

When  $I$  contains a small finite number  $q$  of elements, then it is usual to write the tuple of relations as a list  $Q_1, Q_2, \dots, Q_q$  instead of  $(Q_i)_{i \in [q]}$ .  $S$  is then conveniently expressed as  $(V(S), Q_1, Q_2, \dots, Q_q)$ . Note that the order in this enumeration is arbitrary but fixed for each instance. A common case is when  $q = 1$ , as illustrated in the following example.

**Example 2.10** (graph). A **directed graph**  $G$  is a relational structure  $(V(G), E(G))$ . The **edges** of  $G$ , denoted  $E(G)$ , form a binary relation on the vertices  $V(G)$  of  $G$ . A **graph**  $G$  is a relational structure  $(V(G), E(G))$ , where  $E(G)$  is a binary relation on  $V(G)$  that is symmetric and irreflexive.  $\perp$

If  $u \neq v$ , it is usual to treat the set  $\{(u, v), (v, u)\}$  interchangeably with the set  $\{u, v\}$ . When referring to graphs, I will therefore sometimes refer to edges as 2-element subsets of vertices, and sometimes as symmetric pairs of tuples of distinct vertices. With this definition, a directed graph can have self-loops of the form  $(u, u)$  as edges, but a graph cannot.

Hypergraphs generalize graphs to the case when the number of vertices in an edge is not required to be exactly two. The following definitions are standard [12].

**Definition 2.11** (hypergraph). A **hypergraph**  $H$  is a pair  $(V(H), E(H))$ , where  $V(H)$  are the **vertices** of  $H$  and  $E(H)$  are the **edges** or **hyperedges** of  $H$ . Every edge in  $E(H)$  is a non-empty subset of  $V(H)$ . A **hypergraph on a set**  $X$  is a hypergraph with  $X$  as its vertices. Hypergraph  $G$  is a **subhypergraph** of hypergraph  $H$  if  $V(G) \subseteq V(H)$  and  $E(G) \subseteq E(H)$ , and  $G$  is an **induced subhypergraph** of  $H$  if  $G$  is a subhypergraph of  $H$  and every edge of  $H$  that is a subset of  $V(G)$  is also an edge of  $G$ .  $\perp$

Note that some authors use the term **partial hypergraph** to refer to subhypergraphs.

It will also be useful to consider the hypergraph with edges that are subsets of vertices that are not edges of a given hypergraph.

**Definition 2.12** (complement of hypergraph). The **complement** of a hypergraph  $H = (V, E)$  is a hypergraph  $\overline{H} = (V, \overline{E})$ , where  $\overline{E}$  contains every non-empty subset of distinct vertices from  $V$  which are not contained in  $E$ .  $\perp$

A hypergraph may have singleton edges, consisting of single vertices, but in this thesis I ignore any empty edges.

**Definition 2.13** ( $k$ -section of hypergraph). The  **$k$ -section** of a hypergraph  $H$  consists of all its edges of size up to  $k$ , as well as all size  $k$  subsets of the edges.

**Definition 2.14** (Gaifman graph). For a hypergraph  $H$ , the **Gaifman graph** or **primal graph** of  $H$  is a graph with vertices  $V(H)$  and edges between distinct vertices  $u$  and  $v$  precisely when  $u$  and  $v$  are both contained in some edge of  $H$ .  $\perp$

I will use graphs and hypergraphs throughout the thesis. They also serve as the base case for concepts relating to relational structures, since graphs and hypergraphs can be thought of as special kinds of relational structure, as in Example 2.10.

**Definition 2.15** (graph of directed graph). If  $G$  is a directed graph, then the **graph of**  $G$ , denoted  $\mathcal{G}(G)$ , is the graph with the same vertices as  $G$ , and  $\{u, v\}$  is an edge of  $\mathcal{G}(G)$  whenever  $u, v$  are distinct vertices of  $G$  and  $(u, v) \in E(G)$ .  $\perp$



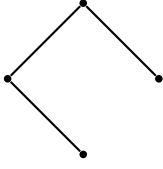


Figure 2.1:  $P_4$ , a path on 4 vertices

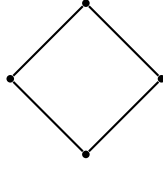


Figure 2.2:  $C_4$ , a cycle on 4 vertices

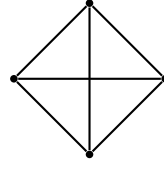


Figure 2.3:  $K_4$ , a complete graph on 4 vertices

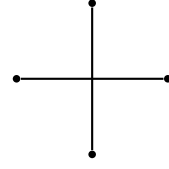


Figure 2.4:  $\overline{C_4}$ , complement of  $C_4$

The following is analogous to Definition 2.12, but restricted to subsets of size 2.

**Definition 2.16** (complement of graph). The **complement** of a graph  $G = (V, E)$  is a graph  $\overline{G} = (V, \overline{E})$ , where  $\overline{E}$  contains every pair of distinct vertices in  $V$  which are not contained in  $E$ . Sometimes  $\overline{G}$  is also denoted **co- $G$** .  $\square$

Let  $G = (V, E)$  and  $H = (W, F)$  be two graphs. A **graph isomorphism** from  $G$  to  $H$  is a bijection  $\phi: V \rightarrow W$  that preserves graph structure, in the sense that  $\{u, v\} \in E$  if, and only if,  $\{\phi(u), \phi(v)\} \in F$ . Graphs  $G$  and  $H$  are **isomorphic**, written  $G \simeq H$ , if there exists an isomorphism from  $G$  to  $H$ . Every subset of vertices  $U \subseteq V(G)$  **induces** a subgraph  $G[U] = (U, E(G) \cap \binom{U}{2})$  of graph  $G$ , containing those edges of  $G$  with both vertices in  $U$ .  $H$  is an **induced subgraph** of  $G$  if  $V(H) \subseteq V(G)$  and  $G[V(H)] = H$ .  $H$  is a **subgraph** of  $G$  if  $V(H) \subseteq V(G)$  and  $E(H) \subseteq E(G)$ .

*Observation 2.17.* An induced subgraph or subhypergraph can be obtained by removing vertices (when a vertex is removed, so are all edges including that vertex). A subgraph or subhypergraph can be obtained by removing either vertices or edges.

Similarly to the case for hypergraphs and graphs, an **induced substructure** of a relational structure  $S = (V(S), (Q_i)_{i \in I})$  is obtained by removing vertices from  $S$ ; when a vertex  $u \in V(S)$  is removed then so are all tuples involving  $u$ . If  $W \subseteq V(S)$  then the substructure  $S[W]$  of  $S$  induced by  $W$  is the structure

$$(W, (Q_i \cap W^{\rho(Q_i)})_{i \in I})$$

obtained by removing from  $S$  all vertices not in  $W$ . A **substructure** of  $S$  is obtained by removing vertices or tuples from  $S$ .

A **path** on  $s$  vertices is a graph isomorphic to  $P_s = ([s], \{\{1, 2\}, \{2, 3\}, \dots, \{s-1, s\}\})$ . A **cycle** on  $s$  vertices, or an  **$s$ -cycle**, is a graph isomorphic to  $C_s = ([s], \{\{1, 2\}, \{2, 3\}, \dots, \{s-1, s\}, \{s, 1\}\})$ . A **complete graph** on  $s$  vertices is a graph isomorphic to  $K_s = ([s], \binom{[s]}{2})$ .

**Example 2.18** (basic graph notions). Every induced subgraph is a subgraph, but not every subgraph is an induced subgraph. The graph  $C_4$  in Figure 2.2 is a subgraph of the graph  $K_4$  in Figure 2.3, but not an induced subgraph. The graph  $P_4$  in Figure 2.1 is a subgraph of  $C_4$  and  $K_4$ , but not an induced subgraph of either. The complement  $\overline{C_4}$  of  $C_4$  is the graph with four vertices and two disjoint edges, illustrated in Figure 2.4.  $\square$

A **clique** is a set of vertices that induces a complete subgraph. A complete graph is sometimes also referred to as a clique. A clique with  $s$  vertices is known as an  $s$ -clique.

**Example 2.19.** Many practical problems can be reduced to finding the largest clique in a suitably constructed graph. The decision version of this problem is in Karp's list of 21 NP-complete problems [99].

CLIQUE

Input: graph  $G$ , non-negative integer  $s$

Question: does  $G$  contain an induced subgraph isomorphic to  $K_s$ ?

A graph is **completely disconnected** if it has no edges. A set of vertices  $X$  of a graph  $G$  is an **independent set** if  $G[X]$  is completely disconnected. Graph  $G$  is **bipartite** if the vertices can be partitioned into two independent sets  $X$  and  $Y$ . Note that  $X$  is a clique in  $G$  precisely when  $X$  is an independent set in  $\overline{G}$ , the complement of  $G$ .

**Example 2.20.** Another commonly encountered practical problem is finding a **proper colouring** of a graph. A **vertex-colouring** is an assignment of colours to vertices of a graph, and is **proper** if no edge connects vertices of the same colour. The **chromatic number** of a graph is the least number of colours required for a proper vertex-colouring. The decision version, CHROMATIC NUMBER, was another of Karp's 21 NP-complete problems [99].

CHROMATIC NUMBER

Input: graph  $G$ , non-negative integer  $t$

Question: can each of the vertices of  $G$  be assigned one of  $t$  colours, so that the vertices of every edge of  $G$  are assigned different colours?

A colouring can also be thought of as a function from vertices to the set of colours. By renaming the  $t$  colours, it is enough to consider colourings using the integers in  $[t]$ . A proper colouring then exists precisely when there is a function  $c: V(G) \rightarrow [t]$  such that whenever  $\{u, v\} \in E(G)$ , then  $c(u) \neq c(v)$ .

A graph  $G$  is **perfect** if for every induced subgraph  $H$  of  $G$ , the chromatic number of  $H$  is equal to the number of vertices of the largest clique contained in  $H$  [114]. Perfect graphs form an important class of graphs with many applications. They can be thought of as an enlargement of the class of bipartite graphs to include graphs that are not bipartite but still have many of the desirable properties of bipartite graphs; see [86] for a survey of many subclasses of perfect graphs. The following is one of these desirable properties: note that the complement of a bipartite graph is not bipartite, in general.

**Proposition 2.21** (Perfect Graph Theorem [114]). *The complement of a perfect graph is a perfect graph as well.*

All graphs with at most 4 vertices are perfect.  $C_5$  is not perfect, and every 5-vertex graph that is not perfect is isomorphic to  $C_5$ . This graph requires 3 colours for a proper colouring, but has no clique on more than 2 vertices. Note that  $C_5$  is (isomorphic to) its own complement.

#### 2.2.4 General foundations

I collect here some simple definitions that will come in useful in later chapters.

**Definition 2.22** (underlying set of tuple). The **underlying set of a tuple**  $\sigma$  indexed by  $I$  is the set  $\{\sigma(i) \mid i \in I\}$ .  $\square$

The underlying set of a tuple  $\sigma$  may contain from 1 to  $\rho(\sigma)$  elements. The underlying sets of all tuples in a relational structure together form the edges of a hypergraph.

**Definition 2.23** (hypergraph of relational structure). The **hypergraph of relational structure**  $S$  is a hypergraph  $H(S)$ , with the same vertices as the structure  $S$ , and edges  $E(H(S)) = \{\{u(1), u(2), \dots, u(r)\} \mid (u_1, u_2, \dots, u_r) \in R, R \in \Gamma(S)\}$ .  $\square$

Complements of relations and relational structures will prove important in subsequent chapters.

**Definition 2.24** (complement of relation). The **complement** of a relation  $R$  over  $D$  of arity  $r$  is  $\bar{R} = D^r \setminus R$ .  $\square$

**Definition 2.25** (complement of relational structure). The **complement** of relational structure  $S = (V(S), (Q_i)_{i \in I})$  is  $\bar{S} = (V(S), (\bar{Q}_i)_{i \in I})$ .  $\square$

In the definition of complement of a relational structure, recall that each relation in the structure is over the vertices of the structure. It is also important that the complement is defined with respect to the class of structures under consideration. The complement of a binary relation may have self-loops. The complement of a graph was defined in Definition 2.16 to be a graph and therefore does not contain self-loops. When a class of graphs may contain self-loops, or a class of relational structures may contain repeated values in a tuple, then the complement must allow these. Similarly, if a class of graphs may not contain self-loops, or a class of relational structures may not contain repeated values in tuples, then these restrictions must be taken into account when taking complements. The correct definition is usually clear from the context, so will not cause ambiguity. In particular, I will take complements of graphs in accordance with Definition 2.16, and complements of relational structures in accordance with Definition 2.25.

### 2.3 Examples of constraint satisfaction problems

In this section I provide examples of constraint satisfaction problems, before reviewing various representations of CSPs in the following section.

A constraint expresses a property of its parameters. The parameters of a constraint may be variables or sets of values from which variables must take their values. I will write constraints in the form

$$\text{CONSTRAINT}(\pi_1, \pi_2, \dots; D_1, D_2, \dots),$$

where **CONSTRAINT** is the name of the constraint,  $\pi_i$  are parameters that are variables, and each  $D_i$  parameter is a set of values. The indices for variables or sets of values are used to distinguish different variables or sets of values from each other. In general, different constraints may have different lists of parameters. Such constraints are sometimes called **global constraints**, as they can potentially be applied to any subset of the variables [152].

The following example illustrates a CSP that naturally contains global constraints.

**Example 2.26** (Nonograms). A nonogram is a type of logic puzzle. Given is a grid of  $m$  by  $n$  squares, each of which must either be 1 (represented by a filled in square) or 0 (represented by a cross). The constraints are represented as sequences of numbers, one sequence for each row and for each column of the grid. Each number  $i$  describes a consecutive sequence of  $i$  filled squares. Such a sequence is adjacent either to the edge of the grid, or to at least one unfilled square.

Nonogram constraints may describe relations with large numbers of tuples: a sequence  $b_1 b_2 \dots b_k$  of  $k$  integers in a row or column containing  $p$  squares describes every possible way to form sums of natural numbers  $\sum_{i=0}^k a_i + \sum_{i=1}^k b_i = p$ , in other words the number of integer partitions of  $p - \sum_{i=1}^k b_i$ . Here  $a_{i-1}$  is the number of contiguous unfilled squares before the  $i$ -th block of  $b_i$  filled squares. Note that the number of integer partitions of  $n$  grows asymptotically as  $2^{3.7\sqrt{n}}/n$ , and therefore grows faster than any polynomial [102].

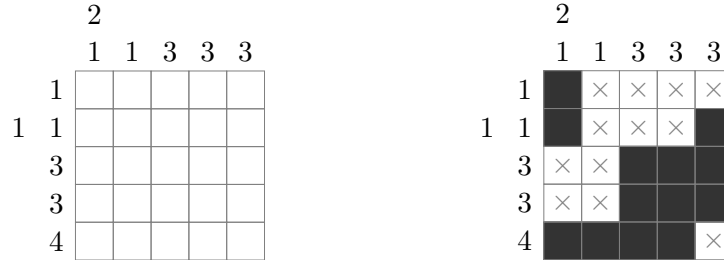


Figure 2.5: An example  $5 \times 5$  nonogram instance and its solution.

The natural way to model nonograms is by Boolean constraints of arity  $m$  and  $n$ , where  $m$  and  $n$  are the dimensions of the grid used in the instance. Nonogram constraints are therefore global constraints. Although nonograms are usually regarded as recreational puzzles, they are closely related to the reconstruction of images, for instance by a medical electromagnetic scanner. One form of such discrete tomography problems requires reconstructing all filled in grid squares by means of single sums in each row and each column. This related problem only allows fairly coarse control over the kinds of structures that can be distinguished, and is tractable for two-dimensional grids [77]. For dimension

three or higher, finer control can be exercised over the instances, and the problem is NP-complete [94]. In contrast, the lengths of individual runs in each row and column are given in a nonogram instance. This allows fine control for expressing structure even for two dimensions, and with arbitrarily large grids allowed, the class of nonogram instances is NP-complete [150].  $\square$

Constraint satisfaction problems in practice generally use constraints involving different numbers of variables, and may also use constraints with different numbers of variables in different instances. In this thesis I will use global constraints, and will explicitly highlight when constraints must involve a fixed number of variables. Otherwise I will not assume this restriction.

Some constraints often occur as building blocks of constraint satisfaction problems. Every constraint can be regarded as a constraint satisfaction problem in its own right, so I will not belabour the distinction.

The simplest constraint is one that has no effect.

**Example 2.27** (anything-goes). The *anything-goes* constraint  $*(u_1, u_2, \dots, u_n; D)$  allows every possible assignment of values in  $D$  to each of the variables  $u_1, u_2, \dots, u_n$ . When the list of variables and domain are clear from the context, the anything-goes constraint is often just denoted  $*$ .  $\square$

Another common constraint simply restricts the values a single variable may take.

**Example 2.28** (UNARY). The constraint  $\text{UNARY}(u; D)$  on variable  $u$  with respect to a set  $D$  of values is satisfied by an assignment  $\phi$  if the value assigned to  $u$  by  $\phi$  belongs to  $D$ .

The UNARY constraint underlies many other constraints. It is a basic building block from which other constraints can be built. Unary constraints are not of interest individually. They are useful when multiple constraints are combined in a problem instance.

Another of these building blocks is the constraint that requires solutions to be injective functions.

**Example 2.29** (INJECTIVE). The constraint

$$\text{INJECTIVE}(u_1, u_2, \dots, u_n; D)$$

on the list of variables  $u_1, u_2, \dots, u_n$  with respect to the set of values  $D$  is satisfied by an assignment  $\phi$  if  $\phi$  assigns values from  $D$  to each  $u_i$  so that no value is repeated.  $\square$

When several constraints are combined, I will write  $\{\text{CONSTRAINT}_1, \text{CONSTRAINT}_2, \dots\}$  to denote the single constraint formed by the combination of this list of constraints. Note that  $\{\text{CONSTRAINT}, \text{CONSTRAINT}\}$  has precisely the same effect as  $\text{CONSTRAINT}$ , so the set notation for combining constraints is justified even though it is slightly imprecise.

Just like UNARY, the INJECTIVE constraint is useful as a building block for constructing problem instances but is not interesting on its own. Combining several INJECTIVE

constraints yields many kinds of interesting problem instances; several of these are common types of puzzles.

**Example 2.30** (LATIN SQUARE COMPLETION). A Latin square is a puzzle on an  $n$  by  $n$  grid of squares, some containing numbers. The grid is to be completed, so that each number in  $[n]$  is used exactly once in each row and each column. When the size of the grid is allowed to be arbitrarily large, this yields the following decision problem:

**LATIN SQUARE COMPLETION**

Input: a positive integer  $n$ , and an  $n$  by  $n$  grid, with some grid squares containing numbers and others blank

Question: can all blank grid squares be assigned numbers from  $[n]$  so that no number occurs twice in any row or column?

This decision problem is NP-complete [38].

LATIN SQUARE COMPLETION can also be expressed using INJECTIVE constraints for each row and column of variables, with additional UNARY constraints specifying the numbers that are initially given for some of the grid squares.  $\square$

**Example 2.31** (SUDOKU). Sudoku is the common name for a type of logic puzzle. Given is a 9 by 9 grid of squares, divided into 9 blocks of  $3 \times 3$  squares. Each square must be filled in with a number from the set  $[9]$ . Some of the squares already have numbers. Each row, column, and block is subject to an INJECTIVE constraint. A Sudoku instance is illustrated in Figure 2.6. A CSP is obtained by allowing grids of arbitrarily large size. For the Sudoku problem instances that are presented as puzzles,  $n$  is usually 3.

**SUDOKU**

Input: a positive integer  $n$ , and an  $n^2$  by  $n^2$  grid of squares containing  $n^2$  blocks of  $n$  by  $n$  squares, with numbers in some squares

Question: can the grid be completed with numbers from  $[n^2]$ , so that no number occurs twice in any row, column or block?

SUDOKU can alternately be described as LATIN SQUARE COMPLETION with additional INJECTIVE constraints on blocks.  $\square$

**Example 2.32** (FUTOSHIKI). Futoshiki is the common name for a type of logic puzzle involving inequalities. Given is a grid of  $n$  by  $n$  squares (typically  $n = 5$  or  $n = 7$  for published puzzles). Each square takes a value from  $[n]$ , and there is an INJECTIVE constraint on each row and column. Further, constraints of the form  $x < y$  are imposed between some adjacent squares  $x$  and  $y$  in the grid. Finally, some of the squares may already be filled in. An instance of Futoshiki is illustrated in Figure 2.7.

When arbitrarily large grid sizes are allowed, the FUTOSHIKI instances form a CSP. FUTOSHIKI may alternately be regarded as LATIN SQUARE COMPLETION with additional strict inequality constraints between some pairs of adjacent squares.

4	6				1			
		2		9	6			
	3						6	8
							3	5
			6		5			
7	1							
8	4						7	
			5	1		9		
			3				2	4

Figure 2.6: An instance of SUDOKU.

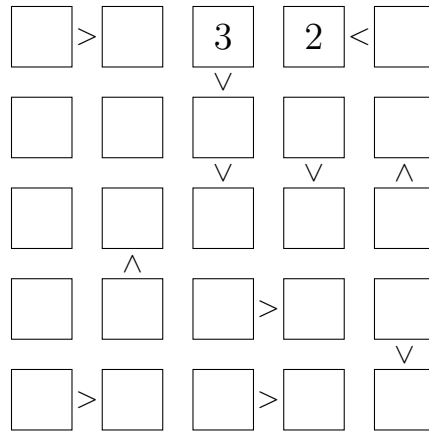


Figure 2.7: An instance of Futoshiki.

#### FUTOSHIKI

**Input:** grid of  $n$  by  $n$  squares, some with numbers, and some  $<$  constraints between pairs of adjacent squares (that are both in the same row or the same column)

**Question:** can the grid be completed with numbers from  $[n]$ , so that no number occurs twice in any row or column, and so that all  $<$  constraints are respected?

An instance of Futoshiki is illustrated in Figure 2.7. └

**Example 2.33** (SUM). The constraint  $\text{SUM}(u_1, u_2, \dots, u_r; \{C\})$  requires that the sum of

the arguments supplied for the parameters  $u_1, u_2, \dots, u_r$  is precisely equal to the value  $C$ :

$$\sum_{i=1}^r u_i = C.$$

Collecting together all instances of  $\text{SUM}(u_1, u_2, \dots, u_r; \{C\})$  for positive integers  $r$  and  $C$  yields a CSP. In the absence of any other constraints, each instance of this CSP is rather easy to solve.  $\square$

**Example 2.34** (Magic Square). A magic square is a square grid containing some numbers, which must be completed so that the sum of numbers in each row and column is the same. Latin squares as well as Futoshiki and Sudoku puzzles are magic squares with the row and column sums equal to  $n(n-1)/2$ . A magic square can be expressed using SUM constraints applied to the variables in each row and each column.  $\square$

## 2.4 Classic representations

Three representations of constraint satisfaction problems have been used most frequently: variable-value, first-order logic, and homomorphism. For each a different syntax is used to describe the instance. The instance description defines the syntactically valid instances  $L_D$  relative to which decision problems  $L$  can be defined, as discussed in Section 2.2.1.

As a running example I use the INJECTIVE constraint from Example 2.29.

### 2.4.1 Variable-value representation

The **variable-value representation** is now standard in the constraint satisfaction literature, and is also often used in the constraint programming community more generally.

**Definition 2.35** (instance description: variable-value). In the variable-value representation, the instance description is a tuple  $(V, D, C)$ .

1. Set  $V$  is a finite set of **variables**.
2. Set  $D$  is a **domain** consisting of **values**. An instance which has a domain with only two elements is called **Boolean**.
3. Set  $C$  contains **constraints**. Each constraint in  $C$  is a pair  $(\sigma, R)$ .
  - (a) The **constraint scope**  $\sigma$  is a tuple over  $V$  of arity  $\rho(\sigma)$ .
  - (b) The **constraint relation**  $R$  is a relation over  $D$  of the same arity  $\rho(\sigma)$  as the constraint scope.
  - (c) The constraint scope and the constraint relation are both indexed by the same set, which can usually be taken to be  $[\rho(\sigma)]$ .  $\square$

In a **binary** CSP instance, all constraint relations have arity 2.



**Example 2.36** (INJECTIVE: variable-value representation). Let  $V = \{u_1, u_2, \dots, u_n\}$  be the set of variables of constraint  $\text{INJECTIVE}(u_1, u_2, \dots, u_n; D)$ . Let

$$R = \{(a_1, a_2, \dots, a_n) \in D^n \mid |\{a_1, a_2, \dots, a_n\}| = n\}$$

be the constraint relation; it contains all tuples without repeated values. Finally, let  $C = \{((u_1, u_2, \dots, u_n), R)\}$  be a set of constraints containing a single constraint. The instance is then  $(V, D, C)$ . In this case, the constraint relation  $R$  will contain  $|D|(|D| - 1) \dots (|D| - n + 1)$  tuples.  $\square$

The **variables in the instance**  $(V, D, C)$  are the variables  $V$ . The **variables in a constraint**  $(\sigma, R) \in C$  are the variables in the underlying set of  $\sigma$ . The **values in the instance**  $(V, D, C)$  are the values in the domain  $D$ .

It is sometimes useful to consider the scope of a constraint as consisting of a set of variables, instead of a tuple. The set of variables of a scope of arity  $r$  may contain fewer than  $r$  variables if the scope contains repeated variables.

## 2.4.2 Partial assignments and solutions

Before dealing with the first-order logic and homomorphism representations, I first need to discuss partial and complete assignments, which map some or all of the variables to values. A complete assignment that respects the constraints is a solution. I first discuss concepts related to props or consistent partial assignments. These form the basic units from which the variable-value representation is built.

**Definition 2.37** (partial assignment). A **partial assignment** of a CSP instance is a function from a subset of the variables in the instance to the values in the instance. When a partial assignment is defined over the set of all variables, it is called a **complete assignment**.  $\square$

A partial assignment  $\phi$  is a set of ordered pairs, each of the form  $(u, \phi(u))$ . Partial assignments can therefore be compared by means of subset inclusion. If a partial assignment  $\phi$  is contained in partial assignment  $\phi'$ , then  $\phi$  can be **extended** to  $\phi'$ . The **variables of**  $\phi$  is the projection of  $\phi$  to its first component.

A partial assignment violates a constraint if the projection of some constraint relation, to the indices corresponding to a subset of variables it shares with the partial assignment, excludes the tuple of values that the partial assignment assigns to that set of variables.

**Definition 2.38** (constraint violation). A partial assignment  $\phi$  **violates a constraint**  $(\sigma, R)$ , where both  $\sigma$  and  $R$  are indexed by  $I$ , if  $\phi$  contains some partial assignment  $\phi' \subseteq \phi$  and there is a subset  $\sigma_J \subseteq \sigma$  indexed by  $J \subseteq I$ , such that  $\{(j, \phi'(\sigma_J(j))) \mid j \in J\} \notin R_J$ . If  $\phi$  does not violate any constraints in the instance description, it is said to **respect the constraints**.  $\square$

**Example 2.39** (constraint violation). Consider an instance in variable-value representation

$$(\{u_1, u_2\}, \{0, 1\}, \{((u_1), \{(0), (1)\}), ((u_1, u_2), \{(0, 1)\})\}).$$

The partial assignment  $\{(u_1, 1)\}$  respects the unary constraint but violates the binary constraint, while  $\{(u_1, 0)\}$  respects the constraints. The complete assignment  $\{(u_1, 0), (u_2, 1)\}$  also respects the constraints.  $\square$

Partial assignments either respect or violate the constraints. I now introduce a term for partial assignments that respect the constraints.

**Definition 2.40** (prop). A **prop** or **consistent partial assignment** of an instance is a partial assignment that respects the constraints of that instance. An **instance prop** is a prop with the same variables as those of some constraint.  $\square$

Every subset of a prop also respects the constraints, but a subset of an instance prop is not generally itself an instance prop.

*Observation 2.41.* If  $\phi$  is a prop and  $\phi' \subseteq \phi$  then  $\phi'$  is also a prop. In this situation,  $\phi'$  is only an instance prop if its set of variables coincides with the set of variables of some constraint.

**Definition 2.42** (solution). A **solution** is a prop that is a complete assignment.  $\square$

Let  $\phi: V \rightarrow D$  be a solution. Since  $\phi$  must respect the constraints, whenever  $(\sigma, R)$  is a constraint in  $C$  then

$$(\phi(\sigma_1), \phi(\sigma_2), \dots, \phi(\sigma_{\rho(\sigma)})) \in R.$$

Some partial assignments respect the constraints, but never lead to a solution; when looking for solutions, such partial assignments lead to dead ends.

**Definition 2.43** (nogood). A **nogood** is a partial assignment that cannot be extended to a solution. An **explicit nogood** is a partial assignment that violates some constraint in the instance description. An **implicit nogood** is a nogood which respects the constraints. An **instance nogood** is an explicit nogood with the same variables as those of some constraint.  $\square$

When a partial assignment is an explicit nogood, then so is any partial assignment that contains it. Extensions of instance nogoods are explicit nogoods, but generally not instance nogoods.

As a consequence of the definitions, in a CSP instance with no solutions every partial assignment is a nogood, even those that respect the constraints.

**Example 2.44** (nogoods). Consider the CSP instance in variable-value representation

$$(\{u_1, u_2, u_3\}, \{0, 1\}, \{((u_1), \{(0), (1)\}), ((u_1, u_2), \{(0, 1), (1, 0)\}), ((u_2, u_3), \{(1, 0)\})\}).$$

This has three variables, domain  $\{0, 1\}$ , and three constraints.

The only solution to this instance is the assignment  $\phi_1 = \{(u_1, 0), (u_2, 1), (u_3, 0)\}$ . The partial assignment  $\phi_2 = \{(u_1, 1)\}$  respects the constraints, but it is an implicit nogood. The assignment  $\phi_3 = \{(u_2, 0)\}$  violates the third constraint, so is an explicit nogood, but it does not have the same set of variables as some constraint, so it is not an instance nogood. The assignment  $\phi_4 = \{(u_1, 1), (u_2, 1)\}$  violates the second constraint, but it also has the same set of variables as that constraint, so it is an explicit nogood. It is also an instance nogood.  $\square$

Nogoods are partial assignments that may or may not respect the constraints, but which are not subsets of any solution. Props may turn out to be implicit nogoods during the search for a solution, but props by definition cannot be explicit nogoods.

I have introduced the term *prop* to denote a consistent partial assignment, suggesting it is “proper” in contrast with nogoods. Moreover, unlike a nogood, which once present in a partial assignment ensures that no partial assignment containing it can be consistent, a prop may assist with finding solutions, but may be rendered irrelevant once it has been shown to be a nogood. The name prop suggests its impermanent nature.

### 2.4.3 Domains

The domain of an instance may contain values not used in the constraints. The domain may be large, even if only a few values are actually allowed by the constraints. Such padding of CSP instances may artificially inflate instance size, and this can change the complexity of the CSP. Some care is therefore required.

**Definition 2.45** (active domain). The **active domain**  $\text{dom}(R)$  of a relation  $R$  of arity  $r$  is the set of values that appear in the relation,

$$\text{dom}(R) = \{t(i) \mid t \in R, i \in [r]\}.$$

The active domain of a set of relations  $\Gamma$  combines the active domains of each relation,

$$\text{dom}(\Gamma) = \bigcup \{\text{dom}(R) \mid R \in \Gamma\}.$$

The active domain of a relational structure  $S = (V(S), (Q_i)_{i \in I})$  is the active domain of the language of  $S$ ,

$$\text{dom}(S) = \text{dom}(\Gamma(S));$$

note that  $\text{dom}(S) \subseteq V(S)$ .

Let  $\mathcal{P} = (V, D, C)$  be a CSP instance in variable-value representation. The active domain of variable  $u$  is then

$$\text{dom}(u) = \{t(i) \mid (\sigma, R) \in C, \sigma(i) = u, t \in R\},$$

and the active domain of instance  $\mathcal{P}$  is

$$\text{dom}(\mathcal{P}) = \bigcup \{ \text{dom}(u) \mid u \in V \};$$

note that  $\text{dom}(\mathcal{P}) \subseteq D$ . ┘

The domain  $D$  is commonly the non-negative integers, the set of rational numbers, or a fixed finite set of labels. If  $D$  is finite then a usual requirement is that it should be listed in the instance description in some way, perhaps implicitly as the set of all values that are allowed for any variable by the constraints, but when  $D$  is infinite then this requirement is not enforced. For intensionally specified constraints (when the constraint tuples are not listed), the domain is also often left implicit.

In this thesis I focus on the case where the single common domain of a CSP instance is a finite set. However, I do not require the domain of different instances to be the same, as this would unnecessarily restrict the CSPs under discussion. A variant of CSPs which allows different domains for each variable is discussed in Section 3.1.3.

Next, I consider the second main representation in common usage, based on seeing constraint satisfaction problems as questions of mathematical logic.

#### 2.4.4 First-order logic representation

A CSP instance can be represented by a formula of first-order logic. In this **first-order logic representation**, a constraint  $(\sigma, R) \in C$  of arity  $r$  is regarded as a predicate  $R(u_1, u_2, \dots, u_r)$  involving the variables  $u_1, u_2, \dots, u_r$ , where  $\sigma = (u_1, u_2, \dots, u_r)$ . The collection of constraints is then interpreted as the conjunction of these predicates, which is satisfiable precisely when every constraint is satisfied by some assignment.

If  $a_1, a_2, \dots, a_r$  are values from  $D$ , the interpretation of the predicate  $R(a_1, a_2, \dots, a_r)$  is that the tuple  $(a_1, a_2, \dots, a_r)$  is in the relation  $R$  over  $D$  of arity  $r$ . These predicates are regarded as part of the language of this logic. The instance is then an existentially quantified conjunction, over all constraints, of the predicate corresponding to each constraint.

**Definition 2.46** (instance description: first-order logic). Suppose the variables that occur in the instance are  $u_1, u_2, \dots, u_s$ , and that the values that occur in the instance are from an underlying domain  $D$ . In the first-order logic representation, the instance description is a formula

$$\exists u_1 \exists u_2 \dots \exists u_s \bigwedge_{(\sigma, R) \in C} R(\sigma_1, \sigma_2, \dots, \sigma_{\rho(\sigma)}).$$
┘

This restricted syntactic form of first-order formulas is called **primitive positive**. Primitive positive formulas contain only existential quantifiers, the relational predicates, and conjunctions. They do not contain universal quantifiers, disjunctions, negations, or implications.

In database theory, the constraint relations are often treated as a database. The relational propositions denote membership of a tuple in the corresponding relation in the database, and primitive positive formulas are also known as **conjunctive queries** due to their restricted syntactic form [1, 21].

Database theory explicitly allows the first-order formula to be quantified over a subset of the variables. The unquantified variables in the formula are then used to construct the result of the query. Constraint satisfaction problem instances are those database queries (called **Boolean queries**) where all variables are quantified. A solution to the CSP instance exists if the query yields a relation containing a single empty tuple, or fails to exist if the query yields an empty relation.

**Example 2.47** (INJECTIVE: first-order logic representation). Let  $u_1, u_2, \dots, u_n$  be the variables. The predicate  $R(u_1, u_2, \dots, u_n)$  should be true for all assignments of values to the variables with no value assigned to more than one variable. The interpretation of this predicate can then be defined analogously to Example 2.36 as the relation

$$R = \{(u_1, u_2, \dots, u_n) \in D^n \mid |\{u_1, u_2, \dots, u_n\}| = n\}.$$

Then the formula is

$$\exists u_1 \exists u_2 \dots \exists u_n R(u_1, u_2, \dots, u_n),$$

consisting of the existentially quantified predicate  $R$ . ┘

A solution of the primitive positive formula

$$\exists u_1 \exists u_2 \dots \exists u_s \phi(u_1, u_2, \dots, u_s)$$

is a **satisfying assignment** of values  $a_1, a_2, \dots, a_s$  to the variables  $u_1, u_2, \dots, u_s$ . This means that  $\phi(u_1 := a_1, u_2 := a_2, \dots, u_s := a_s)$  with value  $a_i$  substituted for variable  $u_i$ , must evaluate to true.

I next consider the third classic representation, which focuses on constraint satisfaction problems as algebraic questions.

### 2.4.5 Homomorphism representation

Each CSP instance can also be regarded as an ordered pair of relational structures, by reconsidering how the constraints are specified.

**Definition 2.48** (instance description: homomorphism). In the **homomorphism representation**, the instance description is an ordered pair  $(S, T)$ .

1.  $S$  is a relational structure, called the **source structure**.
2.  $T$  is a relational structure, called the **target structure**.
3. the relations of  $S$  and  $T$  are indexed by the same set. ┘

The **arity** of CSP instance  $(S, T)$  is the arity of  $S$ , i.e. the largest arity of any of its scopes.

I will usually use  $s$  to denote the number of vertices  $|V(S)|$  in the source structure, and  $t$  to denote the number of vertices  $|V(T)|$  in the target structure. The **variables in the instance** are the vertices  $V(S)$  of the source structure, and the **values in the instance** are the vertices  $V(T)$  of the target structure.

Suppose  $Q$  is a relation of arity  $r$  over set  $V$  and  $\phi$  is a function from  $V$  to  $W$ . Then  $\phi$  **applied to relation**  $Q$ , denoted  $\phi(Q)$ , is the relation obtained by applying  $\phi$  componentwise to each tuple in  $Q$ :

$$\phi(Q) = \{(\phi(u_1), \phi(u_2), \dots, \phi(u_r)) \mid (u_1, u_2, \dots, u_r) \in Q\}.$$

**Definition 2.49.** A **relational structure homomorphism** from  $S = (V(S), (Q_i)_{i \in I})$  to  $T = (V(T), (R_i)_{i \in I})$  is a function  $\phi: V(S) \rightarrow V(T)$  so that  $\phi(Q_i) \subseteq R_i$  for every  $i$  in  $I$ .

In the definition of relational structure homomorphism, the condition that  $\phi(Q_i) \subseteq R_i$  for each index  $i$  is informally expressed as  $\phi$  preserving all relations in the structure.

A solution of a CSP instance in the homomorphism representation is a relational structure homomorphism. An instance  $(S, T)$  of the CSP is therefore sometimes also written as  $S \xrightarrow{?} T$  to emphasize that the instance  $(S, T)$  is of the decision problem. I will generally be considering the decision problem version of problems in this work, so I will usually not make this distinction.

*Observation 2.50.* If there is a relational structure homomorphism from  $S$  to  $T$ , then  $S$  and  $T$  must have the same signature.

*Observation 2.51.* By extending the notion of a relational structure homomorphism to partial functions that preserve all relations in the structure in the obvious way (using projections as in Definition 2.38), props are simply partial relational structure homomorphisms.

**Definition 2.52** (hypergraph of CSP instance). The **hypergraph of CSP instance**  $(S, T)$  is the hypergraph  $H(S)$  of the source structure of the instance.  $\square$

The hypergraph of a CSP instance is sometimes called its **constraint network** or **constraint hypergraph**. When the CSP instance is binary, the hypergraph of a CSP instance is just a graph and is also called its **constraint graph**.

**Example 2.53** (INJECTIVE: homomorphism representation). Let  $V(S) = u_1, u_2, \dots, u_n$ , and let  $V(T) = D$ . As in Example 2.36 and Example 2.47, let

$$R = \{(u_1, u_2, \dots, u_n) \in D^n \mid |\{u_1, u_2, \dots, u_n\}| = n\}.$$

Then the homomorphism representation of INJECTIVE is the pair of relational structures

$$((V(S), \{(u_1, u_2, \dots, u_n)\}), (V(T), R)).$$

These structures both contain just one relation. Note that the source structure in this case also only contains just one tuple, identical to the scope of the single constraint. The hypergraph of this instance has vertices  $V(S)$  and a single edge  $\{u_1, u_2, \dots, u_n\}$ .  $\square$

## 2.5 Historical perspective

The notion of an ordered pair is due to Kuratowski [109].

The *decision problem* classically consisted of finding a procedure that decides for every formula of predicate logic whether it is universally valid, and which does so in finitely many steps. By linking formulas with notions of what a decision procedure can achieve, and demonstrating the existence of uncomputable sets, the classical decision problem was shown not to have a solution by Turing [149].

The focus then shifted to more restricted decision problems, where the set of formulas is limited. This can be rephrased as deciding membership of a given string in a set of strings. The terminology *discrete combinatorial decision problem* was explicitly used by Post to refer to such problems [130]. The many different computation models for these problems are all equivalent by straightforward reductions [145]. Combinatorial decision problems are therefore believed to constitute precisely those that can be defined by a computer. This is now known as the Church-Turing thesis [101, Thesis I].

In an influential early paper, Golomb and Baumert formalized the tree structure of combinatorial search, and discussed many key ideas, such as considering the search space as a product of finite domains, the importance of symmetry of the search space, and solving with respect to a criterion function to capture desirability of different solutions. Their main technical contribution was showing how “inherently suboptimal” partial assignments allow removing from the search tree any assignments that contain these suboptimal partial assignments [67]. For criterion functions with only two values (acceptable and not acceptable), these suboptimal partial assignments were called *nogoods* by Stallman and Sussman [146].

Constraint satisfaction was formalized with its own notation by Montanari in a 1971 technical report, published in 1974 [122]. This focused on constraint networks of binary CSPs. A constraint network corresponds to the variable-value representation, with the variables as vertices of a hypergraph, a constraint relation specified as the label on the edge representing a scope, and the domain implicitly specified as the active domain of the instance.

Using the database terminology of conjunctive queries, Chandra and Merlin noted the NP-completeness of the language consisting of all those CSP instances which have a solution [25, Theorems 7 and 11]. The proof of their Lemma 13 also demonstrates that the CSP instance  $(S, T)$  in the homomorphism representation has a solution precisely when all solutions of  $(T, U)$  are also solutions of  $(S, U)$ , for every relational structure  $U$ . This established the theoretical foundation for optimization of conjunctive queries, by allowing

containment between such queries to be studied via the associated CSP instance.

Tsang was one of the first in the constraint satisfaction community to explicitly discuss constraints as relations described by formulas of first-order logic [148]. Jeavons and Cooper, following Tsang, standardised the variable-value representation and the names *scope* and *constraint relation* [96].

The notion of relational structure homomorphism is implicit in many mathematical papers that deal with constraint satisfaction problems, but it and the homomorphism representation were only explicitly introduced into the study constraint satisfaction by Feder and Vardi [52, 53], and independently by Jeavons [95].

Partial assignments have also been called *instantiations* [54, 14]. A consistent partial assignment has been referred to as a *compound label* [148] or as a *locally consistent instantiation* [14]; in this thesis I use the more suggestive name *prop*.

Partial assignments that can be extended to solutions have also been called *globally consistent instantiations* [14].

The term *nogood* is standard [140]. It was introduced as early as 1976, in the context of constraint satisfaction for circuit analysis, involving for instance expressions relating resistances of different components in the circuit [146].

For some CSPs, efficient algorithms exist for structural reasons; this is discussed in Section 3.2.2. A sequence of papers has shown how tree-like structure can be exploited to obtain efficient algorithms [55, 56, 48, 70, 71, 75, 117]. Increasingly sophisticated notions of width are used in these papers. These measures capture for each instance how far its structure is from a tree. Width close to 0 is a structure that is “close” to a tree, while large width corresponds to a structure that is “far” from being a tree. Providing a counterpoint, it has been shown by Grohe that the underlying explanation for the tractability of all bounded-arity CSPs defined by structural restrictions is structure that is close to tree-like [74]. This result shows that unbounded treewidth allows a reduction of NP-hard problems to CSPs, making use of the grid-like structures of arbitrarily large size that are guaranteed to exist in classes of bounded-arity structures of unbounded treewidth.

For CSPs defined by bounded width measures beyond treewidth, it remains an open problem whether a width measure exists that captures the precise boundary between CSPs for which a polynomial-time algorithm exists, and those that are NP-hard. The existence of a boundary between fixed-parameter tractable cases, and those that are not fixed-parameter tractable, has been shown to follow from a plausible assumption in complexity theory, the Exponential Time Hypothesis [118]. (Note that the Exponential Time Hypothesis implies  $P \neq NP$  [91].)

CSPs can also have efficient algorithms due to restrictions on the set of constraint relations, as discussed in Section 3.2.3. Loosely speaking, if the algebraic structure of the relations used in a CSP contains a factor that cannot be further simplified, then this is known to lead to NP-completeness. It was further conjectured that every other situation leads to a polynomial-time algorithm [22].



The open question for classes of CSPs where only the set of relations is fixed, is to demonstrate that there always exists a polynomial-time algorithm whenever the algebraic structure contains no problematic factor. Several of the known tractable cases have been unified, along lines previously conjectured [13, 9, 10].

The variable-value and homomorphism representations are *extensional*, since the tuples in each relation are listed explicitly in an instance, and all input relations are counted as part of the input size. It is also possible to consider *intensional* representations of constraints, where the allowed or disallowed partial assignments are represented by means of an oracle that can answer queries of the form “is this partial assignment allowed by the constraint”? I do not consider intensional representations in this thesis.

CSPs in which instances may be specified by means of predefined predicates in the language, defined over an infinite set of values, have been studied extensively since Bodirsky introduced their study in his thesis [17]. However, most work on CSPs (including this thesis) continues to assume a finite domain for each instance.

Fixing the signature for all CSP instances in the class under consideration is a common assumption, but in this thesis I do not require this assumption. Different instances of a CSP may have different signatures, unless they have explicitly been specified to be the same. All instances of VICTORIAN LETTERS and DIRECTED GRAPH ACYCLICITY have the same signature. In contrast, instances of INJECTIVE generally have different signatures, and therefore LATIN SQUARE COMPLETION, SUDOKU, and FUTOSHIKI also have instances with different signatures.

## 2.6 Chapter summary

In this chapter I discussed the background material needed for this thesis and covered several standard examples of constraints. The definitions in this chapter are mostly standard. However, Section 2.4.2 and Section 2.4.3 also introduced some new terminology (the notion of props, and active domains). I reviewed the three standard representations that have been used in the literature. Finally, I also provided a high level survey of some of the key milestones in the literature that inform my work in this thesis.

*Whatever may be the true nature of things and of the conceptions which we have of them (into which points we are not here concerned to inquire), in the operations of reasoning they are dealt with as a number of separate entities or units.*

—A. B. Kempe, *A Memoir on the Theory of Mathematical Form*, 1886. §3.

# 3

## Concepts

In this chapter I introduce concepts on which the thesis rests.

I first introduce the infrastructure of a CSP instance, and use it to define a notion of equivalence between instances. I then consider combinations of constraints, which will be useful later. Adding unary constraints to an instance corresponds to a variant of CSPs which allows different domains for each variable, and I discuss these two notions next. This sets the scene to compare my notion of instance equivalence to other notions of equivalence in the literature.

I then examine how classes of CSP instances have been defined by means of restrictions of either structure or of language, and what is known of hybrid CSPs that cannot be so defined. Most known tractable classes of CSPs have been defined by means of such restrictions, and I discuss this work.

Finally, I outline transformations between the three basic representations of CSP instances.

### 3.1 Properties of constraints

I now introduce the notion of infrastructure, and then present some of the consequences of the definitions in Chapter 2.

#### 3.1.1 Infrastructure and equivalence

A CSP instance can be regarded as a description of the set of those partial assignments that are consistent with the constraints, as well as those that conflict with the constraints.

These sets are generally of exponential size in the number of variables and the sizes of the domains, so CSP instances form a compact representation of such sets. I now formalize this notion.

**Definition 3.1** (infrastructure). The **infrastructure** of a CSP instance  $\mathcal{P}$  contains the set of all props of  $\mathcal{P}$ , denoted  $\text{prop}(\mathcal{P})$ .  $\square$

*Observation 3.2.* The infrastructure of a CSP instance is downward-closed, i.e. if  $\phi$  is a prop of the instance then so is any  $\phi' \subseteq \phi$ .

Recall that  $s$  denotes the number of variables and  $t$  the number of domain values in a CSP instance. Most of the props of a CSP instance are not explicitly specified in the instance, and  $\text{prop}(\mathcal{P})$  will have up to  $\sum_{i=0}^s \binom{s}{i} t^i = (t+1)^s$  elements in the worst case, which is usually exponential in the size of the instance. Generally the infrastructure requires too much space to store, as well as a large amount of time to manipulate, and therefore it is not generally feasible to compare and manipulate the infrastructures directly. However, the infrastructure serves as a convenient notion to compare constraint satisfaction problem instances.

**Definition 3.3** (subproblem). A CSP instance  $\mathcal{P}_1$  is a **subproblem** of instance  $\mathcal{P}_2$  when the infrastructure of  $\mathcal{P}_1$  is a subset of the infrastructure of  $\mathcal{P}_2$ .  $\square$

Definition 3.3 requires the variables of the subproblem  $\mathcal{P}_1$  to form a subset of the variables of  $\mathcal{P}_2$ , and the active domain of every variable of  $\mathcal{P}_1$  to be a subset of the active domain of the same variable of  $\mathcal{P}_2$ . This definition of subproblem yields a preorder on instances. The equivalence classes of this preorder are especially useful.

**Definition 3.4** (equivalent CSP instances). A CSP instance  $\mathcal{P}_1$  is **infrastructure-equivalent** to instance  $\mathcal{P}_2$  (denoted  $\mathcal{P}_1 \equiv \mathcal{P}_2$ ) when the infrastructures of  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are the same.  $\square$

In Section 3.1.4 I will compare this notion of instance equivalence to other similar notions in the literature.

**Example 3.5.** Consider the CSP instances  $\mathcal{P}_1 = (\{u\}, \{0\}, \{(u), \{(0)\}\})$  and  $\mathcal{P}_2 = (\{u\}, \{0, 1\}, \{(u), \{(0)\}\})$ . These both have a single prop  $\{(u, 0)\}$ , so  $\mathcal{P}_1 \equiv \mathcal{P}_2$ .  $\square$

Two instances can only be equivalent when they have the same variables. Equivalent instances must have the same variables and domains after any unused domain values have been removed. The following definition helps to clarify this.

**Definition 3.6** (1-consistency). A CSP instance is **1-consistent** if  $D_u = \text{dom}(u)$  for every variable  $u$ .  $\square$

In other words, 1-consistency ensures that every domain value actually is allowed by some constraint. This captures the notion that the domains of a CSP instance should not extend beyond the values appearing in the constraints.

**Definition 3.7** (support). A domain value  $a$  for variable  $u$  is **supported** by a constraint  $(\sigma, R)$  if  $u = \sigma(i)$  for some  $i$  and  $(a) \in R_{\{i\}}$ .  $\square$

Put differently, a value  $a$  for a variable  $u$  is supported by a constraint when there is a partial assignment  $\phi$  that respects the constraint, and which assigns  $a$  to  $u$ .

Establishing 1-consistency involves removing values from the domains of each variable until each value for that variable is supported by every constraint. As discussed in Section 3.1.3, this can be achieved by adding unary constraints to the instance that forbid unsupported values, or by changing the domains associated with the variables. These changes do not affect the infrastructure and therefore lead to infrastructure-equivalent instances.

**Corollary 3.8.** *If CSP instances  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are infrastructure-equivalent then they will have the same set of variables and the same domains after 1-consistency has been established.*

It is possible for two CSP instances to be infrastructure-equivalent except that one instance contains a domain value for one variable that is not supported by any constraint, while the other instance has no such superfluous domain value. Establishing 1-consistency removes such superficial differences.

In Section 3.1.2 I use the notion of infrastructure-equivalence between instances to derive some simple but widely useful results about combinations of constraints.

### 3.1.2 Combining constraints

A CSP instance contains some number of constraints. Usually a CSP instance is built by combining multiple constraints. I will simply observe that it is non-trivial to predict the behaviour of the CSP instance that is obtained by combining even quite simple individual constraints, but that it is possible to understand how a few common constraints behave when they are combined.

In the variable-value representation, if  $C$  is a set of constraints consisting of the union of sets  $C_1$  and  $C_2$  of constraints, then the props of  $(V, D, C)$  are precisely those that are props of both  $(V, D, C_1)$  and  $(V, D, C_2)$ .

**Proposition 3.9.**  $prop(V, D, C_1 \cup C_2) = prop(V, D, C_1) \cap prop(V, D, C_2)$ .

*Proof.* Suppose  $\phi$  is a prop of  $(V, D, C_1 \cup C_2)$ . Then  $\phi$  is consistent with every constraint in  $C_1$  as well as with every constraint in  $C_2$ . Hence  $prop(V, D, C_1 \cup C_2) \subseteq prop(V, D, C_1) \cap prop(V, D, C_2)$ . For the reverse implication, suppose  $\phi$  is a prop of each of  $(V, D, C_1)$  and  $(V, D, C_2)$ . This clearly implies that  $\phi$  is a prop of  $(V, D, C)$ .  $\square$

Proposition 3.9 is not as practically useful as it might first appear. The large size of the infrastructure makes it prohibitive to work directly with it, and instead approximations to the infrastructure are stored by constraint solvers. In these approximations analogues of Proposition 3.9 may not hold.

The building blocks of CSP instances are the individual constraints. Recall that the UNARY constraint was introduced in Example 2.28; this is one of the most useful such building blocks.

**Lemma 3.10.**  $\text{UNARY}(u; D_1 \cap D_2) \equiv \{\text{UNARY}(u; D_1), \text{UNARY}(u; D_2)\}$ .

*Proof.* First observe that  $\{(u, a)\} \in \text{prop}(\text{UNARY}(u; D))$  holds precisely when  $a \in D$ , so  $\sigma \in \text{prop}(\text{UNARY}(u; D_1 \cap D_2)) \Leftrightarrow \sigma(u) \in D_1 \cap D_2 \Leftrightarrow \sigma \in \text{prop}(\text{UNARY}(u; D_1)) \cap \text{prop}(\text{UNARY}(u; D_2))$ . Hence  $\text{UNARY}(u; D_1 \cap D_2) \equiv \{\text{UNARY}(u; D_1), \text{UNARY}(u; D_2)\}$ .  $\square$

It is possible to express constraints in different ways. Since a constraint satisfaction problem instance contains a set of constraints, there may be instances with different sets of constraints, but where the solutions are the same.

**Proposition 3.11.** *Let  $\bar{u}$  denote the list  $u_1, u_2, \dots, u_s$ . Then*

$$\text{CONSTRAINT}(\bar{u}; D_1 \cap D_2) \equiv \{\text{CONSTRAINT}(\bar{u}; D_1), \text{CONSTRAINT}(\bar{u}; D_2)\}.$$

*Proof.* Clearly if  $D \subseteq E$ , then

$$\begin{aligned} \text{CONSTRAINT}(\bar{u}; D) &\equiv \{\text{CONSTRAINT}(\bar{u}; E), \\ &\quad \text{UNARY}(u_1; D), \text{UNARY}(u_2; D), \dots, \text{UNARY}(u_s; D)\}. \end{aligned}$$

Now let  $E = D_1 \cup D_2$ . Then

$$\begin{aligned} \text{CONSTRAINT}(\bar{u}; D_1 \cap D_2) &\equiv \{\text{CONSTRAINT}(\bar{u}; E), \\ &\quad \text{UNARY}(u_1; D_1 \cap D_2), \dots, \text{UNARY}(u_s; D_1 \cap D_2)\}. \end{aligned}$$

By Lemma 3.10, this is equivalent to

$$\begin{aligned} &\{\text{CONSTRAINT}(\bar{u}; E), \text{UNARY}(u_1; D_1), \dots, \text{UNARY}(u_s; D_1), \\ &\quad \text{UNARY}(u_1; D_2), \dots, \text{UNARY}(u_s; D_2)\}, \end{aligned}$$

and the result follows.  $\square$

Any constraint can also be expressed as a constraint with respect to a suitable set  $D$ , together with additional UNARY constraints on each variable.

**Proposition 3.12.** *Let  $D = D_1 \cup D_2 \cup \dots \cup D_s$ . Then*

$$\begin{aligned} \text{CONSTRAINT}(u_1, u_2, \dots, u_s; D_1, D_2, \dots, D_s) &\equiv \{\text{CONSTRAINT}(u_1, u_2, \dots, u_s; D), \\ &\quad \text{UNARY}(u_1; D_1), \\ &\quad \text{UNARY}(u_2; D_2), \\ &\quad \dots \\ &\quad \text{UNARY}(u_s; D_s)\}. \end{aligned}$$

*Proof.* By using  $D_i = D_i \cap D$  and working analogously to the proof of Proposition 3.11, it can be shown that  $\text{CONSTRAINT}(u_1, u_2, \dots, u_s; D_1, D_2, \dots, D_s)$  is equivalent to

$$\{\text{CONSTRAINT}(u_1, u_2, \dots, u_s; D, D_2, \dots, D_s), \text{UNARY}(u_1; D_1)\}.$$

Iterating this for each  $i = 2, 3, \dots, s$  yields the desired result.  $\square$

A useful consequence of Proposition 3.11 and Proposition 3.12 shows that additional unary constraints can always be added to an instance in a reasonable way.

**Corollary 3.13.**

$$\begin{aligned} &\{\text{CONSTRAINT}(u_1, u_2, \dots, u_s; D_1, D_2, \dots, D_s), \text{UNARY}(u_i; D'_i)\} \\ &\equiv \text{CONSTRAINT}(u_1, u_2, \dots, u_s; D_1, \dots, D_{i-1}, D_i \cap D'_i, D_{i+1}, \dots, D_s). \end{aligned}$$

Recall that the INJECTIVE constraint was introduced in Example 2.29. This is another constraint that is commonly combined with other constraints. In isolation, the INJECTIVE constraint has a solution precisely when  $D$  contains at least as many different values as there are variables.

**Proposition 3.14.** INJECTIVE( $u_1, u_2, \dots, u_s; D$ ) has a solution if, and only if,  $|D| \geq |\{u_1, u_2, \dots, u_s\}|$ .

*Proof.* If there is a solution then there is an injective function from  $\{u_1, u_2, \dots, u_s\}$  to  $D$ , so  $|D| \geq |\{u_1, u_2, \dots, u_s\}|$ . For the converse, if  $|D| \geq |\{u_1, u_2, \dots, u_s\}|$  then there is an injection  $\phi: \{u_1, u_2, \dots, u_s\} \rightarrow D$  which is then a solution.  $\square$

I discuss combinations of INJECTIVE and UNARY constraints in Section 3.1.3.

### 3.1.3 More about domains

A variant of the variable-value representation allows the domains of each variable to be different, so that there is a domain  $D_u$  specified for each variable  $u \in V$  [24]. The relation corresponding to scope  $\sigma$  is then not a subset of  $D^{\rho(\sigma)}$  but of the product

$$\prod_{i=1}^{\rho(\sigma)} D_{\sigma(i)}.$$

For the variant with different domains, the requirement that solutions respect constraints is the same, but the solution is no longer a function  $V \rightarrow D$  but instead a function

$$\phi: V \rightarrow \bigcup_{u \in V} D_{\sigma(i)}$$

such that  $\phi(u) \in D_u$  for every  $u \in V$ .

Specifying a different domain for each variable is not, strictly speaking, necessary. The domains do not need to be explicitly specified, as they can be recovered from the constraints by setting  $D_u = \text{dom}(u)$  for each variable  $u$ , using the notion of active domain in Definition 2.45, or they can be added explicitly as unary constraints  $\text{UNARY}(u; D_u)$ . Proposition 3.12 shows that  $\text{UNARY}$  constraints can be used to capture different domains for each variable in an instance either directly, or by using  $\text{UNARY}$  constraints. It is therefore possible to either work with the representation that keeps different domains for each variable, or with a representation that keeps track of changes to the domains by means of separate unary constraints.

Allowing different domains corresponds to constraint problems found in practice, and relates closely to the implementation choices made by standard constraint solvers. However, the additional precision of allowing different domains is usually not necessary for theoretical analysis of constraint satisfaction, and a single common domain has generally been preferred in the literature for notational convenience [32, 33, 31, 147].

The following constraint expresses a requirement that every variable takes a different value. The possibility that different variables have different sets of values makes this constraint non-trivial.

**Example 3.15** ( $\text{ALL-DIFFERENT}$ ). The constraint

$$\text{ALL-DIFFERENT}(u_1, u_2, \dots, u_n; D_1, D_2, \dots, D_n)$$

on variables  $u_1, u_2, \dots, u_n$  and sets of values  $D_1, D_2, \dots, D_n$  is satisfied by assignment  $\phi$  if  $\phi$  assigns values  $a_i \in D_i$  to  $u_i$  (for each  $i = 1, 2, \dots, n$ ), so that no value is repeated.  $\square$

Note that  $\text{INJECTIVE}(u_1, u_2, \dots, u_n; D)$  is the special case of  $\text{ALL-DIFFERENT}$  where all the domains are  $D$ .

*Observation 3.16.*

$$\text{INJECTIVE}(u_1, u_2, \dots, u_n; D) = \text{ALL-DIFFERENT}(u_1, u_2, \dots, u_n; D, D, \dots, D).$$

The following then immediately follows from Observation 3.16 and Proposition 3.12.

**Corollary 3.17.** *Let  $D = D_1 \cup D_2 \cup \dots \cup D_n$ . Then*

$$\begin{aligned} \text{ALL-DIFFERENT}(u_1, u_2, \dots, u_n; D_1, D_2, \dots, D_n) \quad \equiv \quad & \{ \text{INJECTIVE}(u_1, u_2, \dots, u_n; D), \\ & \text{UNARY}(u_1; D_1), \\ & \text{UNARY}(u_2; D_2), \\ & \dots \\ & \text{UNARY}(u_n; D_n) \}. \end{aligned}$$

In principle, it appears possible to modify the definition of relational structures to allow a version of the homomorphism representation that corresponds to the version of the

variable-value representation with different domains. The same effect can be obtained by using a single set of vertices, and instead simply adding additional unary constraints to the signature to represent the different domains. This approach was considered previously by Bulatov and Jeavons, who show that it is reasonable to consider only relational structures where all relations are defined over the same set of vertices [24, Proposition 1].

On the one hand, if  $D_u = D$  for every variable  $u$ , the single domain variant of the variable-value representation is obtained as a special case. On the other hand, letting  $D = \text{dom}(\mathcal{P})$  and adding unary constraints  $\text{UNARY}(u; D_u)$  results in an infrastructure-equivalent instance with additional constraints  $\{\text{UNARY}(u; D_u) \mid u \in V\}$ . I will therefore often assume a single common domain in this thesis; this is also a common assumption in the literature.

### 3.1.4 Different notions of instance equivalence

**Nogood equivalence** Definition 3.4 of infrastructure-equivalence is an equivalence relation. It differs from Bessiere’s notion of nogood-equivalence [14, Definition 3.11], which is a preorder defined by comparing the sets of nogoods. The subproblem preorder of Definition 3.3 is also slightly more general than the preorder used by Bessiere based on nogoods, which requires that the sets of variables be the same. I only require a subset relation.

The difference between the two notions appears most starkly when considering instances that have no solutions. Any two such instances will have the same set of nogoods (being all possible partial assignments), while their infrastructures may be far from similar. The following example illustrates this.

**Example 3.18.** Consider two instances  $\mathcal{P}_1$  and  $\mathcal{P}_2$ , both with domain  $D = [t]$  and set of variables  $V = [s]$ , such that  $t = s - 1$ . Suppose instance  $\mathcal{P}_1$  has an ALL-DIFFERENT constraint, while instance  $\mathcal{P}_2$  has an ALL-DIFFERENT constraint but also a constraint that requires all variables to be set to the value 1 in a solution. In  $\mathcal{P}_1$ , there are props of every cardinality up to  $t$ , while in  $\mathcal{P}_2$ , the only props are all unary partial assignments to 1. However,  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are nogood-equivalent as neither has any solutions.  $\square$

Note that whenever two instances are infrastructure-equivalent, then they are also nogood-equivalent. Nogood-equivalence is therefore a coarser notion of equivalence. The complexity of nogood-equivalence is also at least as high as that of infrastructure-equivalence.

**Proposition 3.19.** *Nogood-equivalence is co-NP-hard and in  $\Sigma_2^P$ . Infrastructure equivalence is in co-NP.*

*Proof.* Nogood-equivalence is in  $\Sigma_2^P$ , since an alternating Turing machine can first in its existential state guess a partial assignment that is a nogood in the one instance but not the other, and then in its universal state verify it being a nogood by checking whether each of the complete assignments containing it respects the constraints.



The NP-complete problem of deciding whether a CSP has a solution, can be many-one logspace reduced to the problem of deciding nogood-nonequivalence. The reduction works by adding all unary nogoods to the instance, thereby ensuring it has no solutions; this instance is nogood-equivalent to the original instance precisely when the original instance does not have a solution. Hence nogood-nonequivalence is NP-hard and therefore nogood-equivalence is co-NP-hard.

Infrastructure-nonequivalence can be decided by exhibiting a partial assignment that can be checked against the constraints in polynomial time. Hence infrastructure-equivalence is in co-NP.  $\square$

**Mutual reducibility** Another coarser definition of equivalence of instances was investigated by Rossi et al. by defining two instances as equivalent when they are mutually reducible to each other [139]. This highly nontrivial notion of reducibility requires a mapping from variables in one instance and tuple expressions built from the variables in the other, and a second mapping involving tuple expressions over sets of parameters that range over the domains of the two instances. The syntactic form of mutual reducibility is so general that the complexity of deciding mutual reducibility seems likely to be high. In its original form, it can capture equivalence between a CSP instance and the result of applying the dual transformation that is discussed in Section 5.1.4, but I will not deal with the details of equivalence via mutual reducibility.

It may be worth while to reconsider the notion of mutual reducibility, restricting its generality to more easily allow analysis of its complexity.

Definition 3.4 has the significant advantage over mutual reducibility that it is purely syntactic. Whenever  $\mathcal{P}_1$  is equivalent to  $\mathcal{P}_2$  by Definition 3.4, then they are also equivalent by the mutual reducibility definition. The converse is not necessarily true, since mutual reducibility can equate instances which involve relational structures with different signatures, such as a non-binary instance  $\mathcal{P}$  and the result of applying the dual transformation to  $\mathcal{P}$  (which is binary).

**Same-solution equivalence** Another coarser definition of equivalence of instances is used in the context of database theory. Two conjunctive queries (CSP instances in the first-order logic representation) are said to be equivalent when they have the same solutions. In contrast, Definition 3.4 distinguishes between equivalent conjunctive queries with different infrastructures. When two instances have the same infrastructures they have the same solutions, and are therefore equivalent in the conjunctive query sense. Note that mutual reducibility and having the same solutions are not in general comparable as relations between instances.

All of the different notions of equivalence discussed in this section are illustrated in the following example.

**Example 3.20** (instance equivalences). Let  $\mathcal{P}_1$  be the CSP instance in variable-value

representation with variables  $V = \{u, v\}$ , domain  $D = \{0, 1\}$ , and a set of constraints containing just a single constraint  $C = \{((u, v), \{(0, 0)\})\}$ . Let  $\mathcal{P}_2$  have the same set of variables  $V$  and the same domain  $D$ , but constraints  $C \cup \{((u), \{(0)\})\}$ . Let  $\mathcal{P}_3$  have the same set of variables and domain, with constraints  $C \cup \{((v), \{(0)\})\}$ . Finally, let  $\mathcal{P}_4$  have the same variables and constraints as  $\mathcal{P}_1$ , but with domain  $\{0, 1, 2\}$ . The infrastructures of  $\mathcal{P}_1$ ,  $\mathcal{P}_2$ , and  $\mathcal{P}_3$  are all different, yet these instances all have the same solution.  $\mathcal{P}_4$  has the same infrastructure as  $\mathcal{P}_1$ , but these instances differ in terms of Bessiere's nogood-equivalence due to the unused domain value 2 in  $\mathcal{P}_4$ . Further,  $\mathcal{P}_2$  and  $\mathcal{P}_3$  are mutually reducible but  $\mathcal{P}_1$  is not reducible to  $\mathcal{P}_2$  or  $\mathcal{P}_3$ . Finally,  $\mathcal{P}_1$  is a subproblem of both  $\mathcal{P}_2$  and  $\mathcal{P}_3$ , but  $\mathcal{P}_2$  and  $\mathcal{P}_3$  are not subproblems of each other.  $\square$

In the following I will use Definition 3.3 of subproblems based on the infrastructure, and the associated infrastructure-equivalence of Definition 3.1, to compare problems.

## 3.2 Tractable CSPs

An important question in constraint satisfaction is: *which classes of CSPs are tractable?*

**Definition 3.21** (tractability). A constraint satisfaction problem is **tractable** if it is decidable in polynomial time.  $\square$

By restricting the way a class of CSP instances is specified, it becomes possible to construct CSPs that are tractable. In Chapter 2, I explicitly excluded from consideration trivially tractable CSPs that contain only a finite number of instances, or that contain all but a finite number of instances. However, there are many CSPs that are tractable for more interesting reasons.

Two main reasons for tractability have been investigated, corresponding to restrictions of the structure or the language. Some previous work also considered CSPs formed by means of hybrid restrictions, combining structural and language restrictions.

### 3.2.1 CSPs from restrictions

A broad class of CSPs can be defined by choosing classes  $\mathfrak{A}$  and  $\mathfrak{B}$  of relational structures. A CSP is then defined as  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  in terms of these classes of structures. A CSP instance is a member of  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  if there are some  $S \in \mathfrak{A}$  and  $T \in \mathfrak{B}$  such that the instance is the question of whether there exists a homomorphism from  $S$  to  $T$ . This defines infinitely many CSPs, parameterized by the choice of classes  $\mathfrak{A}$  and  $\mathfrak{B}$ .

**Definition 3.22** ( $\text{CSP}(\mathfrak{A}, \mathfrak{B})$ ). Suppose  $\mathfrak{A}$  and  $\mathfrak{B}$  are classes of relational structures. The constraint satisfaction problem  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  consists of all CSP instances  $(S, T)$  in the homomorphism representation such that the source structure  $S$  is in class  $\mathfrak{A}$  and the target structure  $T$  is in class  $\mathfrak{B}$ .

$\text{CSP}(\mathfrak{A}, \mathfrak{B})$

Input: relational structures  $S \in \mathfrak{A}$ ,  $T \in \mathfrak{B}$

Question: does there exist a relational structure homomorphism from  $S$  to  $T$ ?

When  $\mathfrak{A}$  or  $\mathfrak{B}$  are singletons, I will use  $\text{CSP}(S, \mathfrak{B})$  to denote  $\text{CSP}(\{S\}, \mathfrak{B})$  and  $\text{CSP}(\mathfrak{A}, T)$  to denote  $\text{CSP}(\mathfrak{A}, \{T\})$ .  $\square$

Although the usual treatment of  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  considers only classes defined by some fixed signature, Definition 3.22 can also be applied to classes of structures which do not have a fixed signature. Given an instance  $(S, T)$  from  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$ , when  $S$  and  $T$  do not have the same signature then there is no homomorphism from  $S$  to  $T$ , so it is possible to reject the instance immediately.

If  $\mathfrak{A}$  and  $\mathfrak{B}$  both contain finitely many structures, then  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  contains finitely many instances. When considering problems of the form  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$ , I will therefore assume that either  $\mathfrak{A}$  is infinite, or  $\mathfrak{B}$  is infinite, or that both  $\mathfrak{A}$  and  $\mathfrak{B}$  are infinite.

The CSP consisting of all constraint satisfaction instances in the homomorphism representation is then sometimes denoted  $\text{CSP}(\_, \_)$ . The underscore denotes “unrestricted”, although sometimes additional implicit restrictions are intended, such as restrictions on the signatures in the CSP, or a single finite domain of values used for all instances. Restricting either the structure or the language yields the following two types of CSPs that have both been studied extensively.

**Definition 3.23** (restriction).

- A **structural restriction** is a class  $\mathfrak{A}$  of relational structures corresponding to the class of instances  $\text{CSP}(\mathfrak{A}, \_)$ , which is the **structural class** defined by  $\mathfrak{A}$ .
- A **language restriction** is a class  $\mathfrak{B}$  of relational structures corresponding to the class of instances  $\text{CSP}(\_, \mathfrak{B})$ , which is the **language class** defined by  $\mathfrak{B}$ .  $\square$

**Definition 3.24** (equality relation). The binary **equality** relation  $=_D$  over domain  $D$  relates all pairs of values where both components are identical. Formally,  $=_D$  is the set  $\{(x, x) \mid x \in D\}$ . When the domain is clear,  $=_D$  is simply written  $=$ .  $\square$

**Definition 3.25** (disequality relation). The binary **disequality** relation  $\neq_D$  over domain  $D$  relates pairs of values that are different. Formally,  $\neq_D$  is the relation  $\{(x, y) \mid x, y \in D; x \neq y\} = (D \times D) \setminus (=_D)$ . When the domain is clear,  $\neq_D$  is simply written  $\neq$ .

Let  $\mathfrak{N}$  be the class of relational structures with a single binary disequality relation  $\neq$ .  $\mathfrak{N}$  can then also be thought of as the set of complete graphs  $\{K_s \mid s \in \mathbb{N}\}$ , since by using the disequality relation, the complete graph  $K_s = ([s], \binom{[s]}{2})$  can be expressed as a relational structure as  $([s], \neq)$ .

**Example 3.26** (CLIQUE as CSP).  $\text{CSP}(\mathfrak{N}, \_)$  contains all CSP instances where the source structure is a complete graph, and the target structure is any graph.  $\text{CSP}(\mathfrak{N}, \_)$  is log-space equivalent to the problem CLIQUE in Example 2.19; an instance  $(K_s, G)$  of the former problem corresponds to an instance  $(s, G)$  of the latter.  $\square$

**Example 3.27** (CHROMATIC NUMBER as CSP).  $\text{CSP}(\_, \mathfrak{N})$  contains all CSP instances where the target structure is a complete graph, and the source structure is any graph.  $\text{CSP}(\_, \mathfrak{N})$  is log-space equivalent to the problem CHROMATIC NUMBER in Example 2.20. The number of colours  $t$  in the CHROMATIC NUMBER instance determines the target relational structure  $K_t$  of the CSP instance. The input graph in the CHROMATIC NUMBER instance is the source relational structure in the CSP instance.  $\square$

**Example 3.28** (GRAPH  $t$ -COLOURING as CSP). The number of colours in CHROMATIC NUMBER can be fixed in advance. This yields a different problem, log-space equivalent to  $\text{CSP}(\_, K_t)$ . The problem is NP-complete for any fixed  $t \geq 3$  but can be decided in polynomial time for  $t = 2$  [82].

GRAPH  $t$ -COLOURING

Input: graph  $\mathcal{G}$

Question: can each of the vertices of  $\mathcal{G}$  be assigned one of  $t$  colours, so that the vertices of every edge of  $\mathcal{G}$  are assigned different colours?

The following follows immediately from the definitions.

*Observation 3.29.*  $\text{CSP}(\mathfrak{A}, \mathfrak{B}) = \text{CSP}(\mathfrak{A}, \_) \cap \text{CSP}(\_, \mathfrak{B})$

Observation 3.29 can be useful if  $\text{CSP}(\mathfrak{A}, \_)$  or  $\text{CSP}(\_, \mathfrak{B})$  is tractable, since then  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  is also tractable. Unfortunately it does not provide any information about the tractability of  $\text{CSP}(\mathfrak{A}, \mathfrak{B})$  when both  $\text{CSP}(\mathfrak{A}, \_)$  and  $\text{CSP}(\_, \mathfrak{B})$  are intractable. The following example illustrates an extreme version, where the intersection has low time complexity even though both single restrictions are NP-complete.

**Example 3.30** (INJECTIVE via restrictions). An instance of  $\text{CSP}(\mathfrak{N}, \mathfrak{N})$  requires deciding if a complete graph can be homomorphically mapped into another complete graph. The problem  $\text{CSP}(\mathfrak{N}, \mathfrak{N})$  is log-space equivalent to the INJECTIVE constraint in Example 2.29; an instance  $(K_s, K_t)$  of the former corresponds to an instance  $\text{INJECTIVE}(u_1, u_2, \dots, u_s; [t])$  of the latter.

Both  $\text{CSP}(\mathfrak{N}, \_)$  and  $\text{CSP}(\_, \mathfrak{N})$  are NP-complete. In contrast, by Proposition 3.14  $\text{CSP}(\mathfrak{N}, \mathfrak{N})$  can be solved in linear time, by comparing the number of vertices  $s$  in the source structure and the number of vertices  $t$  in the target structure.  $\square$

### 3.2.2 Reasons for tractability: structure

When constraints only interact in limited ways, there is scope to exploit these limitations to design efficient algorithms. In the extreme case, if variables are completely unrelated by

any constraints, then they can take any values in their domains, and this can be established in linear time. At the other extreme, it is possible that every variable interacts with every other variable by means of a constraint involving both of them, so they limit each other's possible values.

To discuss the structural classes, some terminology is needed. Enforcing *k-consistency* extends the notion of 1-consistency, requiring any partial assignment involving  $k - 1$  variables to be extendible to some  $k$ -th assignment. For CSP instances, the notion of *width* relates to how tree-like the structure of the instance is. A class of structures is of *bounded width* if there is a finite upper bound on how large the width may be of any structure in the class.

Several restrictions on the structure of a CSP are known to guarantee tractability. CSPs which can be solved using the  $k$ -consistency algorithm are those with width  $k$  [5].

Versions of bounded width that guarantee tractability include binary CSPs with tree structure [55], binary CSPs with structures of bounded treewidth [49, 57], CSPs with structures that have bounded generalized hypertree width [71], and CSPs which have cores with bounded fractional hypertree width [75]. I discuss fractional hypertree width in more detail in Chapter 7.

Requiring bounded width for structures of a CSP rules out NP-complete CSPs that are constructed in a manner akin to  $\text{CSP}(\mathfrak{N}, \_)$ . Such CSPs obtain their high complexity from the arbitrarily large width of the structures in each instance. Intuitively, large width guarantees that there is a large subproblem in each instance where a large fraction of all possible combinations of values need to be checked.

More recently, Marx introduced two related width measures called adaptive width and submodular width [117, 118]. Every class with bounded width by any other measure also has bounded adaptive and submodular width, and can be decided in polynomial time. In contrast, there exist classes with bounded submodular and adaptive width for which other width measures are unbounded, yet such classes can be decided in polynomial time. To date, adaptive and submodular width together define the most general width measures that can guarantee tractability, but these measures require the tree decomposition to change dynamically as the constraint solver makes progress. It is therefore difficult to relate them to structural classes that are defined statically.

### 3.2.3 Reasons for tractability: language

Many restrictions of the form  $\text{CSP}(\_, \mathfrak{B})$  have been studied. The specific case where  $\mathfrak{B}$  contains a single structure, is known as the *nonuniform* problem [104].

The simplest case is when the structures in  $\mathfrak{B}$  are restricted to the signature of one binary relation. In this case,  $\text{CSP}(\_, \mathfrak{B})$  can be solved in polynomial time when  $\mathfrak{B}$  is a class of bipartite graphs, but is NP-complete otherwise [82].

Another language dichotomy is known over Boolean domains, when all structures in  $\mathfrak{B}$  have a domain with just two elements. For the Boolean case, if each structure in  $\mathfrak{B}$  is of

one of six types identified by Schaefer, then  $\text{CSP}(\_, \mathfrak{B})$  can be solved in polynomial-time, and is NP-complete otherwise [142].

In both of these cases the class  $\mathfrak{B}$  takes a very specific form, but the class  $\mathfrak{A}$  of source structures is not restricted. The dichotomy conjecture, first posed by Schaefer, states that these are the only cases, even for domains of larger size: every problem of the form  $\text{CSP}(\_, \mathfrak{B})$  is conjectured to be either NP-complete or to be possible to solve in polynomial time [142]. Bulatov showed this conjecture is true for several important cases, for instance if the domain has three elements [23], but it remains open in the general case.

The language of relations used to express a CSP may also be restricted enough to guarantee tractability of the CSP. Algebraic properties of the relations provide a unifying framework for explaining tractability.

**Definition 3.31** (language of class of structures). Let  $\mathfrak{A}$  be a class of relational structures. The **language** defined by  $\mathfrak{A}$ , denoted  $\Gamma(\mathfrak{A})$ , is the class of all relations that occur in some structure in  $\mathfrak{A}$ :  $\Gamma(\mathfrak{A}) = \bigcup_{S \in \mathfrak{A}} \Gamma(S)$ .  $\square$

**Definition 3.32** (structures over language). Let  $\Gamma$  be a class of relations. The **class of structures** over language  $\Gamma$ , denoted  $\text{Struct}(\Gamma)$ , is the class  $\mathfrak{A}$  of relational structures constructed only using relations in  $\Gamma$ , such that  $S \in \mathfrak{A}$  precisely when  $\Gamma(S)$  is contained in  $\Gamma$ .  $\square$

A CSP instance may possess a linear ordering of the domain of every variable. For instance, each domain may be some subset of integers, ordered by the usual ordering on integers. This notion will be used in Chapter 6 in constructing a class of tractable CSPs.

**Definition 3.33** (max-closed). Suppose  $R$  is a relation of arity  $r$  on ordered domains with the following property: if  $u$  and  $v$  are tuples of  $R$ , then the tuple

$$(\max\{u_1, a_1\}, \max\{u_2, a_2\}, \dots, \max\{u_r, a_r\})$$

is also in  $R$ . In this case  $R$  is called **max-closed**.  $\square$

Fix an ordered domain, and let  $\mathfrak{M}$  be the class of all max-closed relations with respect to the domain. A max-closed CSP instance with respect to this domain is then any instance which has all its relations in  $\mathfrak{M}$ , i.e. any instance in  $\text{CSP}(\_, \text{Struct}(\mathfrak{M}))$ . Max-closed instances form a tractable class that is maximal, in the sense that the class  $\text{CSP}(\_, \text{Struct}(\mathfrak{M}))$  is tractable, but  $\text{CSP}(\_, \text{Struct}(\mathfrak{M} \cup \{R\}))$  is NP-complete if  $R$  is any relation over the domain that is not max-closed [96].

Cohen and Green further considered those CSPs where the domains of variables of every instance can be provided with an ordering so that the instance is max-closed [37]. This class is NP-hard to recognise.

Bulatov, Krokhin, and Jeavons have conjectured that  $\text{CSP}(\_, \text{Struct}(\Gamma))$  is NP-complete precisely when  $\Gamma$  has a particular algebraic property [22, Conjecture 7.5, “tractable algebras conjecture”]. They conjectured that when this property does not hold, then the witness

to this algebraic fact can be exploited to construct a polynomial-time algorithm for  $\text{CSP}(\_, \text{Struct}(\Gamma))$ .

### 3.2.4 Reasons for tractability: hybrid

While the restrictions in Section 3.2.2 and Section 3.2.3 are natural from a theoretical perspective, they do not correspond well to CSPs that occur in practice. The class of all-different constraints, which is central in constraint satisfaction, falls into no tractable structural or language class. Yet, it is in itself not difficult to work with – it is possible to decide whether a solution exists in quadratic time [134]. It turns out that satisfactory explanations for tractability can be constructed through restrictions that are neither purely structural nor purely language-based.

The following definition is essentially the same one used by Cohen [34].

**Definition 3.34.** A *hybrid* CSP is a CSP that cannot be defined as either a structural restriction, or a language restriction. □

Some examples of hybrid classes have appeared sporadically in the literature. Dechter and Pearl considered CSPs that form causal theories, when a solution can be found without backtracking [50]. These form hybrid classes. Van Beek and Dechter examined row-convex constraint networks, which are also hybrid [151]. Cooper and Živný discussed a class defined by restricting how assignments between three variables may be constrained [40].

Hybrid classes have also been defined indirectly.

Chen, Thurley, and Weyer considered hybrid classes in which both the source and target structures of each instance have inequalities [26]. For completeness, I show that this enforces injectivity of solutions.

**Lemma 3.35.** *Let  $S = (V(S), (Q_i)_{i \in I})$  and  $T = (V(T), (R_i)_{i \in I})$  be relational structures. There exists an injective relational structure homomorphism from  $S$  to  $T$  if, and only if, there is a relational structure homomorphism from  $S^\neq = (V(S), (Q_i)_{i \in I}, \neq_{V(S)})$  to  $V^\neq = (V(T), (R_i)_{i \in I}, \neq_{V(T)})$ .*

*Proof.* Recall the INJECTIVE constraint in Example 2.29.

Suppose  $\phi$  is an injective relational structure homomorphism from  $S$  to  $T$ . As  $\phi$  is injective, it preserves the additional INJECTIVE constraint in  $(S^\neq, T^\neq)$ , and is therefore also a relational structure homomorphism from  $S^\neq$  to  $T^\neq$ .

Now suppose  $\phi$  is a relational structure homomorphism from  $S^\neq$  to  $T^\neq$ . The added inequality relations in  $(S^\neq, T^\neq)$  form an INJECTIVE constraint. By Proposition 3.9, the infrastructure of the CSP instance  $(S^\neq, T^\neq)$  then contains all partial assignments that are injective and that are also props of  $(S, T)$ . Hence any solution of  $(S^\neq, T^\neq)$  is also a solution of  $(S, T)$ , and such a relational structure homomorphism must also be injective.

It follows that the two conditions are equivalent. □



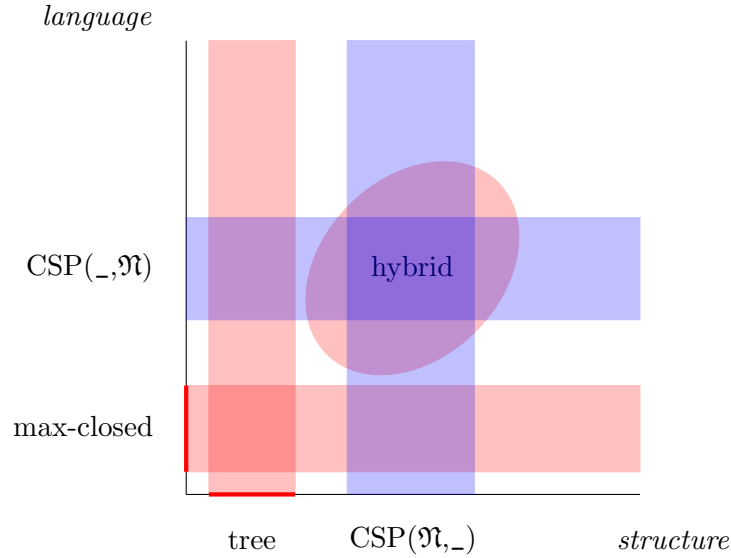


Figure 3.1: Hybrid tractability illustrated.

By Lemma 3.35, the classes considered by Chen, Thurley, and Weyer can equivalently be thought of as defined by source restrictions but over the class of all injective instances, which each include an INJECTIVE constraint. A similar construction was also used by Bodirsky and Grohe [18].

The conceptual motivation of investigating hybrid tractable classes is illustrated in Figure 3.1. Here a hybrid class is shown that is not defined by means of either tractable structure or tractable language, yet is tractable for other reasons (such as those I discuss in later chapters). A tractable class defined by language restrictions is shown (max-closed), and a tractable class defined by structural restrictions is also illustrated (tree). Two further classes are also illustrated. The class  $\text{CSP}(\mathfrak{N}, \_)$  is defined by structural restrictions and the class  $\text{CSP}(\_, \mathfrak{N})$  by language restrictions. Neither of these classes is tractable, but their intersection is tractable because it falls within the hybrid class defined by the ALL-DIFFERENT constraints. This hybrid class is discussed in Chapter 5, and is shown to be part of an even larger hybrid class defined in terms of properties of the microstructure.

### 3.3 Transformations between representations

The representation of the relations of a CSP as sentences of first-order logic is an intensional description of the nogoods of each relation by means of a formula. The variable-value and homomorphism representations are extensional, with the relations (or sometimes the nogoods) explicitly listed as sets of tuples [59].

Starting with the variable-value representation, I now show how to transform between the three representations.

To set up the transformations formally, a function  $n(i, j): I \times \mathbb{N} \rightarrow [s]$  is needed to label the  $j$ -th variable appearing in the  $i$ -th predicate, and the  $j$ -th variable appearing



in the scope of the  $i$ -th constraint. It is also convenient to let  $r_i$  denote  $\rho(Q_i) = \rho(R_i)$ , the arity of the  $i$ -th constraint or the  $i$ -th predicate, and to let  $\bar{u}$  denote the tuple of all variables indexed by  $[s]$ .

### 3.3.1 Variable-value to first-order logic

The description of the first-order logic representation already provided the transformation. A primitive positive formula expresses the requirement that each constraint be satisfied, by means of a conjunction of predicates obtained from the relations in each constraint, each predicate being applied to the variables in the associated scope.

Source representation	Target representation
$(V, D, C)$	$\exists \bar{u} \bigwedge_{(\sigma, R) \in C} R(\sigma_1, \sigma_2, \dots, \sigma_{\rho(\sigma)})$ where $\bar{u}$ is the list of all variables in $V$

### 3.3.2 Variable-value to homomorphism

To transform the variable-value representation to the homomorphism representation, the constraints simply need to be rearranged. Instead of pairs containing scopes and relations, each scope finds a home as a tuple in a relation over the variables in the source structure, and the corresponding relation over values moves to the target structure.

Source representation	Target representation
$(V, D, C)$	$\langle (V, (Q_i)_{i \in I}), (D, (R_i)_{i \in I}) \rangle$ where $I = [ C ]$ $C = \{C_i \mid i \in I\}$ $r_i = \rho(C_i)$ $C_i = ((u_1, u_2, \dots, u_{r_i}), R_i)$ $Q_i = \{(u_1, u_2, \dots, u_{r_i})\}$

The set  $I$  provides an index into the set  $C$  of constraints. The variables are used to provide the vertices of the source structure, and the domain values are used for the vertices of the target structure. Note that each of the relations  $Q_i$  consists of a single tuple of variables.

Whenever  $R_i = R_j$  for some distinct indices  $i$  and  $j$ , then it is possible to remove  $j$  from the index set  $I$ , and to replace  $Q_i$  by a new relation  $Q_i \cup Q_j$  containing all tuples in  $Q_i$  and  $Q_j$ . This process can be continued until each relation only occurs once in the target structure.

If  $Q_i = Q_j$  for some distinct indices  $i$  and  $j$ , then it is again possible to remove  $j$  from the index set  $I$ , and to replace  $R_i$  by a new relation  $R_i \cap R_j$  containing all tuples that  $R_i$  and  $R_j$  have in common.

Applying these two optional simplifications can make the CSP more compact. However, it may be desirable to maintain a regular structure or a language containing only a few relations, and these simplifications are therefore not always desirable.

### 3.3.3 First-order logic to variable-value

It is straightforward to transform an instance in the first-order logic representation to the variable-value representation. Each conjunct becomes a constraint. The list of variables in the relational predicate becomes the scope of the constraint, and the relation symbol denotes the constraint relation.

Source representation	Target representation
$\exists \bar{u} \bigwedge_{i \in I} R_i(u_{n(i,1)}, u_{n(i,2)}, \dots, u_{n(i,r_i)})$	$(V, D, C)$ where $V = \{u_1, u_2, \dots, u_s \mid \bar{u} = (u_1, u_2, \dots, u_s)\}$ $D = \bigcup \{\{a_1, a_2, \dots, a_{r_i}\} \mid R_i(a_1, a_2, \dots, a_{r_i})\}$ $C = \{((u_{n(i,1)}, u_{n(i,2)}, \dots, u_{n(i,r_i)}), R_i) \mid i \in I\}$

### 3.3.4 First-order logic to homomorphism

It is also straightforward to transform an instance in the logic representation to the homomorphism representation. This can be seen by first transforming to the variable-value representation as in Section 3.3.3, then to the homomorphism representation as in Section 3.3.2.

### 3.3.5 Homomorphism to variable-value

Given a pair of relational structures with the same signature, the vertices of the source structure become the variables, the vertices of the target structure become the values, and each tuple in a source relation generates a constraint.

Source representation	Target representation
$\langle (V(S), (Q_i)_{i \in I}), (V(T), (R_i)_{i \in I}) \rangle$	$(V, D, C)$ where $V = V(S)$ $D = V(T)$ $C = \bigcup \{ \bigcup \{ (\sigma, R_i) \mid \sigma \in Q_i \} \mid i \in I \}$

An optional simplification can be used if one wishes to avoid constraints with repeated variables in the scope. Whenever some variable  $u$  occurs in some scope where it has already appeared, then its second appearance is replaced by a new variable  $u'$  that does not already appear in the set of variables  $V$ . Variable  $u'$  is added to  $V$ , and an additional constraint  $((u, u'), =)$  is added to the set of constraints  $C$ . This process can be continued until in each

scope variables are distinct. This transformation can be applied if the equality relation is available.

For some restricted classes equality may not be available. In such cases, a scope with repeated variables can be replaced by a lower-arity scope with no such repeats, by replacing the associated relation with one that has only those tuples that are equal in the positions of the repeated variable.

### 3.3.6 Homomorphism to first-order logic

Transforming from the homomorphism representation to first-order logic is again straightforward: first transform to the variable-value representation as in Section 3.3.5, then to the first-order logic representation as in Section 3.3.1.

## 3.4 Chapter summary

In this chapter I defined several key concepts, including the infrastructure, subproblems, infrastructure-equivalent instances, and how constraints can be combined. I also discussed structural, language, and hybrid tractability in more detail. Finally, I examined transformations between the standard representations.

I first defined the infrastructure of a CSP instance and the associated notion of equivalence between instances. I discussed how constraints may be combined, in particular with the UNARY constraint, and how adding UNARY constraints is related to maintaining different domains associated with each variable. I contrasted three different notions of instance equivalence, and discussed their complexity.

I then reviewed classes of tractable CSPs that have been defined using restrictions on the structure or the language, and some prior work on combinations of structural and language restrictions.

Finally, I considered the three standard representations that have been used in the literature. They can be transformed to each other in straightforward ways, since they are essentially just different syntax.

The following three chapters deal with the rather different microstructure representation about which I develop new theory, first in the context of general arity and then for fixed arity. I then modify this representation to allow additional features to be specified.

*Not only is every fact really a relation, but your thought of the fact implicitly represents it as such.*

—C. S. Peirce, *The Critic of Arguments*, 1892. §417.

# 4

## Microstructure representation

In the next three chapters, I discuss the microstructure representation. This representation captures all the information about a CSP instance in a single structure. There are two versions of the microstructure representation, either by means of instance props in the microstructure, or via instance nogoods in the clause structure.

Sometimes a fixed upper bound is assumed on the arity of relations that appear in a CSP instance, as in the following chapter. I allow CSPs of arbitrary arity in this chapter.

In Section 4.1 I first define literals and how they are combined in the microstructure representation. I discuss the microstructure and the clause structure. These collect together sets of literals, forming partial assignments.

In Section 4.2, I then discuss some applications of the microstructure representation. Props of the CSP instance correspond to independent sets in the clause structure, and to down-cliques in the microstructure. The microstructure representation can sometimes be drawn as pictures to aid understanding. Section 4.3 deals with examples.

Transformations to the microstructure representation are treated in some detail in Section 4.4. After briefly discussing how instance size changes via such transformations, I cover some standard techniques to normalise instances in Section 4.4.1, and in Section 4.4.2, discuss how combinations of constraints affect the microstructure representation. Section 4.4.3 covers the link between the clause structure of a CSP instance and a SAT instance that is obtained by applying the direct encoding transformation to the instance. Finally, in Section 4.4.4 I develop an alternative way of understanding the microstructure as a product of augmented relational structures. These are obtained by augmenting both the source and target relational structures with one additional equality relation, together with relational structure complementation.

## 4.1 Building a representation from literals

Microstructures are collections of facts, represented as structures with literals as vertices.

I first review the definitions of partial assignments and nogoods in terms of literals. Each literal corresponds to a fact about a variable-value pair. An assigning literal is a proposition allowing a particular value to be assigned to that variable, and an avoiding literal is a proposition forbidding a particular value from being assigned to that variable.

**Definition 4.1** (literal). A **literal** is a pair  $(u, a_u)$ , where  $u$  is a variable and  $a_u \in D_u$  is a value in the domain of  $u$ . This represents a proposition of the form  $(u = a_u)$  (called an **assigning literal**) or  $(u \neq a_u)$  (an **avoiding literal**).  $\square$

Props and nogoods are sets of literals; props are sets of assigning literals and nogoods are sets of avoiding literals. Every partial assignment is a set of assigning literals. Conversely, when a set of assigning literals is a function, then it is a partial assignment. Whether a literal is assigning or avoiding will be clear from the context.

A partial assignment of literals is a set of pairs, so it is a relation over the Cartesian product of the set of variables and the set of values in the instance. Moreover, a partial assignment of literals is by definition a function.

Recall that the infrastructure of a CSP instance is the set of all props. The infrastructure of a CSP instance therefore consists of partial assignments, each of which is just a set of assigning literals. The infrastructure of an instance  $\mathcal{P}$  with variables  $V$  and values  $D$  can then also be regarded a hypergraph with vertices  $V \times D$ , and with the set of all props as the edges. Each of the representations considered in previous chapters represents the potentially exponentially many partial assignments in the infrastructure in a more or less compact form.

Literals are used to assemble a representation of a CSP instance. The choice of whether literals are assigning or avoiding is made once for the representation. Assigning literals are used in the *microstructure* and avoiding literals in the *clause structure*, and I refer to either of these as the *microstructure representation*. These are now discussed in turn.

### 4.1.1 Microstructure

Recall that an instance prop is a prop with the same variables as those of some constraint. The instance props are those which are directly specified in the CSP instance.

**Definition 4.2** (microstructure). The **microstructure** of a CSP instance  $\mathcal{P}$  with variables  $V$  and values  $D$ , denoted  $\text{MS}(\mathcal{P})$ , is the hypergraph with vertices  $V \times D$  and the set of instance props of  $\mathcal{P}$  as edges.  $\square$

By definition, the edges of the microstructure form a subset of the infrastructure, and the microstructure can be regarded as a subhypergraph of the infrastructure.

It is sometimes useful to add anything-goes constraints to a CSP instance. When such constraints involve variables that are not already constrained, then adding such constraints

does not change the infrastructure of the instance; a formal proof of this is postponed to Corollary 4.22. It can be helpful to force the microstructure to include such props, by including all the implicit anything-goes constraints of relevant arities in the instance.

**Definition 4.3** (explicit CSP instance). Suppose  $\mathcal{P}$  is a CSP instance of arity  $r$  (so it has some constraint of arity  $r$  and no constraints of arity larger than  $r$ ). The instance  $\mathcal{P}$  is **explicit** if it includes a constraint involving every set of  $r'$  distinct variables, where  $1 \leq r' \leq r$ . Given a CSP instance  $\mathcal{P}$  of arity  $r$ , the **explicit version** of  $\mathcal{P}$  is the explicit instance  $\mathcal{P}^+$  obtained from  $\mathcal{P}$  by adding anything-goes constraints to all otherwise unconstrained sets of  $r'$  variables of  $\mathcal{P}$ , for every  $1 \leq r' \leq r$ .

Note that the explicit version of a binary CSP instance is no longer binary, since it has an anything-goes unary constraint for each variable. Also note that only one anything-goes constraint is added per subset of variables; this can easily be achieved by choosing the scope for each such constraint that is lexicographically smallest in some arbitrary ordering of the variables.

Further, note that the instance props are not meaningful in isolation. Individually they simply specify that a partial assignment may be possible, but the assignment may violate a different constraint.

However, the collection of all instance props provides a lot of information when it is considered as a whole. For each constraint, *only those partial assignments defined by the constraint are possible*; everything else violates the constraint. Hence the practical value of the instance props is really in the partial assignments they forbid: these are the instance nogoods. I now turn to a representation in which the instance nogoods are in the foreground.

#### 4.1.2 Clause structure

A clause is a set containing avoiding literals.

**Definition 4.4** (clause). A **clause**  $\phi$  is a set of avoiding literals, representing a proposition of the form  $(u_{i_1} \neq a_{i_1}) \vee \cdots \vee (u_{i_r} \neq a_{i_r})$ , where  $u_{i_1}, \dots, u_{i_r}$  are variables (not necessarily all distinct), and  $v_{i_1}, \dots, v_{i_r}$  are values (again not necessarily all distinct).  $\square$

A clause  $\phi = \{(u_1, v_1), (u_2, v_2), \dots, (u_r, v_r)\}$  is associated with the proposition

$$\phi = \bigvee ((u_1 \neq v_1) \vee (u_2 \neq v_2) \vee \cdots \vee (u_r \neq v_r))$$

which is equivalent to

$$\phi = \neg((u_1 = v_1) \wedge (u_2 = v_2) \wedge \cdots \wedge (u_r = v_r)),$$

the negation of the proposition  $((u_1 = v_1) \wedge (u_2 = v_2) \wedge \cdots \wedge (u_r = v_r))$ , which is associated with the partial assignment  $\{(u_1, v_1), (u_2, v_2), \dots, (u_r, v_r)\}$ . A clause can therefore also be regarded as a set of assigning literals that violates some partial assignment.

A clause is a set of literals, so it forms a relation over the Cartesian product of the set of variables and the set of values. A clause is not necessarily a function, and it will be convenient not to force clauses to be functions by definition. Instead, to capture the notion that every partial assignment is a function, either this requirement can be enforced by means of colours, as done in Chapter 6, or additional clauses can be recorded forbidding multiple values being assigned to the same variable by any partial assignment.

**Definition 4.5** (variable clause). A **variable clause** for variable  $u$  is a clause of the form  $\{(u, a_1), (u, a_2)\}$  representing a proposition of the form  $(u \neq a_1) \vee (u \neq a_2)$ , where  $a_1 \neq a_2$ . The variable clauses for a variable  $u$  consist of all variable clauses for all possible pairs  $v_1 \neq v_2$  of values  $v_1, v_2 \in D_u$ . The variable clauses of a CSP instance are formed as the union of all variable clauses for each variable  $u$  in the instance.  $\square$

Recall that an instance nogood is a nogood with the same variables as those of some constraint. The instance nogoods are those which are directly specified in the CSP instance.

**Definition 4.6** (clause structure). The **clause structure** (or **microstructure complement**) of a CSP instance  $\mathcal{P}$  with variables  $V$  and values  $D$  is the hypergraph with literals  $V \times D$  as vertices, and instance nogoods and variable clauses of  $\mathcal{P}$  as edges. The clause structure is denoted  $\text{CS}(\mathcal{P})$ .  $\square$

To ensure the instance nogoods in the clause structure are functions, the additional variable clauses are included in the clause structure, disallowing any variable being assigned more than one value. Because the clause  $\{(u, a), (u, b)\}$  cannot be extended to a solution for any variable  $u$  and any distinct values  $a, b \in \text{dom}(u)$ , it is a nogood. The collection of all such pairwise clauses forms an important part of the clause structure.

### 4.1.3 Microstructure representation

The **microstructure representation** refers to either the microstructure, containing partial assignments that are instance props; or the clause structure, containing instance nogoods and variable clauses to enforce that solutions only have one value assigned to each variable.

It will be clear whether the context is the microstructure or the clause structure of a CSP instance. In a set of clauses, the literals are all avoiding; in a set of partial assignments the literals are all assigning. Instead of interpreting clauses as logical formulae, a clause forms an edge in the hypergraph which has the set of all literals as its vertices. One of the semantics for edges needs to be chosen, either partial assignments (for microstructure) or clauses (for clause structure).

The interpretation of the microstructure relies on a **closed-world assumption**, in which partial assignments not explicitly allowed by the constraints are forbidden. Hence in a CSP instance each constraint relation specifies all possible combinations of values for the associated constraint scope, and all tuples not in the constraint relation specify instance

nogoods. Equivalently, any partial assignment that is not contained in some prop defines an instance nogood.

In contrast, for the clause structure an *open-world assumption* is appropriate. In a CSP instance each relation specifies information that is known so far, but the arrival of new nogoods may have to be accommodated. Each new nogood may cause some possible solutions to violate the constraints. Even if a nogood introduces a literal that wasn't previously in the microstructure, then that nogood can simply be added to the microstructure by adding a new constraint to the instance, forbidding that single nogood. No new solutions are introduced in this way, but no existing solutions are removed either. In other words, the clause structure allows the graceful monotonic addition of new information.

Because the set of nogoods naturally admits addition of additional nogoods, while incorporating new constraints in the infrastructure may require additional information about the nogoods implied by any new prop, it can be argued that the clause structure is a more natural representation for CSP instances than the microstructure. However, when the constraint relations are small compared to their complements, then the microstructure may require much less space. Similarly, when only a few tuples are not present in each constraint relation, the clause structure requires less space to represent than the microstructure. These differences can be large enough that they result in changing the complexity of the CSP [87]. It is therefore not reasonable to restrict attention to only the microstructure or only the clause structure.

The microstructure and the clause structure represent the information that is explicit in the instance description. Collections of clauses that include implicit clauses form a useful abstraction of the collection of all information available at any stage of the process of looking for a solution. Knowledge about the CSP instance, through the instance nogoods specified in the instance together with those nogoods that have been found during search, forms a set of clauses that approximates the set of nogoods. If by a systematic process of testing each possible clause is checked for being a prop or a nogood, the set of nogoods is obtained, and in tandem its complement the infrastructure. However, this set has exponential size in the general case, so such a procedure is not useful for practical reasons. In special cases either the infrastructure or the set of nogoods may be small, but together they contain all subsets of variable-value assignments which has exponential size in the number of such assignments.

The microstructure specifies an over-approximation of one part of the infrastructure. It consists of those partial assignments defined by the constraints, and some of these may actually be nogoods. Even if a partial assignment is an instance prop, further search may establish that it is actually an implicit nogood. On the other hand, the clause structure is an under-approximation of the part of the set of nogoods that relates to the constraints, and therefore of the set of nogoods: some nogoods are usually not explicit, and may need to be discovered while attempting to find a solution.

The partial assignments or clauses in the microstructure and the clause structure can



name	microstructure	infrastructure	clause structure	explicit nogoods
literal semantics	assigning	assigning	avoiding	avoiding
scopes	instance only	all scopes	instance only	all scopes

Table 4.1: The structures associated with the family of microstructure representations.

be represented in various ways. For the purposes of this chapter both of these variations on the microstructure representation may be hypergraphs. In Chapter 5 I will consider the case where all constraints are at most binary, which leads to microstructures and clause structures that are graphs. In Chapter 6 I will treat the microstructure representation of binary instances as vertex-coloured graphs.

The collection of all explicit nogoods contains all partial assignments that are not props, so it forms the complement of the infrastructure with respect to the set of all partial assignments. I mention this form of the microstructure representation for completeness of the following table, but will not discuss it further.

Table 4.1 provides a summary of how the two main features differ between the different collections of partial assignments that feature in the microstructure representation family. The first main feature is which semantics is used for literals, either assigning or avoiding. This determines whether the partial assignment is allowed or disallowed, respectively. The second feature is whether the partial assignments in a form of the microstructure representation correspond to only those scopes that appear in the instance, or whether all scopes are considered.

#### 4.1.4 Graphical notation

It is often useful to draw pictures illustrating microstructures, infrastructures, clause structures, and the set of nogoods. These are all hypergraphs on the set of literals. Each vertex is a literal, and the edges are partial assignments that are either props or nogoods (depending on whether the picture represents the microstructure or the clause structure). A vertex is represented as a dot  $\bullet$  and an edge is represented either by dots connected by a line  $\bullet\text{---}\bullet$  or a region enclosing some dots  $\bullet\text{---}\bullet$  or a shaded region enclosing some dots  $\bullet\text{---}\bullet$  where the choice of line, region, or shaded region is not significant but simply serves to differentiate different edges visually. I also adopt the convention that the literals corresponding to one variable are all collinear, forming a vertical line  $\vdots$  containing all the dots relating to that variable. I provide pictures of the microstructure representation for several examples in the next section.

There are two non-obvious consequences of the definitions, which affect the pictures of the microstructure representation. In the microstructure additional edges are needed to record anything-goes constraints between unconstrained sets of variables, if it is desired to maintain the correspondence between props of the instance and down-cliques in the microstructure. The clause structure does not require anything to be done about unconstrained sets of variables, but additional edges are needed in the clause structure to

record the variable clauses between literals with the same variable. In either version of the microstructure representation, the visual clutter of the additional edges reduces the utility of drawing pictures of the microstructure representation except for quite small instances.

In the microstructure there are edges for each partial assignment that is an instance prop. By definition, when a set  $U$  of variables is not wholly contained as a subset in the underlying set of any constraint scope, then every partial assignment that is defined on  $U$  is a prop. In particular, if every constraint has arity at most  $r$  but  $U$  contains more than  $r$  variables, then the microstructure contains an anything-goes constraint on a scope with set  $U$ . This feature of the definition makes it difficult to represent microstructures graphically, as there are typically many edges defined by these anything-goes constraints, and they form dense collections of edges. In practice, such anything-goes props can be omitted from pictures of the microstructure, but it is important to keep in mind their implicit presence if they are omitted.

In the clause structure there are edges for each partial assignment that is explicitly disallowed by some constraint. In addition, there are additional edges to enforce the variable clauses requiring that each variable may only be assigned at most one value. These additional edges form cliques of binary edges connecting all the vertices that have the same variable. I will often indicate such a clique of variable clauses by shading the region containing the vertically collinear literals for that variable, or by enclosing literals with the same variable by a vertical oblong (which for this application should not be interpreted as a clause).

Unlike the microstructure, the clause structure does not use additional edges for anything-goes constraints between variables that are not constrained. In the clause structure, if a set of variables does not appear as the set of any constraint scope, then there are simply no edges involving all those variables.

The microstructure representation is usually difficult to present graphically. However, for small instances and instances with a lot of regular structure it is possible to use pictures for the microstructure representation. The next section discusses several examples, for many of which the microstructure representation can be presented graphically.

**Example 4.7** (nogoods illustrated). The microstructure of the CSP instance

$$(\{u_1, u_2\}, \{0, 1\}, \{((u_1), \{(0), (1)\}), ((u_1, u_2), \{(0, 1)\})\})$$

in variable-value representation contains edges  $\{(u_1, 0), (u_2, 1)\}$ ,  $\{(u_1, 0)\}$ , and  $\{(u_1, 1)\}$ . These are precisely the partial assignments specified in the instance description, illustrated in Figure 4.1. The only clauses explicitly forbidden by the instance description are those illustrated in the clause structure in Figure 4.2, including the variable clauses. An implicit anything-goes constraint  $((u_2), \{(0), (1)\})$  involving the variable  $u_2$  would be added to the microstructure of the explicit version of this instance.

Note the unary props in the microstructure. Usually unary constraints are removed by means of a preprocessing step, as discussed in Section 4.4.1.  $\perp$

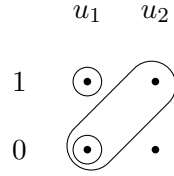


Figure 4.1: Microstructure of CSP instance in Example 4.7.

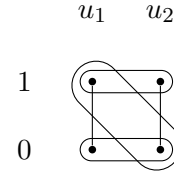


Figure 4.2: Clause structure of CSP instance in Example 4.7.

## 4.2 Solutions in the microstructure representation

The microstructure representation has some interesting properties. I first discuss the relationship between partial solutions of a CSP instance with sets of vertices in its microstructure representation that form cliques or independent sets, and the appropriate decision problem associated with the microstructure representation.

The following definition of independent set generalizes the usual definition for graphs (see Section 2.2.3) to hypergraphs, and appears to date back at least several decades [112]. I introduce the down-clique notion as the natural counterpart of an independent set in a hypergraph.

### Definition 4.8.

1. An **independent set** in a hypergraph is a subset of the vertices which does not contain any edge in the hypergraph as a subset. An independent set containing  $s$  vertices is an  **$s$ -independent set**.
2. A **down-clique** in a hypergraph is a subset  $X$  of the vertices such that every non-empty subset  $Y$  of  $X$  forms an edge in the hypergraph. A down-clique containing  $s$  vertices is an  **$s$ -down-clique**.

Note that Definition 4.8 requires a hypergraph to contain all non-empty subsets of any of its down-cliques as edges. In comparison, the definition of a clique in a graph (see Section 2.2.3) only requires all subsets of cardinality two to be edges. This discrepancy can be resolved as follows. Consider the graph  $G = (V, E)$ . Let  $E^+$  contain all singleton edges and all subsets  $U \subseteq V$  that form cliques in  $G$ , to form a hypergraph  $G^+ = (V, E^+)$ . Note that  $E^+$  contains precisely the same edges of cardinality two as does  $E$ . Then  $X$  is a down-clique in  $G^+$  precisely when  $X$  is a clique in  $G$ . I will therefore elide the distinction between down-clique and clique when discussing graphs.

$X$  is an independent set in a graph  $G$  if, and only if,  $X$  forms a clique in  $\overline{G}$ . For hypergraphs and their complements, an analogous relationship holds between independent sets and down-cliques, by Definition 4.8. (Definition 2.12 defines the complement of a hypergraph, and recall that hypergraphs contain only non-empty edges.)

*Observation 4.9.*  $X$  is an independent set in a hypergraph  $G$  if, and only if,  $X$  forms a down-clique in the complement  $\overline{G}$  of  $G$ .

A solution  $\phi$  to a CSP instance is a function mapping the set of variables to the set of values. Each tuple in  $\phi$  is therefore of the form  $(u, \phi(u))$ , where  $u$  is a variable and  $\phi(u)$  is the corresponding value assigned to  $u$  by  $\phi$ .

In the microstructure of an explicit binary CSP instance  $\mathcal{P}$  with variables  $u_1, u_2, \dots, u_s$ , a solution is a set of literals  $\phi = \{(u_1, \phi(u_1)), (u_2, \phi(u_2)), \dots, (u_s, \phi(u_s))\}$ , such that for every pair of distinct literals  $(u, a)$  and  $(v, b)$  in  $\phi$ , the tuple  $((u, a), (v, b))$  is an edge of  $\text{MS}(\mathcal{P})$ . Note that this means no two literals in  $\phi$  have the same variable, so  $\phi$  contains precisely  $n$  literals, each with a different variable. This then means that  $\phi$  forms a clique in  $\text{MS}(\mathcal{P})$ , an observation that has been made before.

**Proposition 4.10** ([97, Property 2]).  *$\phi = \{(x_1, a_1), (x_2, a_2), \dots, (x_n, a_n)\}$  is a solution to a binary CSP instance with  $n$  variables if, and only if,  $\phi$  forms a clique in the microstructure of the explicit instance.*  $\square$

Similarly, in the clause structure every pair  $((u, a), (v, b))$  in  $\phi$  must not be an edge of  $\text{CS}(\mathcal{P})$ , so  $\phi$  forms an independent set in  $\text{CS}(\mathcal{P})$ . Combining Observation 4.9 with a version of Proposition 4.10 for CSP instances of general arity yields the following.

**Proposition 4.11.** *Suppose  $\mathcal{P}$  is a binary CSP instance with  $s$  variables  $u_1, u_2, \dots, u_s$  which has no constraints with repeated variables in any scope. The following are equivalent:*

1.  $\phi = \{(u_1, a_1), (u_2, a_2), \dots, (u_s, a_s)\}$  is a solution to  $\mathcal{P}$ ,
2.  $\phi$  is an  $s$ -clique in  $\text{MS}(\mathcal{P}^+)$ , or
3.  $\phi$  is an  $s$ -independent set in  $\text{CS}(\mathcal{P})$ .

*Proof.* First I will show  $(2 \Rightarrow 1)$ . Consider an  $s$ -clique  $\phi$  in the microstructure of the explicit version of  $\mathcal{P}$ , and let  $\{(u_i, v_i), (u_j, v_j)\}$  be a pair of distinct literals in  $\phi$ . Since every constraint in  $\mathcal{P}$  involves a scope with no repeated variables, and this pair is an instance prop of  $\mathcal{P}^+$ , it must be the case that  $u_i \neq u_j$ .

Since  $\phi$  is a clique, this pair is an edge of the microstructure of  $\mathcal{P}^+$ . If this pair comes from some anything-goes constraint, then there is no other constraint with scope having variables  $u_i$  and  $u_j$ , so  $\phi$  does not violate any constraint involving variables  $u_i$  and  $u_j$ . Similarly, if this pair is an instance prop of  $\mathcal{P}$ , then  $\phi$  again violates no constraint involving variables  $u_i$  and  $u_j$ .

Since the choice of the pair of distinct literals in  $\phi$  was arbitrary,  $\phi$  violates no constraint of  $\mathcal{P}$ . Hence  $\phi$  is a prop. Further, the list  $u_1, u_2, \dots, u_s$  must contain  $s$  distinct variables. Therefore  $\phi$  must be a complete assignment and hence a solution.

For  $(1 \Rightarrow 2)$ , suppose  $\phi$  is a solution to  $\mathcal{P}$ . Hence  $\phi$  is a function, and  $u_i \neq u_j$  whenever  $i \neq j$ . If  $(u_i, v_i)$  and  $(u_j, v_j)$  are distinct literals of  $\phi$  then  $u_i \neq u_j$ , so  $\{(u_i, v_i), (u_j, v_j)\}$  is either a prop of  $\mathcal{P}$ , or there is no constraint with scope containing both  $u_i$  and  $u_j$ . Hence  $\phi$  must form a clique in the microstructure of  $\mathcal{P}^+$ .

The remaining implications follow since a clique in a graph  $G$  is an independent set in  $\overline{G}$ , and vice versa.  $\square$

Proposition 4.11 extends to CSP instances with no restriction on arity.

**Proposition 4.12.** *Suppose  $\mathcal{P}$  is a CSP instance with  $s$  variables  $u_1, u_2, \dots, u_s$ . The following are equivalent:*

1.  $\phi = \{(u_1, a_1), (u_2, a_2), \dots, (u_s, a_s)\}$  is a solution to  $\mathcal{P}$ ,
2.  $\phi$  is an  $s$ -down-clique in  $MS(\mathcal{P}^+)$ , or
3.  $\phi$  is an  $s$ -independent set in  $CS(\mathcal{P})$ .

*Proof.* The proof is completely analogous to that of Proposition 4.11, with the focus on instance props of arbitrary arity instead of just pairs of literals. Two additional ingredients are required, first to recall Observation 2.41, that any subset of a prop is itself a prop, and second to recall Observation 4.9.  $\square$

It is also possible to weaken the notion of solution in a natural way. Consider the following optimisation problem.

**MAXIMUM PARTIAL CSP**

Input: CSP instance  $(S, T)$  in homomorphism representation  
Output:  $W \subseteq V(S)$ , function  $\phi: W \rightarrow V(T)$  forming a relational structure homomorphism from  $S[W]$  to  $T$   
Criterion: maximize  $|W|$ .

Stated differently, MAXIMUM PARTIAL CSP requires a largest partial homomorphism from  $S$  to  $T$ . A solution will clearly be a largest such partial homomorphism, but this formulation allows smaller partial homomorphisms that do not assign all the variables to values. It is therefore a way to capture “near-solutions”. The decision version of this problem is:

**PARTIAL CSP**

Input: positive integer  $k$ , CSP instance  $(S, T)$  in homomorphism representation  
Question: is there a subset  $W \subseteq V(S)$  of size at least  $k$ , and a relational structure homomorphism from  $S[W]$  to  $T$ ?

It is clear that Proposition 4.12 can be extended to apply to props instead of solutions. Props are just down-cliques in the microstructure of the explicit version of the instance, or independent sets in the clause structure.

**Corollary 4.13.** *Suppose  $\mathcal{P}$  is a CSP instance with  $s$  variables  $u_1, u_2, \dots, u_s$ . The following are equivalent:*

1.  $\phi = \{(u_1, a_1), (u_2, a_2), \dots, (u_s, a_s)\}$  is a prop of  $\mathcal{P}$ ,

2.  $\phi$  is an  $s$ -down-clique in  $MS(\mathcal{P}^+)$ , or
3.  $\phi$  is an  $s$ -independent set in  $CS(\mathcal{P})$ .

Note that the usual CSP decision problem for the homomorphism representation can be trivially transformed into PARTIAL CSP, by setting  $k = |V(S)|$ . It follows that PARTIAL CSP is NP-complete, and that MAXIMAL PARTIAL CSP is NP-hard.

Due to the way I have defined the microstructure, down-cliques in the infrastructure are solutions, but solutions do not have to be down-cliques in the microstructure. This is why the explicit version of the instance has to be used, to include the implicit anything-goes constraints in the microstructure for the purposes of checking for down-cliques. The microstructure may contain exponentially many partial assignments if the anything-goes constraints are included, and may thus be too large to manipulate. I will not concern myself with this issue: instead, for arity greater than 2 it is enough to observe that characterizing solutions as independent sets in the clause structure is a more useful transformation.

### 4.3 Examples

I now provide some examples of the microstructure representation.

**Example 4.14** (ternary CSP instance). This example illustrates the microstructure representation of a CSP instance with three variables and a single constraint of arity 3, presented in the variable-value representation:

$$(\{u, v, w\}, \{0, 1\}, \{((u, v, w), \{(0, 0, 1), (0, 1, 0), (1, 0, 0)\})\}).$$



Figure 4.3: Microstructure and clause structure of CSP instance in Example 4.14.

In the microstructure the highlighted prop is  $\{(u, 0), (v, 0), (w, 1)\}$ ; this is also a solution. In the clause structure the highlighted instance nogood is  $\{(u, 0), (v, 1), (w, 1)\}$ .  $\square$

**Example 4.15** (microstructure and clause structure of INJECTIVE). The INJECTIVE constraint was introduced in Example 2.29. There are two common ways to represent this constraint as a CSP instance; these both have the same infrastructures. (I postpone a proof of this to Proposition 5.8.) Using the definition, the instance props of  $\text{INJECTIVE}(u_1, u_2, \dots, u_n; D)$  are partial assignments  $\phi: \{u_1, u_2, \dots, u_n\} \rightarrow D$  which are injective functions. These props all have size  $n$ . The microstructure then consists of all of these instance props. If  $|D| \geq n$  then there are  $|D|!/(|D| - n)!$  such instance props while if

$|D| < n$  then the microstructure contains no instance props. The instance nogoods are the partial assignments of this form which are not injective. There are  $|D|^n - p$  instance nogoods, where  $p$  is the number of instance props. The clause structure is formed by these instance nogoods together with all binary variable clauses  $\{(u, a_1), (u, a_2)\}$  where  $a_1$  and  $a_2$  are distinct values in  $D$ . There are  $n \binom{|D|}{2}$  such variable clauses in the clause structure. Another representation, using binary constraints only, is discussed in Example 5.9.  $\square$

A unary constraint restricts the values that a variable may take.

**Example 4.16** (microstructure of ALL-DIFFERENT+UNARY). A unary constraint can be thought of as a removal of some values from the domain of the variable to which the constraint is applied. I refer to combinations of a single INJECTIVE and an arbitrary number of unary constraints as ALL-DIFFERENT+UNARY. This provides an appropriate model for the case when there is an ALL-DIFFERENT constraint specified over some scope, and during search some values are removed from the domains of some of these variables. For domain size  $d$ , the microstructure of ALL-DIFFERENT+UNARY has regular arity  $d$ , and consists of all  $d$ -tuples of literals, where the value of every element of the tuple is different. Further, there are unary constraints which restrict the values that some of the variables may take: effectively the unary constraints remove some literals from the microstructure, together with any edges that include those literals.

As a more specific example of this, let  $D = \{0, 1, 2\}$ . Figure 4.4 illustrates the microstructure of a particular instance of ALL-DIFFERENT:

$$\text{ALL-DIFFERENT}(u, v, w; D, D, D),$$

which has 3 variables each with domain  $\{0, 1, 2\}$ . This microstructure has 6 edges. The clause structure of this CSP instance has  $3^3 - 6 = 21$  edges as well as 9 additional variable clauses. The clause structure is not useful to illustrate. It is also not generally useful to illustrate larger examples. Sensibly drawing hypergraphs with a non-trivial number of edges is a long-standing challenge, and is beyond the scope of this thesis. Figure 4.5 illustrates an instance of ALL-DIFFERENT+UNARY:

$$\{\text{ALL-DIFFERENT}(u, v, w; D, D, D), \text{UNARY}(u; \{1, 2\})\}.$$

As with Figure 4.4, there are again 3 variables and a common domain of size 3, but in addition there is a unary constraint that restricts variable  $u$  to take only values 1 or 2. After removing the literal  $(u, 0)$  and the two edges that include it, the microstructure then has 4 edges. The clause structure has  $2 \times 3 \times 3 - 4 = 14$  edges and 7 additional variable clauses, and is not useful to illustrate.

The six hyperedges of the microstructure in Figure 4.4 are  $\{(u, 2), (v, 1), (w, 0)\}$ ,  $\{(u, 2), (v, 0), (w, 1)\}$  (this edge is highlighted),  $\{(u, 1), (v, 2), (w, 0)\}$ ,  $\{(u, 1), (v, 0), (w, 2)\}$ ,  $\{(u, 0), (v, 1), (w, 2)\}$ , and  $\{(u, 0), (v, 2), (w, 1)\}$ . The last two hyperedges are missing from

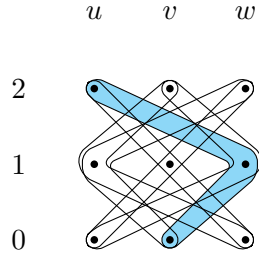


Figure 4.4: Microstructure of ALL-DIFFERENT with 3 variables and domain size 3.

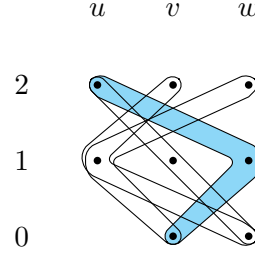


Figure 4.5: Microstructure of ALL-DIFFERENT+UNARY with  $(u, 0)$  and its edges removed.

the the microstructure in Figure 4.5, since they both include the literal  $(u, 0)$  which is forbidden by the unary constraint.  $\square$

In the following chapter I will consider a second way to capture ALL-DIFFERENT+UNARY, using binary constraints only.

In a realistic implementation, the microstructure of ALL-DIFFERENT+UNARY would not be represented as a number of tuples that is exponential in the square root of the instance size. Instead, it would be preferable to represent ALL-DIFFERENT+UNARY intensionally. In such a representation the microstructure would be a black box, in the form of an oracle that can be queried. For a given tuple, the oracle would indicate whether it is allowed by the ALL-DIFFERENT+UNARY constraint, by checking whether the input tuple satisfied the ALL-DIFFERENT constraint by checking the components pairwise, and then checking against the UNARY constraints in effect. This could be done with a number of comparisons that is at most cubic in the arity of the ALL-DIFFERENT constraint.

I next discuss one of the examples introduced in Chapter 2, the Sudoku problem. It is naturally modelled using unbounded arity constraints.

**Example 4.17** (microstructure of Sudoku). Each Sudoku instance consists of a grid of squares, with ALL-DIFFERENT constraints for each row, column, and block. An ALL-DIFFERENT constraint applied to  $n$  variables is a  $n$ -ary constraint, encoding the  $n$ -ary instance nogoods. In the next chapter I represent the ALL-DIFFERENT constraints as cliques of binary inequality constraints.

For  $9 \times 9$  Sudoku, the ALL-DIFFERENT constraints result in  $9! = 362,880$  instance props or  $9^9 - 9! = 387,057,609$  instance nogoods.

For generalized Sudoku, there are  $k^4$  squares, with  $k^2$  squares in each row, column, and block. This yields  $(k^2)!$  instance props and  $(k^2)^{k^2}$  instance nogoods. As  $(k^2)^{k^2} - (k^2)! > (k^2)!$  as long as  $k > 1$ , the microstructure always contains fewer edges than the clause structure.

Now  $k^2$  ALL-DIFFERENT constraints are required for each of the  $k^2$  rows,  $k^2$  columns, and  $k^2$  blocks, a total of  $3(k^2)(k^2) = 3k^4$  ALL-DIFFERENT constraints, or  $3k^4(k^2)!$  instance props. The size of the instance is  $2k^4 \log k$  in this case, and the microstructure contains  $3k^4(k^2)!$  partial assignments.



The microstructure has also been used in work on computer vision [93] and robotics [2, 3]. In such cases the underlying constraint satisfaction problem is implicit, being expressed as a system of mutually compatible assignments. I am not aware of any previous discussion of the implicit use of the microstructure in such problems. It may be possible to use the microstructure to make explicit the use of constraint satisfaction in applications that are currently regarded as unrelated to constraints; I leave this for further work.

## 4.4 Properties of the microstructure representation

The size of the microstructure of a binary CSP instance can be significantly larger than the size of the variable-value or homomorphism representation. However, the increase in size cannot change the tractability of the CSP once it is transformed to a problem involving down-cliques in the microstructure representation. A CSP is tractable with one of the other representations precisely when it is tractable in terms of the size of the microstructure for the transformed problem. If the CSP is NP-complete in one of the other representations, then it will remain so when the instance size is measured in terms of the microstructure for the transformed problem.

The size of the microstructure is closely related to the size of the original instance, as long as it is given extensionally (as is usual in the complexity-theoretic literature on CSPs). Unlike the infrastructure, the microstructure always has size that is polynomially related to the input size, if the instance description is one of the three standard ones I introduced in Chapter 2. With a given number  $s$  of variables in the instance, and at least 2 values for each variable, the infrastructure contains an exponential number of props (in  $s$ ), while most inputs of interest only have polynomially many instance props (in  $s$ ).

In the variable-value representation, there is a one-to-one correspondence between tuples that appear in constraint relations, instance props, and the edges of the microstructure. Each tuple in a constraint relation corresponds to a single edge of the microstructure, and every edge in the microstructure corresponds to a single tuple in some relation. This means that the microstructure representation is not significantly larger than the variable-value representation, so results about the complexity of CSPs in variable-value representation carry over to microstructures.

I now discuss some normalisations that are useful (and may sometimes be necessary), and how the microstructure representation is affected by combining constraints.

### 4.4.1 Normalisation

Two kinds of normalising transformations have the aim of simplifying how an instance is represented and how it is analysed.

First, by Proposition 3.9, if there are several different constraints with the same scopes or scopes that are permutations of each other, then they can be replaced by a single constraint on the same scope with its constraint relation being the intersection of the

corresponding relations (permuted, if necessary, to match any permutation applied to the associated scope). As is common practice, I will assume that this transformation is always performed [14].

Note that if this transformation is applied to a SAT instance (see Section 4.4.3), then the result is an instance of the so-called *generalized satisfiability* problem [142]. This is simply a CSP instance with a domain of size 2.

A second kind of normalisation is to remove unary constraints from an instance. Recall that a binary CSP instance is 1-consistent if there is no variable  $u$  and value  $a \in D_u \setminus \text{dom}(u)$ . This means that the unary constraints in a 1-consistent CSP instance do not change the instance in any way, and can be disregarded.

Enforcing 1-consistency can be extended to more involved kinds of local consistency, and I now discuss these. I base the following definition on the microstructure representation, rather than the usual (and equivalent) direct formulation [14, Definition 3.24].

**Definition 4.18** (generalized arc-consistency). A literal  $(u, a)$  is supported in a constraint  $(\sigma, R)$  if  $u$  occurs in  $\sigma$ , and there is a partial assignment  $\phi$  so that  $\phi(\sigma) \in R$  and  $\phi(u) = a$ . A CSP instance is **generalized arc-consistent** or **GAC** if every literal  $(u, a)$  in the microstructure is supported in every constraint.  $\square$

When ALL-DIFFERENT is applied to large scopes (in other words, its arity is large) then it is actually preferable in practice to maintain an *implicit* representation of the underlying disequality relation. It is possible to access the information encoded by the relation efficiently, while avoiding storing the large number of tuples. In particular, it is possible to maintain GAC efficiently [134, 64]. This insight forms the starting point of the theory and practice of global constraints.

Generalized arc-consistency is often encountered in its binary form. I now give its usual definition for completeness, but it can also be stated as a special case of GAC, or directly in terms of the microstructure.

**Definition 4.19** (arc-consistency). A binary CSP instance is **arc-consistent** if for every variable  $u$  and value  $a \in \text{dom}(u)$ , for every variable  $v$  distinct from  $u$  and constraint  $((u, v), R)$ , there is some  $b \in \text{dom}(v)$  such that  $(a, b) \in R$ .  $\square$

Removing unary constraints is usually done as part of the routines that enforce 1-consistency. Moreover, 1-consistency is usually performed at the same time as the process that enforces arc-consistency [63].

*Observation 4.20.* After normalisation, the microstructure and clause structure of a binary CSP instance will both be graphs.

#### 4.4.2 Microstructure representation of combined constraints

As discussed in Section 3.1.2, CSP instances are often constructed by combining separate constraints. The microstructure representation of a CSP instance can also be constructed by combining the microstructure representations of each separate constraint.

The **union** of hypergraphs  $G$  and  $H$  is the hypergraph  $(V(G) \cup V(H), E(G) \cup E(H))$  and is denoted  $G \cup H$ . The **intersection** of hypergraphs  $G$  and  $H$  is the hypergraph  $((V(G) \cap V(H), E(G) \cap E(H)))$  and is denoted  $G \cap H$ .

**Proposition 4.21.** *If  $\mathcal{P}_i = ((V, Q_i), (W, R_i))$  are CSP instances in the homomorphism representation for each  $i \in I$ , then the clause structure of the CSP instance  $\mathcal{P} = ((V, (Q_i)_{i \in I}), (W, (R_i)_{i \in I}))$  consists of the union of the clause structures of each of the  $\mathcal{P}_i$ 's:*

$$CS(\mathcal{P}) = \bigcup_{i \in I} CS(\mathcal{P}_i).$$

Further, the microstructure of  $\mathcal{P}^+$  is the intersection of the microstructures of the explicit versions of the  $\mathcal{P}_i$ 's:

$$MS(\mathcal{P}^+) = \bigcap_{i \in I} MS(\mathcal{P}_i^+).$$

*Proof.* First, suppose that  $\phi$  is an edge of  $CS(\mathcal{P})$ . Then this is an explicit nogood, and there must be some constraint of  $\mathcal{P}$  on the same variables as  $\phi$  that is violated by  $\phi$ . This constraint must occur in some  $\mathcal{P}_i$ , so  $\phi$  is an edge of  $CS(\mathcal{P}_i)$ . Hence  $\phi \in \bigcup_{i \in I} CS(\mathcal{P}_i)$ . For the reverse containment, it is clear that an edge that occurs in any of the clause structures of the constituent constraints corresponds to a clause, so it will be an edge of the microstructure of the combined instance.

The argument for the microstructure is analogous. Note that the microstructure of the explicit version contains the edges relating to anything-goes constraints between unconstrained variables, so if the variables in the prop of one instance are not constrained in any other instance then that will be a prop in the combined instance.  $\square$

This leads to the following formal justification for adding implicit anything-goes constraints when forming explicit instances.

**Corollary 4.22.** *If  $(V, D, C)$  is a CSP instance in variable-value representation, and  $\{u_1, u_2, \dots, u_r\} \subseteq V$  is a subset of variables that does not completely contain the set of variables of any constraint, then  $(V, D, C)^+ \equiv (V, D, C \cup *(u_1, u_2, \dots, u_r; D))^+$ .*

*Proof.* Let  $\mathcal{P} = (V, D, C)$ . If  $U$  does not completely contain the set of variables of any constraint, then any partial assignment with  $U$  as its set of variables must be a prop in the explicit version. Hence  $\text{prop}(*(u_1, u_2, \dots, u_r; D)^+) \subseteq \text{prop}(\mathcal{P}^+)$ . Now by Proposition 4.21,

$$\begin{aligned} \text{prop}((V, D, C \cup *(u_1, u_2, \dots, u_r; D))^+) &= \text{prop}(\mathcal{P}^+) \cup \text{prop}(*(u_1, u_2, \dots, u_r; D)^+) \\ &= \text{prop}(\mathcal{P}^+), \end{aligned}$$

as desired.  $\square$

#### 4.4.3 Clause structure as direct encoding

The clause structure corresponds quite closely to a SAT instance that is often associated with a CSP.

The direct encoding of an instance to SAT is used frequently in constraint programming [154]. I have used the term “clause structure” to emphasize the link between this form of the microstructure representation and the direct encoding, instead of the term “microstructure complement” which has also been used but which lacks this connotation [33].

One reason to prefer the clause structure to the microstructure, when this does not lead to an unacceptable increase in the size of the representation, is that it corresponds to the set of SAT clauses that is obtained by encoding the CSP instance in a straightforward way.

**Definition 4.23** (SAT instance). A CSP instance in the variable-value representation, with a set  $V$  of variables and a set  $D = \{0, 1\}$  of values, with each constraint relation of the form  $D^r \setminus \{v\}$  for some  $v \in D^r$ , is called a **SAT instance**.

Constraint  $(\sigma, D^r \setminus \{v\})$  indexed by  $I$  is usually called a **SAT clause** over the variables in the underlying set of  $\sigma$ , and is written as  $\bigvee_{i \in I} \lambda_i$ , where  $\lambda(i) = \sigma(i)$  if  $v(i) = 0$  and  $\lambda(i) = \sigma(i)'$  if  $v(i) = 1$ . For each variable  $u \in V$ , the expression  $u$  is called a **positive literal** and  $u'$  is called a **negative literal**.  $\square$

In a SAT instance, each constraint forbids precisely one of the possible assignments to the scope. SAT instances therefore often have multiple constraints on the same scope.

**Example 4.24** (SAT instance). Let  $D = \{0, 1\}$ . The SAT instance

$$( \{x, y, z\}, D, \{ ((x, y), D^2 \setminus \{(1, 1)\}), ((x, y), D^2 \setminus \{(1, 0)\}), ((z), D^1 \setminus \{(1)\}) \} )$$

is usually written as  $(x' \vee y') \wedge (x' \vee y) \wedge (z')$ . This is equivalent to the CSP instance (in the variable-value representation)

$$( \{x, y, z\}, D, \{ ((x, y), \{(0, 0), (0, 1)\}), ((z), \{(0)\}) \} ) . \quad \square$$

This suggests one way to transform any CSP instance into a SAT instance. The SAT instance uses a variable for each literal  $(u, a)$  in the CSP instance, representing whether  $u$  is assigned a value other than  $a$ . Each instance nogood  $\phi$  indexed by  $U \subseteq V$  is represented by a SAT clause  $\bigvee_{u \in U} (u, \phi(u))$ , and therefore captures the requirement that any prop must assign a value other than  $\phi(u)$  to some variable  $u$ . However, the usual semantics of SAT are different to the semantics of CSP instances.

First, every variable in a CSP instance can only be assigned a single value (as partial assignments are functions). This requirement is not represented by the SAT clauses representing the instance nogoods. Several variables in the SAT instance represent a single CSP variable; for  $a \neq a'$  it may be possible to set  $(u, a)$  false and also to set  $(u, a')$  false, yet this would mean that CSP variable  $u$  is assigned both value  $v$  and value  $v'$ . The

requirement that CSP variables can only take single values can be enforced by adding a variable clause for each variable.

Second, by setting each SAT variable corresponding to  $(u, a)$  true, the SAT clauses representing the instance nogoods would then consist of disjunctions of true propositions, as would the variable clauses. Hence all these clauses would be trivially satisfied, yielding a solution to the SAT instance, yet such a solution would require a solution to the original CSP instance to set each variable to a value outside its domain which violates the semantics of the CSP instance. It is therefore important to explicitly capture the requirement that  $u \in D_u$  for each variable  $u$  in the CSP instance; this can be done by adding domain clauses for each variable.

**Definition 4.25** (domain clause). A **domain clause**  $\phi$  for a variable  $u$  is a set of literals  $\{(u, a) \mid v \in D_u\}$  representing a proposition of the form  $\bigvee_{a \in D_u} (u = a)$ . The domain clauses of a CSP instance consist of the union of all domain clauses for each variable  $u$  in the instance.  $\square$

This leads to a definition of the direct encoding, which includes variable and domain clauses in addition to clauses representing each instance nogood.

**Definition 4.26.** The **direct encoding** of a CSP instance  $\mathcal{P} = (V, D, C)$  in the variable-value representation is the related SAT instance  $\text{SAT}(\mathcal{P})$ , in which there is a variable for each literal  $(u, a)$ , for every variable  $u \in V$  and every value  $v \in D$ .

The SAT variable corresponding to  $(u, a)$  represents the truth value of the statement  $(u \neq a)$ . Each instance nogood  $\phi$  of  $\mathcal{P}$  indexed by  $U \subseteq V$  is represented by a SAT clause  $\bigvee_{u \in U} (u, \phi(u))$  that forbids that specific partial assignment: this SAT clause is true when  $\bigvee_{u \in U} (u \neq \phi(u))$ . The direct encoding also includes all variable clauses to ensure that each variable of the CSP instance is not assigned to more than one value, as well as all domain clauses to ensure that each variable of the CSP instance is assigned at least one value in its domain.  $\square$

The domain clause for a variable  $u$  requires  $u$  to be assigned at least one of the values in  $D$ . This captures an important aspect of the semantics of a CSP instance, in SAT terms.

It follows that the clause structure of a CSP instance  $\text{CS}(\mathcal{P})$  and the SAT instance obtained by the direct encoding  $\text{SAT}(\mathcal{P})$  are closely related. The major difference is the presence of domain clauses in  $\text{SAT}(\mathcal{P})$ , which in the clause structure are instead captured by explicitly requiring every variable to be assigned to some value in its domain. This is discussed in the following section.

**Example 4.27** (implicit nogood). Let  $R(x, y) \equiv x \vee y$  and  $S(y) \equiv \neg y$  be Boolean relations. Let  $\phi(x, y) = R(x, y) \wedge S(y)$  be a Boolean formula in conjunctive normal form. Consider the CSP instance in first-order logic representation

$$\exists x, y \phi(x, y)$$

which is just the SAT instance  $(x \vee y) \wedge (y')$ . The only solution to this instance is  $\{(x, 1), (y, 0)\}$ . Therefore the prop  $\{(x, 0)\}$  which in isolation does not violate any constraint, is also a nogood. It is an implicit nogood since there is no constraint with scope containing just the variable  $x$ .  $\square$

#### 4.4.4 Product form

Recall from Definition 3.24 that  $=_X$  denotes the equality relation on a set  $X$ .

**Definition 4.28** (augmented relational structure). For any relational structure  $S = (V(S), (Q_i)_{i \in I})$ , let  $S' = (V(S), ((Q_i)_{i \in I}, =_{V(S)}))$  be the structure augmented with the equality relation on  $V(S)$ .  $\square$

The notation in Definition 4.28 was discussed in Definition 2.3. Note that Definition 4.28 implies that the structure  $S'$  will always contain one more relation than  $S$ , even when  $S$  already contains an equality relation.

**Example 4.29** (augmented structure).

1.  $(V, E)' = (V, (E, =_V))$ .
2. Suppose  $I'$  is obtained from  $I$  by adding a new element  $k \notin I$ , and let  $Q_k$  be the relation  $=_V$ . Then  $(V, (Q_i)_{i \in I})' = (V, (Q_i)_{i \in I'})$ .  $\square$

I now formally define products of relations and products of relational structures.

**Definition 4.30** (product of relations). For relations  $Q$  and  $R$  of the same arity  $r$ , the **product** of  $Q$  and  $R$  is

$$Q \times R = \{((x_i, y_i))_{i \in [r]} \mid (x_i)_{i \in [r]} \in Q, (y_i)_{i \in [r]} \in R\}.$$

The product of  $Q$  and  $R$  is the set of tuples formed by pairs of vertices, where each component of a tuple is derived from the corresponding components in a pair of tuples, one from  $Q$  and one from  $R$ .

Note that the product of two relations of arity  $r$  will again be a relation of arity  $r$ , but it will be over a set of ordered pairs. This notion can be applied componentwise to relational structures of the same signature.

**Definition 4.31** (product of relational structures). Given two relational structures  $S = (V(S), (Q_i)_{i \in I})$  and  $T = (V(T), (R_i)_{i \in I})$  with the same signature, the **product** of  $S$  and  $T$  is

$$S \times T = (V(S) \times V(T), (Q_i \times R_i)_{i \in I}),$$

the component-wise product.  $\square$

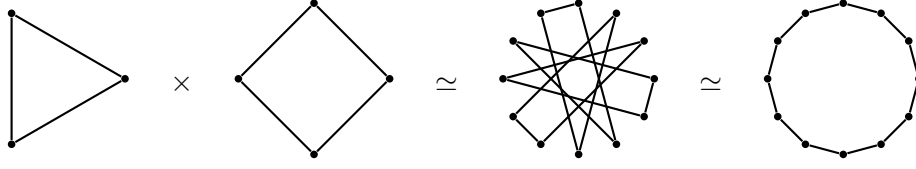


Figure 4.6:  $C_3 \times C_4 \simeq C_{12}$

For the special case of binary relations  $Q$  and  $R$ , Definition 4.30 can be stated as

$$Q \times R = \{((x_1, y_1), (x_2, y_2)) \mid (x_1, x_2) \in Q, (y_1, y_2) \in R\},$$

the set of tuples formed by pairs of vertices, where each component of a tuple is derived from the corresponding components in a pair of tuples, one from  $Q$  and one from  $R$ .

**Example 4.32** (product of undirected cycles). Let  $S = C_k$  and  $T = C_{k+1}$ . Then

$$S \times T = \{((i, p), (j, q)) \mid (i, j) \in C_k, (p, q) \in C_{k+1}\}.$$

In particular,  $\{(k-1, k), (0, 0)\} \in E(S \times T)$ ; the two cycles are joined into a single cycle in the product.

See Figure 4.6 for an illustration of the product of undirected cycles.  $\square$

The product of two relational structures  $S$  and  $T$  has the same signature as the structures, but its vertices are ordered pairs of the vertices of  $S$  and  $T$ .

**Proposition 4.33.** *For a CSP instance  $\mathcal{P}$  with arity  $r$ , the  $r$ -section of the complement of the microstructure of  $\mathcal{P}^+$  is the clause structure of  $\mathcal{P}$ , and the  $r$ -section of the complement of the clause structure of  $\mathcal{P}$  is the microstructure of  $\mathcal{P}^+$ .*

*Proof.* Clearly the vertices of the microstructure and clause structure of  $\mathcal{P}$  are the same, and the vertices are not affected by taking complements nor by constructing the explicit version.

Suppose  $\phi$  is an edge of  $\overline{\text{MS}(\mathcal{P}^+)}$  of cardinality at most  $r$ . Then  $\phi$  is not a prop of  $\mathcal{P}^+$ , and since  $\text{MS}(\mathcal{P})$  is a subhypergraph of  $\text{MS}(\mathcal{P}^+)$ , also not a prop of  $\mathcal{P}$ . If no constraint of  $\mathcal{P}$  has a scope with the same variables as  $\phi$ , then there must be an anything-goes constraint in  $\mathcal{P}^+$  that forbids  $\phi$ , a contradiction. Hence there is a constraint of  $\mathcal{P}$  with the same variables as  $\phi$ . As  $\phi$  is not a prop, it must violate this constraint, so  $\phi$  is an edge of  $\text{CS}(\mathcal{P})$ .

Now suppose that  $\phi$  is an edge of  $\text{CS}(\mathcal{P})$ . Then  $\phi$  violates some constraint  $(\sigma, R)$  of  $\mathcal{P}$ . Therefore the sets of variables of  $\phi$  and of  $\sigma$  are the same, and  $|\phi| = \rho(\sigma)$ . For sake of the argument, suppose that  $\phi$  is an edge of  $\text{MS}(\mathcal{P}^+)$ . Since  $\phi$  cannot originate from some constraint of  $\mathcal{P}$ , it must originate from some all-different constraint added when creating the explicit version of  $\mathcal{P}$ . However, the explicit version only adds anything-goes constraints with scopes that do not already occur in  $\mathcal{P}$ , contradicting that the variables

of  $\phi$  and  $\sigma$  are the same. Hence  $\phi$  is not an edge of  $\text{MS}(\mathcal{P}^+)$ , and since its cardinality is at most as large as the arity of any scope of  $\mathcal{P}^+$ , it must be an edge of the  $r$ -section of  $\overline{\text{MS}(\mathcal{P}^+)}$ .

The second identity follows from the first by taking hypergraph complements and enforcing the cardinality bound based on arity.  $\square$

When the CSP instance is specified in the homomorphism representation, Proposition 4.33 provides a nice algebraic characterization of the microstructure and its complement. When the instance is not given in homomorphism representation, some subtleties may arise. If variables may have domains that have different numbers of elements as discussed in Section 3.1.3, then during the transformation to homomorphism representation the domain of the target structure will be as large as the largest domain. The clause structure will in this case become unnecessarily large. Note that the microstructure will not contain any unnecessary edges in this case; the additional new literals will simply not appear in any edge of the microstructure, so will be discarded when arc-consistency is enforced.

The microstructure and its complement can now be characterized algebraically as the hypergraph of a product of relational structures. Recalling Definition 2.25, the complement of a relational structure  $S$  is denoted  $\overline{S}$ .

**Proposition 4.34.** *The clause structure of a CSP instance  $(S, T)$  in homomorphism representation consists of the hypergraph of the relational structure  $S' \times \overline{T'}$ .*

*Proof.* Clearly  $V(S') = V(S)$  and  $V(\overline{T'}) = V(T)$ , so  $V(S' \times \overline{T'}) = V(S) \times V(T)$ .

First consider the vertices of the clause structure. These are partial assignments of size 1, so are elements of  $V(S) \times V(T)$ . Now consider an element of  $V(S) \times V(T)$ . This corresponds to every possible assignment of a value from  $V(T)$  to a variable from  $V(S)$ . Hence the vertices of the clause structure are precisely the same as the vertices of hypergraph  $\mathcal{G}(S' \times \overline{T'})$  of  $S' \times \overline{T'}$ .

Now consider some binary clause  $\{(u, a), (v, b)\}$  in the clause structure. This must then be an explicit nogood. If  $u = v$  then  $a \neq b$  and then  $((u, a), (u, b))$  is in the part of the product structure formed by the relations involved in the augmentation of  $S$  and  $T$ . So suppose  $u \neq v$ , when  $(u, v)$  is a scope (or  $(v, u)$  is, in which case the argument holds by symmetry), and  $(a, b)$  does not appear in the corresponding relation. Hence  $((u, a), (v, b))$  is a tuple in the product structure  $S' \times \overline{T'}$ .

Finally, consider some binary edge  $\{(u, a), (v, b)\}$  of the graph of the product structure. If this edge is an edge corresponding to the relations introduced by augmentation, then  $u = v$  and  $a \neq b$ . The set of literals in the edge therefore is a nogood, since no solution can assign different values to the same variable. The other case is that this edge must correspond to a scope  $(u, v)$  and an edge  $(a, b)$  that is not in the corresponding relation of  $T$ . This means that  $a$  can never be assigned to  $u$  at the same time as  $b$  is assigned to  $v$ , so  $\{(u, a), (v, b)\}$  is an explicit nogood.



For the ternary case, there is a clause  $\{(u, a), (v, b), (w, c)\}$  in the clause structure. If any of  $u, v, w$  were the same then by the assumption that variables cannot repeat in a scope, this clause must have been obtained from the augmentation of  $S$  and  $T$ ; however, this is not possible, as these are all binary clauses. It then follows that  $u, v, w$  are three distinct variables. Hence the clause is an explicit nogood, so there is some scope with set  $\{u, v, w\}$ . Without loss of generality, assume this scope is  $(u, v, w)$ . Then  $((u, a), (v, b), (w, c))$  is a tuple in the product structure  $S' \times \overline{T'}$ . In the other direction, if  $\{(u, a), (v, b), (w, c)\}$  is an edge in the hypergraph of the product structure  $S' \times \overline{T'}$ , then again this cannot have arisen from the augmentation of  $S$  and  $T$ , and there must be a tuple, without loss of generality  $(u, v, w)$ , in a relation of  $S$  and a tuple  $(a, b, c)$  that is missing from the corresponding relation of  $T$ . Hence  $\{(u, a), (v, b), (w, c)\}$  is an explicit nogood, and therefore in the clause structure.

The case of larger edges (with more than three literals) is essentially the same as the ternary case. Since by assumption no repeated scopes occur in the instance, this completes the proof.  $\square$

Recalling Definition 2.12, the complement of a hypergraph  $G$  is denoted  $\overline{G}$ .

The following is then an immediate consequence of Proposition 4.33 and Proposition 4.34.

**Corollary 4.35.** *The microstructure of the explicit version of a CSP instance  $(S, T)$  in homomorphism representation of arity  $r$  consists of the  $r$ -section of the hypergraph of the relational structure  $\overline{S' \times \overline{T'}}$ .*

In light of Proposition 4.34 and Corollary 4.35, it is therefore appropriate to consider the **complement** of the microstructure of the explicit version of the instance to be the clause structure, and vice versa.

However, I will not make much use of Corollary 4.35 for high arity CSP instances, since if  $H$  is a hypergraph on  $s$  vertices with  $q$  edges, then its complement  $\overline{H}$  will contain  $2^s - 1 - q$  edges, and even with the arity  $r$  restriction as in Corollary 4.35, the arity will occur as the degree of the polynomial enumerating the number of edges. Such large structures are neither useful for analysis of computational complexity, nor for practical applications. Corollary 4.35 does sometimes yield reasonable results for CSP instances with small arity (such as binary CSP instances), but even then the overhead of creating the explicit version of the instance can be unacceptably high.

**Example 4.36** (Clause structure of  $k$ -CLIQUE). The clause structure of an instance  $G = (V, E)$  of  $k$ -CLIQUE is  $([k], (\neq_k, =_k)) \times (V, (\overline{E}, \neq_V)) = ([k] \times V, (\neq_k \times \overline{E}, =_k \times \neq_V))$ .

Regarding either form of the microstructure representation as a kind of product, as in Proposition 4.33 and Corollary 4.35, may have implications that rule out some natural approaches to defining tractable classes by means of low width. I leave this for future work (see Section 8.2.2).

## 4.5 Context and summary

The concept of the microstructure of a binary CSP instance was introduced informally to the constraints literature in 1991 by Freuder [58]. This was made more precise by Jégou, who also showed that finding cliques in the microstructure corresponds to finding solutions [97]. The reformulation in terms of independent sets in the clause structure was made explicitly a few years later by Weigel and Bliet [155] and by Cohen [34]. Clause structures as hypergraphs have been used to rigorously define symmetries in CSP instances [33]. The clause structure as a system of nogoods is also used in the work on generalized resolution by Kullmann and by Mitchell [107, 121].

The distinction between open-world and closed-world assumptions has been an important part of database theory for decades [89]. I am not aware of these notions previously having been linked to constraint satisfaction.

The binary version of the MAXIMUM PARTIAL CSP optimisation problem was investigated by Ambler et al., motivated by robot motion planning [3, Section 6.4]. Originally the problem was specified in terms of maximal cliques in the binary microstructure. Transforming a constraint satisfaction problem into the problem of finding a large clique in the microstructure is therefore an idea with a long history, going at least as far back as 1973 [2]. However, note that the underlying constraint satisfaction problem was not made explicit in these early papers.

Kozen defined a product of two graphs which produces the microstructure of the graph isomorphism problem, when the graphs have the same number of vertices [106]. Kozen's construction is elsewhere called the modular product [92]. The modular product is not used in this thesis, but the product construction of a microstructure is conceptually similar.

For relational structures that are graphs, in other words that contain a single symmetric binary relation, the notion that a relational structure homomorphism corresponds precisely to an independent set of sufficient size in a derived product structure was apparently mentioned by Hell in his PhD thesis in 1972. This correspondence is also mentioned in the 2004 book of Hell and Nešetřil [83, Exercise 2.7]. For the special case of  $t$ -colouring of graphs, Chvátal constructed the binary clause structure as a technical tool [30].

In this chapter I have shown that this correspondence holds also in the general arity case, with suitable definitions of independent set and clique in hypergraphs. Specifically, in Proposition 4.12 I have shown that the existence of a solution of a CSP corresponds precisely to finding a large enough independent set in the clause structure, and that this holds for any CSP instance, not just for binary instances with a single relation. Moreover, Proposition 4.34 characterized the clause structure as a product.

[Mr. A. B. Kempe] makes diagrams of spots connected by lines; and it is easy to prove that every possible system of relationship can be so represented . . . He thus represents every possible relationship by a diagram consisting of only two different kinds of elements, namely, spots and lines between pairs of spots.

—C. S. Peirce, *The Critic of Arguments*, 1892. §423.

# 5

## Hereditary classes of binary microstructures

This chapter continues the microstructure representation of constraint satisfaction problems. In this chapter I examine the case when each instance in a problem is binary, so all constraints relate to either single variables or to pairs of variables. The microstructure representation of a binary CSP instance can then be regarded as a graph.

I explore the tension between binary instances and those of arbitrary arity. Any CSP instance can be transformed to a binary instance. This transformation may introduce a significant number of new variables, required to maintain equivalence between the original instance and its binary version. The new variables may also hide structure that is inherent in the problem. I discuss the transformation of CSP instances to binary instances, and the well-behaved case of conformal CSP instances where this transformation can be done efficiently and in a straightforward manner.

I consider the size of the microstructure representation of binary instances, and how this affects the computational complexity of a class of instances.

I then discuss decomposition of constraints.

I then define hereditary CSPs, which are classes of CSPs that are closed under induced subproblems: if an instance is in the CSP, then so is any induced subproblem of that instance. Such CSPs can be obtained by requiring that if an instance is in the CSP, then so is any instance obtained by deleting literals from its microstructure representation. An equivalent definition is by means of a set of forbidden substructures in the microstructures or clause structures of the instances of the CSP.

For binary instances, it is possible to apply many existing results about hereditary families of graphs.

In a binary CSP instance, finding a solution corresponds to finding a sufficiently large

independent set in the clause structure, which is just a graph. For many hereditary classes of graphs, it is known either that Independent Set is NP-complete or that it admits a polynomial-time algorithm. Perfect graphs form an important hereditary class which admits polynomial-time algorithms. I show that the microstructures of several previously studied classes of binary CSPs form classes of graphs that are strictly contained in the class of perfect graphs. These classes form a lattice, and I classify these classes of CSPs in terms of this lattice. Several previously published results follow as special cases of this classification. In this chapter, I also tighten the results which I published previously with Peter Jeavons.

I explain why hereditary classes are important for the study of constraint satisfaction, and show that hereditary classes correspond to forbidden substructures in the microstructure. After discussing classes of graphs excluding all single graphs of small size, and the computational complexity of these classes for the INDEPENDENT SET problem, I then look at how these classes generalize to form CSPs that forbid certain subgraphs in their microstructure representation. This leads to some complexity results and a discussion of open problems. Hereditary CSPs can be combined to form new CSPs in various ways.

I then provide some examples of non-binary constraint satisfaction problems. For some of these problems it is possible to decompose the constraints into binary constraints so that the microstructure representation becomes a graph. Some of the examples in the previous chapter occur again here. Other problems cannot be stated in binary form without introducing new variables.

I close this chapter with a detailed lattice of classes of CSPs which have perfect graphs as microstructures. This lattice shows how these classes are related and provides more precise characterization of the classes of microstructures of several published classes of CSPs. One of the hereditary classes in the lattice leads to a unification of the ALL-DIFFERENT+UNARY and tree structured CSPs through both of these quite different-seeming constraints having their microstructures being *perfect* graphs.

## 5.1 Transforming to the binary microstructure

I now discuss how a CSP instance can be transformed to another equivalent representation with only binary constraints. Several different methods to do this have been considered in the literature, and I will survey these. After normalisation, the microstructure representation of a binary CSP instance will be a graph.

When the CSP instance is specified in the homomorphism representation, there is a nice algebraic characterization of the microstructure and its complement. When the instance is not given in homomorphism representation, some subtleties may arise. If variables may have domains that are of different sizes, then during the transformation to homomorphism representation the domain of the target structure will be as large as the largest domain. The clause structure will in this case become unnecessarily large. Note that the microstructure

will not contain any unnecessary edges in this case; the additional new literals will simply not appear in any edge of the microstructure, so will be discarded when arc-consistency is enforced.

Any CSP can be reduced to one with lower arity. One way to do this is by lumping together variables into blocks, with new variables defined over a larger domain representing each block, and defining new binary relations between blocks that maintain the underlying variable structure. The blocks can be made as large as desired, in the extreme case any CSP instance can be made into a CSP containing only unary relations together with the new binary relations between blocks.

Removing unary relations by applying 1-consistency, the result is a binary CSP.

There are also other ways to transform a CSP into binary form, for instance the dual transformation, but these do not usually preserve the structure of the instance. The structure in a CSP can sometimes be exploited to obtain polynomial-time algorithms. An important reason to avoid transforming CSPs into binary form is that the transformations above will usually hide the structure of the non-binary CSP, and often transforming even a tractable CSP into a set of instances that is not obviously a tractable CSP.

### 5.1.1 Global constraints

**Definition 5.1.** A class of CSP instances  $\mathcal{C}$  has **bounded arity** if there is a positive integer  $r$  such that every instance in  $\mathcal{C}$  has arity at most  $r$ .

The possible presence of global constraints in the constraint language of a class of CSP instances is the reason that it is not sufficient to look at constraints up to some fixed arity.

Recall from Chapter 2 that a global constraint is a constraint which may be applied to any subset of the variables, and which therefore does not have an upper bound on its arity. Global constraints are typically applied to all variables in some part of the problem that is allowed to grow with instance size. Global constraints are regarded as completely natural in constraint programming, and this provides an important motivation for studying CSPs which do not necessarily have bounded arity.

The ALL-DIFFERENT constraint can be expressed equivalently (or **decomposed**) as a conjunction of separate binary constraints, so these problems can be easily reformulated so that every instance has the same signature. However, when a global constraint is decomposed there is generally a loss of performance: using binary constraints can produce an equivalent instance but is less efficient than using global constraints which can often be dealt with by special-purpose routines that exploit their global nature [14, 62].

Other constraints (such as individual SAT clauses) cannot be decomposed into an equivalent conjunction of a small number of constraints of bounded arity, unless additional variables are introduced [122]. Many global constraints can be decomposed by introducing a small number of new variables to capture the semantics of the global constraint [143].

It is always possible to decompose the constraints in the problem into constraints of smaller arity, by introducing some number of additional variables. However, in general this

comes at the cost of each new variable having a domain that is exponential in the arity of the relations it represents. A relation  $R$  of arity  $r$  can be represented by means of  $r$  binary relations involving a new hidden variable that takes as its values the tuples of  $R$ .

The following important example is already binary but its microstructure will be useful in Chapter 6.

**Example 5.2** (microstructure of  $t$ -COLOURING).  $t$ -COLOURING was introduced in Example 3.28. The natural way to represent this as a constraint satisfaction problem is by letting  $T$  be a  $t$ -clique, when  $(S, T)$  is a CSP instance in homomorphism representation that has a solution if, and only if,  $S$  can be  $t$ -coloured.

The microstructure of  $(S, T)$  contains all literals  $(u, c)$  corresponding to vertex  $u$  being coloured  $c$ , where  $u$  ranges over all elements of  $V(S)$  and  $c$  over  $[t]$ . The pair of distinct literals  $(u, c)$  and  $(v, d)$  are adjacent if, and only if, either  $u = v$  and  $c \neq d$  (representing the requirement that every variable be assigned at most one value), or  $\{u, v\} \in E(S)$  and  $c \neq d$ , or  $\{u, v\} \notin E(S)$ .  $\square$

Three straightforward transformations of CSP instances to binary form have been considered in the literature. These are the exact 2-section, the hidden variable transformation, and the dual and hidden transformations via the incidence graph.

### 5.1.2 Exact 2-section

A constraint that can be decomposed into equivalent binary constraints with no additional variables is called **network decomposable** [62]. I now formalize this notion in terms of conormal microstructures.

Recall Definition 2.13. If singleton edges of the 2-section are removed, the resulting hypergraph is a graph. The 2-section with such singleton edges removed is just the primal or Gaifman graph of the hypergraph (recall Definition 2.14). The 2-section of the microstructure then leads naturally to the exact 2-section of a CSP instance.

**Definition 5.3** (exact 2-section of CSP instance). The **exact 2-section** of a CSP instance  $\mathcal{P}$  is a binary CSP instance  $\mathcal{P}_2$ , with the same variables and values as  $\mathcal{P}$ , and whenever  $(\sigma, R)$  is a constraint of  $\mathcal{P}$  and  $u_{i_1}$  and  $u_{i_2}$  occur as distinct variables of  $\sigma$ , then  $(\sigma_{\{i_1, i_2\}}, R_{\{i_1, i_2\}})$  is a constraint of  $\mathcal{P}_2$ .  $\square$

The original instance is a subproblem of its exact 2-section [139]. (Recall the definition of subproblem in Definition 3.3.) For completeness I now prove that the exact 2-section always preserves solutions.

**Proposition 5.4** (exact 2-section). *Every CSP instance  $\mathcal{P}$  is a subproblem of its exact 2-section.*

*Proof.* Let  $\phi$  be a partial assignment in the infrastructure of  $\mathcal{P}$ . If  $|\phi| \leq 2$  then  $\phi$  is consistent with the constraints and, recalling Observation 2.41, will also be consistent

with any  $|\phi|$ -subset of every constraint. Hence  $\phi$  is in the infrastructure of  $\mathcal{P}_2$ . Now suppose  $|\phi| \geq 3$ . Suppose that  $\phi$  violates some constraint  $(\sigma, R)$  of  $\mathcal{P}_2$ . Since  $\mathcal{P}_2$  is binary, the scope can be written as  $\sigma = \{(u_i, u_j)\}$  for some variables  $u_1, u_2$ . By Observation 3.2 the infrastructure is downward-closed, so the partial assignment  $\phi' = \phi_{\{i,j\}}$  of  $\phi$  projected to the variables in  $\sigma$  is a binary edge of the infrastructure of  $\mathcal{P}_2$ . Hence  $(\phi'(u_i), \phi'(u_j)) = (\phi(u_i), \phi(u_j)) \in R$ , which contradicts the assumption that  $\phi$  violates this constraint. Hence  $\phi$  does not violate any constraint of  $\mathcal{P}_2$ , so is in the infrastructure of  $\mathcal{P}_2$ .  $\square$

Although the transformation to the exact 2-section preserves solutions, it may result in a CSP instance of which the original is a strict subproblem [122]. The following example illustrates that the exact 2-section may lead to additional solutions that were not present in the original instance.

**Example 5.5** (instance is strict subproblem of exact 2-section). Consider the CSP instance with microstructure in Figure 5.1. There are three hyperedges, each allowing a triple with

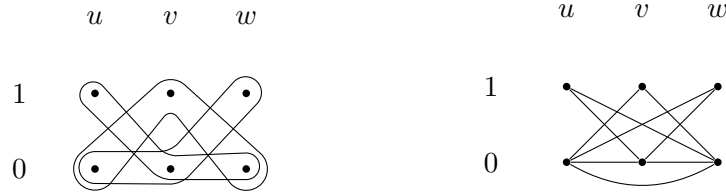


Figure 5.1: CSP instance with three solutions, but its exact 2-section has an additional solution.

a single 1 value and two 0 values; each of these is a solution. However, its exact 2-section also has the solution assigning 0 to each variable.  $\square$

A hypergraph is **conformal** when every clique of its Gaifman graph is contained in some edge of the hypergraph. The following results then follow from the definitions.

**Proposition 5.6.** *The Gaifman graphs of  $\text{prop}(\mathcal{P})$  and  $\text{prop}(\mathcal{P}_2)$  coincide.*

*Proof.* By Proposition 5.4,  $\text{prop}(\mathcal{P})$  is a subhypergraph of  $\text{prop}(\mathcal{P}_2)$ . The Gaifman graph of  $\text{prop}(\mathcal{P})$  is therefore a subgraph of the Gaifman graph of  $\text{prop}(\mathcal{P}_2)$ . What remains to be shown is that whenever  $\{(u_1, v_1), (u_2, v_2)\}$  is an edge of the Gaifman graph of  $\text{prop}(\mathcal{P}_2)$ , then there is some  $\text{prop } \phi \in \text{prop}(\mathcal{P})$  such that  $\phi(u_1) = v_1$  and  $\phi(u_2) = v_2$ . This follows directly from the definition of  $\mathcal{P}_2$ .  $\square$

Note that  $\text{prop}(\mathcal{P}_2)$  is conformal. Further, the conformality of  $\text{prop}(\mathcal{P})$  is directly related to whether  $\text{prop}(\mathcal{P}) \equiv \text{prop}(\mathcal{P}_2)$ .

**Proposition 5.7.** *A CSP instance  $\mathcal{P}$  is infrastructure equivalent to its exact 2-section if, and only if,  $\text{prop}(\mathcal{P})$  is conformal.*

*Proof.* Suppose  $\mathcal{P} \equiv \mathcal{P}_2$ . Let  $\phi$  be a clique in the Gaifman graph of the infrastructure of  $\mathcal{P}$ . By Proposition 5.6,  $\phi$  is also a clique in the Gaifman graph of the infrastructure of  $\mathcal{P}_2$ . As  $\text{prop}(\mathcal{P}_2)$  is conformal, there is some  $\phi \in \text{prop}(\mathcal{P}_2) = \text{prop}(\mathcal{P})$  such that  $\phi \subseteq \phi$ . It follows that  $\text{prop}(\mathcal{P})$  is conformal.

Now suppose that  $\text{prop}(\mathcal{P})$  is conformal. By Proposition 5.4, each edge of  $\text{prop}(\mathcal{P})$  is also an edge of  $\text{prop}(\mathcal{P}_2)$ . Now suppose  $\phi \in \text{prop}(\mathcal{P}_2)$ . Then  $\phi$  is a clique in the Gaifman graph of  $\mathcal{P}_2$ , and by Proposition 5.6, also in the Gaifman graph of  $\mathcal{P}$ . As  $\text{prop}(\mathcal{P})$  is conformal, there is some  $\phi \in \text{prop}(\mathcal{P})$  such that  $\phi \subseteq \phi$ . By Observation 3.2 it then follows that  $\phi \in \text{prop}(\mathcal{P})$ . Hence  $\text{prop}(\mathcal{P}_2) = \text{prop}(\mathcal{P})$ .  $\square$

The conformality criterion of Proposition 5.7 provides a useful test for checking whether a CSP instance can be equivalently represented by its exact 2-section: if the infrastructure is conformal then the exact 2-section forms an equivalent instance.

Some global constraints can be easily decomposed into binary constraints. An important example where the exact 2-section does lead to an equivalent instance is the INJECTIVE constraint.

**Proposition 5.8** (decomposition of INJECTIVE). *The INJECTIVE constraint is equivalent to its exact 2-section.*

*Proof.* By Proposition 5.7 it is enough to show that the infrastructure is conformal. Suppose  $\phi$  forms a clique in the Gaifman graph of the infrastructure. I claim that  $\phi$  is a partial assignment, and moreover is consistent with every constraint.

Suppose that  $(u_1, v_1)$  and  $(u_2, v_2)$  are distinct assigning literals in the Gaifman graph of the infrastructure. If they are adjacent then  $u_1 \neq u_2$  and  $v_1 \neq v_2$  since each pair of adjacent vertices must occur together in some injective complete assignment. This is true of every pair of adjacent vertices, so  $\phi$  must be an injective partial assignment.  $\square$

**Example 5.9** (binary microstructure and clause structure of INJECTIVE). The INJECTIVE constraint was introduced in Example 2.29 and its microstructure discussed in Example 4.15. I now discuss the binary representation of this constraint. This is equivalent to the representation in Example 4.15. The clause structure of this constraint contains two kinds of cliques. The first is a clique of literals for each value  $a \in D$  in the domain, of the form  $\{(u, a) \mid u \in V\}$ . These nogoods (represented as horizontal cliques in Figure 5.2) force different values to be assigned to each variable. There are also variable clauses: for each variable  $u \in V$ , these are of the form  $\{(u, a) \mid a \in D\}$ . These nogoods ensure that at most one value can be assigned to each variable and are represented by vertical cliques in Figure 5.2.

The microstructure of INJECTIVE is a rather cluttered graph, and not shown here. This is due to the large number of instance props involving every variable. Recall Proposition 4.33 and note that INJECTIVE is its own explicit version; therefore every edge between literals that is not part of the clause structure is contained in the microstructure.  $\square$



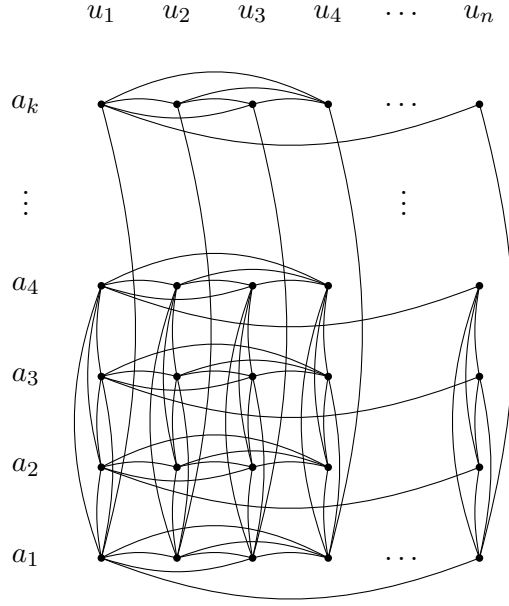


Figure 5.2: Clause structure of INJECTIVE.

The ALL-DIFFERENT constraint is another important constraint that can be represented by its exact 2-section.

**Example 5.10** (decomposition of ALL-DIFFERENT). The ALL-DIFFERENT constraint was discussed in Example 3.15. Instead of the microstructure containing high arity constraints, the microstructure of ALL-DIFFERENT can be expressed as a binary structure. The exact 2-section of ALL-DIFFERENT contains the ALL-DIFFERENT constraint as a subproblem by Proposition 5.4. However, notice that as with the INJECTIVE constraint, the infrastructure of ALL-DIFFERENT is conformal. By Proposition 5.7, ALL-DIFFERENT can be equivalently represented by its 2-structure.

$\text{ALL-DIFFERENT}(u_1, u_2, \dots, u_r; D_1, D_2, \dots, D_r)$  is the constraint (in homomorphism representation)

$$((V, \{(u_1, u_2, \dots, u_r)\}), (D, \{(v_1, v_2, \dots, v_r) \in D_1 \times D_2 \times \dots \times D_r \mid i \neq j \Rightarrow v_i \neq v_j\})),$$

where  $V = \{u_1, u_2, \dots, u_r\}$  and  $D = \bigcup \{D_i \mid i = 1, 2, \dots, r\}$ . By Proposition 5.7 this is then equivalent to the instance with set of variables  $V$ , domain  $D$ , and set of constraints

$$\{((u_i, u_j), \neq_D) \mid 1 \leq i < j \leq r\} \cup \{\text{UNARY}(u_1; D_1), \dots, \text{UNARY}(u_r; D_r)\}.$$

Let  $d = |D|$  be the number of distinct domain values used. If  $D_i = D$  for each  $i$ , the relation of arity  $r$  in the original may have as many as  $(d - 1)^r$  tuples, while in the binary representation via the 2-structure there are  $r(r - 1)/2$  relations each containing at most  $d(d - 1)$  tuples.  $\square$

In Example 5.10, expressing ALL-DIFFERENT in terms of a clique of inequalities results

in a quadratic number of binary constraints with a quadratic number of tuples, instead of the potentially exponential number of tuples in a single constraint relation obtained by using the definition directly.

Since Sudoku can be expressed using ALL-DIFFERENT constraints, it is another example where the exact 2-section suffices.

**Example 5.11** (microstructure of Sudoku). Using the binary expression of the ALL-DIFFERENT constraint in Example 5.10, with unary constraints for the numbers already in the grid, the Sudoku puzzle (see Example 2.31 and Figure 2.6) can then be modelled as a binary CSP.

Part of the clause structure of a natural way to represent this problem is shown in the following figure. The variables represent the unknown values for each square, with  $x_{ij}$  representing the square in row  $i$  and column  $j$ .

The clauses are indicated using different colours. Red is used for the 9 cliques of the form

$$\mathcal{C}_{i..a} = \{(x_{ij} \neq a) \vee (x_{ik} \neq a) \mid j, k \in [9]; j \neq k\},$$

representing the ALL-DIFFERENT constraint on row  $i$ . Blue is used for the 9 cliques of the form

$$\mathcal{C}_{.j.a} = \{(x_{ij} \neq a) \vee (x_{kj} \neq a) \mid i, k \in [9]; i \neq k\},$$

representing the ALL-DIFFERENT constraint on column  $j$ . Black is used for the 9 cliques of the form

$$\mathcal{C}_{i,j,a} = \{(x_{3i+k,3j+l} \neq a) \vee (x_{3i+k',3j+l'} \neq a) \mid k, k', l, l' \in [3]; k \neq k' \vee l \neq l'\},$$

representing the ALL-DIFFERENT constraint on block  $i, j$ . This part of the clause structure represents the constraints involved between all literals containing the domain value  $a \in [9]$ .

The clause structure then consists of nine layers of these structures, one for each domain value, tied together with 81 variable cliques of the form

$$\mathcal{C}_{i,j} = \{(x_{ij} \neq a) \vee (x_{ij} \neq b) \mid a, b \in [9]; a \neq b\}$$

that enforce each variable is assigned at most one value.

In addition, the unary constraints representing the values already in the grid are represented by removing all the literals inconsistent with the unary constraint from the clause structure. Suppose the cell  $x_{11}$  already contains the value 4, as in the instance in Figure 2.6. Then  $(x_{11} = 4)$  must be true, so the literals  $(x_{11} \neq a)$  are removed for each  $a \neq 4$ . In Figure 5.3, all the literals associated with unary constraints have been applied in this way, and there are 22 literals missing from this layer. I indicate these missing literals with circles, instead of solid dots.

Notice that the  $r$ -ary representation discussed in Example 4.17 is much larger than the representation discussed here via binary constraints.  $\square$

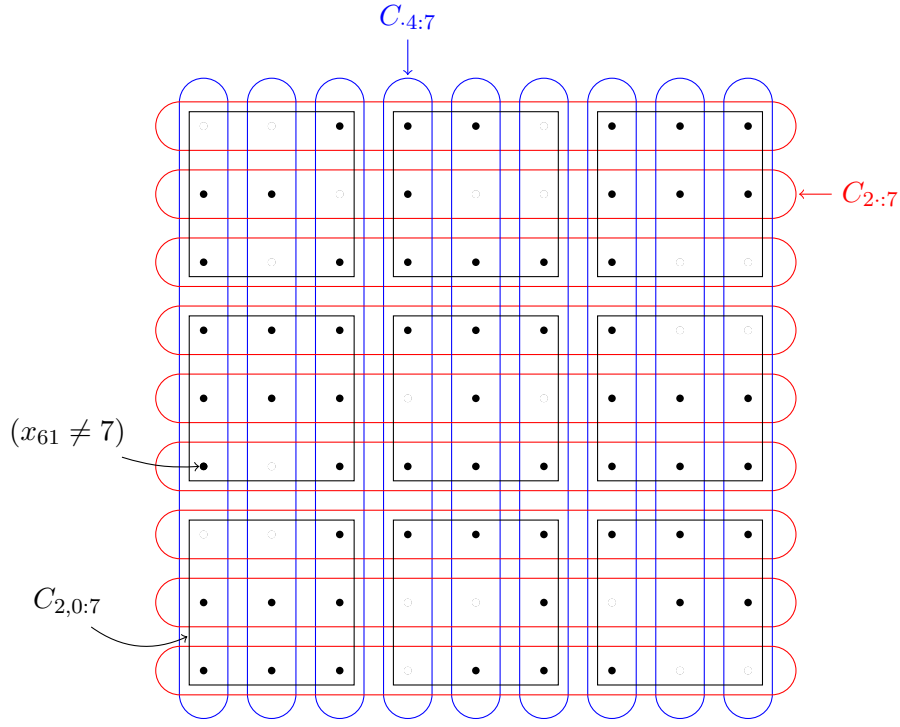


Figure 5.3: A slice of the clause structure of the Sudoku instance in Figure 2.6, relating to value 7.

The final example where the exact 2-section is equivalent to the original instance is Futoshiki.

**Example 5.12** (microstructure of Futoshiki). One way to model a Futoshiki instance (defined in Example 2.32) is by a combination of binary ALL-DIFFERENT constraints for the rows and columns, binary  $<$  constraints between some squares, and unary constraints specifying the values that are filled in. This then forms a binary CSP instance.

Consider the Futoshiki instance in Example 2.32. The vertices of the clause structure are pairs of the form  $(x_{ij}, a)$ , representing the assignment  $(x_{ij} = a)$ . Here  $i$  and  $j$  are elements of  $[5]$ , as is  $a$ . The edges of the microstructure complement are of three types. The first type of edges form a set

$$\{((x_{ij}, a), (x_{ik}, a)) \mid i, j, k \in [5]; j \neq k\},$$

representing the clique of inequalities for each row. The second type

$$\{((x_{ij}, a), (x_{kj}, a)) \mid i, j, k \in [5]; i \neq k\}$$

represents the clique of inequalities for each column. Finally, for every constraint  $x_{ij} < x_{kl}$  between adjacent distinct squares  $x_{ij}$  and  $x_{kl}$  of the grid, either  $i = k$  or  $j = l$  but not

both, and there are edges

$$\{((x_{ij}, a), (x_{kl}, b)) \mid a, b \in [5]; a < b\}$$

in the clause structure. ┘

### 5.1.3 Hidden variable transformation

Every CSP instance can be represented by an equivalent binary instance, by introducing new variables [47]. In an instance with  $q$  constraint relations  $R_i$  for  $i = 1, \dots, q$ , the number of new  $d$ -valued variables can be restricted to be at most  $\sum_{i=1}^q (|R_i| - 2)/(d - 2)$ . This is polynomial in the instance size if the relations are written extensionally in the instance, which is often the assumption made in the literature.

Moreover the number of new variables is also polynomial if the relations each contain only polynomially many tuples.

In contrast, the following important constraint is often represented intensionally, and if represented extensionally, the size of the instance may grow faster than any polynomial.

**Example 5.13** (decomposing SUM).  $\text{SUM}(u_1, u_2, \dots, u_r; C)$  is satisfied when  $\sum_{i=1}^r u_i = C$ . A single SUM constraint of arity  $r$  can equivalently be expressed by decomposing it into  $r - 1$  ternary SUM constraints, using  $r - 2$  new variables. This can be done by introducing new variables  $w_1, w_2, \dots, w_{r-2}$ , as follows:

$$\begin{aligned} \{\text{SUM}(u_1, u_2, \dots, u_r; C)\} &\equiv \{\text{SUM}(u_1, u_2; w_1)\} \\ &\cup \{\text{SUM}(w_i, u_{i+2}; w_{i+1}) \mid i = 1, 2, \dots, r - 3\} \\ &\cup \{\text{SUM}(w_{r-2}, u_r; C)\}. \end{aligned}$$

This tree of  $r - 1$  ternary constraints is path-like; it has depth  $r - 1$ . If desired, the tree can be rearranged so its depth is logarithmic in  $r$ .

The SUM constraint generally has a fast-growing number of tuples, being the number of different ways for  $r$  values from the domain to sum to a given value  $C$ . This is known to grow as  $C^{r-1}$  as  $C$  grows, when the domain is the positive integers and  $r$  is fixed [102]. There is no polynomial bound if the arity of SUM constraints with sum  $C$  that appear in a class of CSP instances grows faster than  $\Omega(\sqrt{C})$ . ┘

Contrast the SUM constraint with Example 2.26, where the number of partitions of  $C$ , not restricted to have  $r$  parts, also grows faster than any polynomial.

I note the following standard arity reduction for SAT instances, obtained by introducing new variables.

**Example 5.14** (SAT as  $k$ -SAT). A SAT instance with  $m$  clauses and at most  $K$  literals in any clause can be represented as a  $k$ -SAT instance (where  $3 \leq k < K$ ), by introducing at most  $(\lceil \frac{K}{k-1} \rceil - 1)m = O(mn)$  new variables. Each clause with  $p > k$  literals is broken into

$\lceil \frac{p}{k-1} \rceil$  clauses, each with at most  $k$  literals, by introducing  $\lceil \frac{p}{k-1} \rceil - 1$  new variables. Each new variable represents a proposition of the form “the literals of clause  $i$  up to the  $j$ -th literal are all false”. (Note that this fails for  $k$  smaller than 3, since then this procedure does not make the clauses shorter.)  $\square$

#### 5.1.4 Dual and hidden transformation

Any hypergraph has an associated **incidence graph** or **dual graph**, with vertices of two types: the vertices of the hypergraph, and the edges of the hypergraph. A vertex  $v$  of the hypergraph is connected to an edge  $e$  of the hypergraph in the incidence graph precisely when  $v$  appears in  $e$ . The incidence graph of a hypergraph is by definition bipartite. Moreover, it is possible to retrieve the original hypergraph from its incidence structure [12].

The incidence graph of the hypergraph of a CSP instance has also been called the dual constraint graph [49, 139]. This is a bipartite graph, containing as its vertices all constraint scopes and all variables, with a variable adjacent to a scope whenever the scope includes the variable. This gives rise to a binary CSP instance, and is known as the **dual transformation** [49, 7].

Another approach, known as the **hidden transformation**, or sometimes even **hidden variable transformation** (but not to be confused with the approach of introducing hidden variables discussed in Section 5.1.3), is closely related to the dual transformation [7].

In the hidden transformation new variables are added to the CSP instance to capture the notion of “related by” [139]. For each constraint  $(\sigma, R)$ , a new variable  $v_\sigma$  is introduced, with domain  $R$ . For each variable  $u$  in  $\sigma$  a binary constraint with scope  $(u, v_\sigma)$  is added, relating each value of  $u$  with the corresponding tuples in  $R$ ; this enforces the semantics of the original constraint. This transformation turns every CSP instance into a binary one, using a single new hidden variable per constraint that encodes the tuples of the relation. This means that the domain is exponentially larger.

In contrast the dual transformation has a single variable for each of the constraints in the original instance, but does not carry over the original variables. The constraints in the dual transformation only relate pairs of these new variables.

Key disadvantages of the dual and hidden transformations are that new variables must be introduced for each constraint, the domains that result from such a transformation are usually large, and much of the structure of the CSP is obscured via these transformations.

#### 5.1.5 Reflections on transforming to binary

Peirce argued that relations of higher arity can more naturally express information about the real world. The theory and practice of relational databases can be seen as a demonstration that high arity relations are an effective representation. In contrast, classical network databases, and more modern key-value and triple stores used by so-called NoSQL databases can be regarded as an argument that data can be effectively represented as low arity relations. As can be seen with the back-and-forth struggle in the database world between

the different kinds of database paradigms, there does not appear to be a final answer as to whether data should be represented using high arity relations, or whether all relations should have some low fixed arity (such as binary or ternary).

However, there is currently one important reason to prefer binary CSPs: the microstructure representation of a binary CSP instance is a graph, and there is a large body of research about finding independent sets in graphs. The Information System on Graph Classes and their Inclusions at the time of writing lists nearly 1500 classes of graphs [46]. For many of these classes it is known either that finding an independent set of specified size is NP-hard, or that a polynomial-time algorithm exists to find a largest independent set. These results can be used to carve out tractable classes of CSPs based on their microstructure representation. In contrast, no such resource exists for hypergraphs, and I have not been able to find even one class of hypergraphs for which there is a polynomial-time algorithm for finding largest independent sets and which also relates to an interesting CSP.

Therefore, my focus in this and the following chapter is on the binary case. Much of the theory does apply to arbitrary arity, but without convincing case studies available I have not pursued this direction as far as for the binary case. Although my natural inclination is to work with arbitrary arity, the scarcity of usable material to build on has meant that most of the text is for the binary case. This tension regarding arity is regrettable but unfortunately necessary.

## 5.2 Hereditary classes

Hereditary classes are those that do not have some class of structures as induced substructures, or equivalently, are downward-closed. Forbidden induced substructures are reasonably straightforward, though some care is required when comparing different classes.

### 5.2.1 Forbidden substructures

Classes of structures defined by forbidding some induced substructures occur frequently. The following definition is standard in the graph theory literature (see for instance [29, 84, 46]). I extend the notation to more general kinds of structures.

**Definition 5.15** (free, for structures). A structure  $H$  is  $G$ -**free** if  $H$  contains no induced substructure isomorphic to  $G$ . For a set of structures  $X$ , a structure  $H$  is  $X$ -**free** if  $H$  contains no induced substructure isomorphic to any  $G$  in  $X$ . Also,  $X$ -free denotes the class of all structures that are  $X$ -free.  $(G_1, G_2, \dots)$ -free denotes  $\{G_1, G_2, \dots\}$ -free.  $\square$

I use the term *free* since the usage of *forbidden* is often associated in the graph theory literature with forbidden subgraphs that are not necessarily induced. The following example illustrates forbidden induced substructures for the case of graphs.

**Example 5.16** (free).  $C_4$  contains no substructure isomorphic to  $C_3$ , so is  $C_3$ -free.  $K_4$  contains no induced substructure isomorphic to  $C_4$  (although it does contain substructures

isomorphic to  $C_4$ ), so is  $C_4$ -free.  $K_4$  contains  $C_3$  as an induced substructure, so is *not*  $C_3$ -free.

In some cases higher arity constraints contain useful structure, which is lost when moving to binary relations. The following result and example make explicit this intuition.

**Lemma 5.17.** *If  $G$  is a graph with at most  $r$  vertices, and  $K_r$  is  $G$ -free, then  $K_{r+1}$  is also  $G$ -free.*

*Proof.* Suppose that  $K_r$  is  $G$ -free but  $K_{r+1}$  contains  $G$  as an induced subgraph. Then  $G$  must be isomorphic to  $K_s$  for some  $s \leq r+1$ . However, as  $K_r$  is  $G$ -free,  $s > r$ . Hence  $G$  must contain  $r+1$  vertices, a contradiction.  $\square$

**Example 5.18.** Consider the uniform hypergraph  $H_r$  with arity  $r$  edges on vertices  $[n]$ , with dual graph  $\mathcal{DG}(H_r) = K_{r,n}$ . For any  $r \geq 3$ , by considering the cardinality of edges it is clear that  $H_{r+1}$  is  $H_r$ -free, but  $\mathcal{DG}(H_{r+1})$  contains  $\mathcal{DG}(H_r)$ .  $\square$

Lemma 5.17 and Example 5.18 show that there are classes of hypergraphs which can be characterized by means of forbidden induced substructures, but which cannot be characterized by means of forbidden induced subgraphs in their dual graphs.

In particular, this means that there are classes of CSPs that can be characterized by means of forbidden induced substructures in their microstructures, yet cannot be characterized by means of forbidden induced subgraphs if the relations are represented as binary relations.

I therefore argue that the methods in this chapter are more appropriate tools than transforming the CSP into its incidence graph, at least given our current state of knowledge about structural transformations of CSPs.

Some simple rules govern how classes of instances with forbidden substructures can be combined. These follow immediately from Definition 5.15.

**Proposition 5.19** (combining hereditary classes).

1.  $X\text{-free} \cap Y\text{-free} = X \cup Y\text{-free}$ .
2. If  $G$  is an induced substructure of  $H$  and  $H$  is  $X$ -free, then  $G$  is  $X$ -free.
3. If every structure in  $X$  is also in  $Y$ , then  $Y\text{-free} \subseteq X\text{-free}$ .
4. If  $G$  is an induced substructure of  $H$ , then  $\{G\}\text{-free} \subseteq \{H\}\text{-free}$ .
5. If all structures in  $X$  are induced substructures of  $H$ , then  $X\text{-free} \subseteq \{H\}\text{-free}$ .
6.  $H$  is  $G$ -free if, and only if,  $\overline{H}$  is  $\overline{G}$ -free.

When  $\{G_c, G_{c+1}, \dots\}$  is an infinite set of graphs indexed by integers beginning at  $c$ , then I will also use the notation  $G_{c+i}\text{-free}$  to denote  $(G_c, G_{c+1}, \dots)\text{-free}$ .

**Example 5.20** (tree). A **tree** is a graph that is  $\{C_i \mid i = 3, 4, 5, \dots\}$ -free. Equivalently, a tree is  $C_{3+i}$ -free.  $\square$

### 5.2.2 IS-easy and IS-hard classes of graphs

The key transformation considered in this chapter is reducing a CSP instance to an instance of INDEPENDENT SET, with the clause structure as its input. Hence a key question is whether there is an efficient algorithm  $A$  to decide INDEPENDENT SET when presented with the class of input graphs  $C'$  that is obtained as clause structures of a class  $C$  of CSP instances.

Note that the decision problem associated with the class of graphs obtained by the microstructure transformation does not have to be in  $P$ , or even to be decidable at all. It is not necessary to distinguish graphs that are in class  $C'$  from those not in  $C'$ ; to decide  $C$  it is enough to simply run algorithm  $A$  for any input graph in  $C'$ .

I now introduce some terminology to differentiate the notion of deciding a class of CSP instances from the promise problem of deciding whether the associated clause structures have sufficiently large independent sets.

**Definition 5.21.** Class  $C$  of graphs is **IS-hard** if INDEPENDENT SET is NP-complete when the input graph is restricted to be from  $C$ , and **IS-easy** if INDEPENDENT SET can be decided in polynomial time when the input graph is restricted to be from  $C$ .

Note that unless  $P = NP$ , there are classes  $C$  of graphs which are neither IS-easy nor IS-hard [18, Theorem 5].

Hereditary classes are well-behaved with respect to decision problems.

**Proposition 5.22** (promise problems for hereditary classes). *Let  $\text{DECProb}(X)$  be the decision problem DECProb, conditioned on the promise that the input is from class  $X$ . Let  $X \subseteq Y$  be classes of structures.*

1. *If  $\text{DECProb}(X\text{-free})$  is in  $P$ , then  $\text{DECProb}(Y\text{-free})$  is in  $P$ .*
2. *If  $\text{DECProb}(Y\text{-free})$  is NP-complete, then  $\text{DECProb}(X\text{-free})$  is NP-hard.*

*Proof.* The key observation is that  $Y\text{-free} \subseteq X\text{-free}$  by Proposition 5.19.

For the first part, suppose  $\text{DECProb}(X\text{-free})$  is in  $P$ . Then it can be decided by a polynomial-time algorithm; this algorithm will also decide any instance in  $Y\text{-free}$  in polynomial time.

For the second part, suppose  $\text{DECProb}(Y\text{-free})$  is NP-complete. There is then a trivial reduction from  $\text{DECProb}(Y\text{-free})$  to  $\text{DECProb}(X\text{-free})$ , so the latter problem is NP-hard.  $\square$

**Corollary 5.23.** *Let  $X \subseteq Y$  be classes of graphs.*

1. *If  $X$  is IS-easy, then  $Y$  is IS-easy.*
2. *If  $Y$  is IS-hard, then  $X$  is IS-hard.*



### 5.2.3 Hereditary classes, forbidden substructures, and domain reduction

**Definition 5.24.** A hypergraph  $G$  is a **strict substructure** of a hypergraph  $H$  if  $V(G) \subseteq V(H)$  and for every edge  $E$  of  $G$  there exists an edge  $F$  of  $H$  such that  $E \subseteq F$ .  $\square$

Definition 5.24 requires the vertices of  $G$  to be vertices of  $H$  also. This can be extended to a more generally applicable notion of substructure up to isomorphism.

**Definition 5.25.** A hypergraph  $G$  is a **substructure** of a hypergraph  $H$  if there exists a strict substructure  $G'$  of  $H$  such that  $G$  is isomorphic to  $G'$ .  $\square$

The notion of isomorphism in Definition 5.25 is structure isomorphism, by considering the hypergraph as a relational structure.

Definition 5.25 now allows comparison of CSP instances by allowing their microstructures to be compared using the substructure relation.

Note that for two hypergraphs  $G$  and  $H$ , if  $G$  is a subhypergraph of  $H$  then  $G$  is a substructure of  $H$ , but the converse may not hold (consider  $G$  and  $H$  having the same vertices  $V$ , but suppose that  $V$  is the only edge of  $H$ ; then clearly  $G$  is always a substructure of  $H$  while if  $G$  has any other edge then  $G$  is not a subhypergraph of  $H$ ).

The basic operation in constraint propagation is **domain reduction**, the removal of a value from the active domain of a variable. (This is sometimes also called domain pruning or domain filtering.) Information about the instance is captured by the set of possible values for each variable. If an instance is in a class of structures, then the instances formed when domain values are removed from the instance should also be in the class.

When a domain value  $a$  for variable  $u$  is removed, 1-consistency will ensure that all tuples in which  $a$  may be assigned to  $u$  are removed from the constraint relations.

**Theorem 5.26.** Let  $\mathcal{C}$  be a class of CSP instances. The following are equivalent:

1.  $\mathcal{C}$  is closed under the operation of taking subproblems.
2. The class of microstructures  $MS(\mathcal{C}) = \{MS(\mathcal{P}) \mid \mathcal{P} \in \mathcal{C}\}$  of  $\mathcal{C}$  is closed under the operation of removing domain values.
3. The class of clause structures  $CS(\mathcal{C}) = \{CS(\mathcal{P}) \mid \mathcal{P} \in \mathcal{C}\}$  of  $\mathcal{C}$  is closed under the operation of removing domain values.

*Proof.* First, suppose a class  $\mathcal{C}$  of instances is closed under the operation of taking subproblems. Let  $\mathcal{P}$  be an instance in  $\mathcal{C}$  with microstructure  $H = MS(\mathcal{P})$ , and suppose that hypergraph  $H'$  is obtained by removing some vertices from  $MS(\mathcal{P})$ . Without loss of generality, suppose that  $H'$  is obtained by removing just one vertex  $(u, a)$  from  $MS(\mathcal{P})$ . It is straightforward to recover an instance  $\mathcal{P}'$  from  $H'$  as the variables of  $H$  are known. The infrastructure of  $\mathcal{P}'$  is now a subset of the infrastructure of  $\mathcal{P}$ , as some props may have been removed by removing  $(u, a)$ , but none can have been added. Hence  $\mathcal{P}'$  is a

subproblem of  $\mathcal{P}$ , and  $\mathcal{P}'$  is then in  $\mathcal{C}$ . Its microstructure  $H' = \text{MS}(\mathcal{P}')$  is then contained in  $\text{MS}(\mathcal{C})$ .

Now suppose  $\text{MS}(\mathcal{C})$  is closed under the operation of removing domain values. Suppose  $H \in \text{CS}(\mathcal{C})$  and, again without loss of generality, that  $H'$  is obtained from  $H$  by removing just one vertex  $(u, a)$  and any empty edges that may result. Now let  $G'$  be the complement of  $H'$  with respect to the scopes that appear in  $H'$ , and let  $G$  be the complement of  $H$  with respect to the scopes that appear in  $H$ . Note that the only possible edge removed from  $H$  is the singleton edge containing just  $(u, a)$ . This means that every edge in  $H$  (other than possibly this singleton edge) is still present in  $H'$ , once  $(u, a)$  is removed from any edges in which it is present in  $H$ . Hence  $G'$  contains those edges in  $G$  which did not contain  $(u, a)$ , and any edge in  $G$  which did contain  $(u, a)$  is either present in  $G'$  without  $(u, a)$  appearing in such edges, or in the case of the singleton edge, does not appear at all. Hence  $G'$  is obtained from  $G$  by removing  $(u, a)$  from any edge in which it appears, and discarding any empty edges that result. As  $\text{MS}(\mathcal{C})$  is closed under removal of domain values,  $G' \in \text{MS}(\mathcal{C})$ . Hence  $H' \in \text{CS}(\mathcal{C})$ . It follows that  $\text{CS}(\mathcal{C})$  is closed under removal of domain values.

Finally, suppose that  $\text{CS}(\mathcal{C})$  is closed under removal of domain values. Let  $\mathcal{P}$  be an instance in  $\mathcal{C}$  with clause structure  $H \in \text{CS}(\mathcal{C})$ . Let  $\mathcal{P}'$  be a subproblem of  $\mathcal{P}$ , with clause structure  $H'$ . For every variable  $u$  in  $\mathcal{P}$  which does not occur in  $\mathcal{P}'$ , for each  $a$  in the domain of  $u$ , remove each  $(u, a)$  from  $H$ , yielding a subproblem  $\mathcal{P}_0$ . After performing this operation, the variables of  $\mathcal{P}_0$  and  $\mathcal{P}'$  are the same. Then for every value  $a$  in the domain of some variable  $u$  which does not occur in the domain of  $u$  in  $\mathcal{P}'$ , remove  $(u, a)$  from  $\mathcal{P}_0$ , yielding subproblem  $\mathcal{P}_1$ . Now the literals of  $\mathcal{P}_1$  are the same as the literals of  $\mathcal{P}'$ , and  $\mathcal{P}_1$  was obtained from  $\mathcal{P}$  by removing one domain value at a time, preserving all edges which do not feature removed literals. It then follows that  $\text{CS}(\mathcal{P}_1) = H'$ . As  $H'$  was obtained from  $H$  by removing domain values, one at a time,  $H' \in \text{CS}(\mathcal{C})$ . Hence  $\mathcal{P}' \in \mathcal{C}$ . Therefore,  $\mathcal{P}$  is closed under taking subproblems.  $\square$

Due to Theorem 5.26, I therefore require CSPs to be downward-closed with respect to induced substructures.

**Definition 5.27.** Let  $\preceq$  be a partial order over a set  $X$ . A subset  $Y$  of  $X$  is said to be **downward-closed** with respect to  $\preceq$  if whenever  $y \in Y$  and  $x \in X$  such that  $x \preceq y$ , then  $x \in Y$ .

**Definition 5.28** (hereditary class). A class  $\mathcal{C}$  of structures is **hereditary** if whenever  $G$  is a structure in  $\mathcal{C}$ , and  $H$  is an induced substructure of  $G$ , then  $H$  is in  $\mathcal{C}$  also.

Hence hereditary classes are those that are downward-closed with respect to the induced substructure partial order on structures.

The hereditary classes can also be characterized by means of forbidden induced substructures: a hereditary class is  $X$ -free for some class  $X$  of structures. In the graph theory literature this result is usually observed without proof.

**Proposition 5.29** (hereditary vs. forbidden induced substructures).  *$\mathcal{C}$  is hereditary if, and only if, there is a set of structures  $X$  such that  $\mathcal{C} = X$ -free.*

*Proof.* For the forward direction, suppose that  $\mathcal{C}$  is hereditary. Let  $X$  consist of the minimal structures (in terms of the induced substructure partial order) which are not in  $\mathcal{C}$ , where only one of each isomorphism equivalence class is included in  $X$ . This is then a set. (Note that minimality is well-defined for finite structures.) No structure in  $X$  can be in  $\mathcal{C}$ , so therefore  $\mathcal{C}$  is contained in  $X$ -free. Now let  $H$  be a structure that is  $X$ -free. If  $H$  were not contained in  $\mathcal{C}$ , then there would exist some structure  $G$  in  $X$  but that was also an induced substructure of  $H$ . This would then contradict  $H$  being  $X$ -free. Hence  $H$  is in  $\mathcal{C}$ , and as the choice of  $H$  was arbitrary,  $X$ -free is contained in  $\mathcal{C}$ .

For the reverse direction, suppose that  $\mathcal{C} = X$ -free for some set of structures  $X$ . Suppose  $H$  is a structure that is in  $\mathcal{C}$ . Since  $H$  is  $X$ -free, so is any induced substructure of  $H$ . It follows that  $\mathcal{C}$  is hereditary.  $\square$

Proposition 5.29 combined with Theorem 5.26 means that I will only consider those classes which are hereditary. Either the downward-closed property of hereditary classes, or the definition via forbidden induced substructures can be used.

#### 5.2.4 Small forbidden structures

It is useful to consider the classes formed by excluding a single small structure. In the light of Proposition 5.19, these classes form the building blocks for more complex hereditary classes.

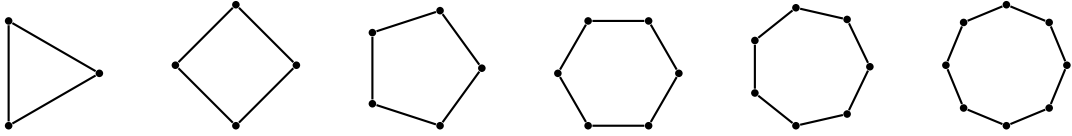


Figure 5.4:  $C_i$  for  $i = 3, 4, 5, 6, 7, 8$

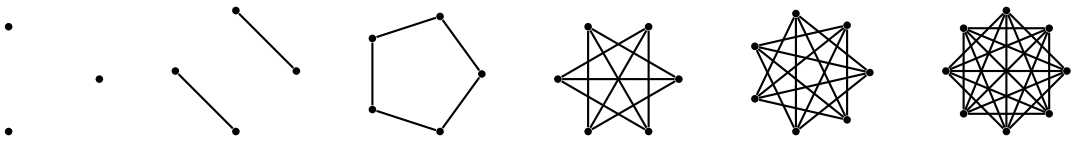


Figure 5.5:  $\overline{C_i}$  for  $i = 3, 4, 5, 6, 7, 8$

A **hole** is an induced cycle with at least 5 vertices [124, 80]. An **antihole** is the complement of a cycle with at least 5 vertices. The terminology for holes and antiholes is not completely settled: some texts refer to induced cycles with at least 4 vertices as holes, and cycles with at least 5 vertices are then called large holes [79]. Side-stepping the size controversy, I will refer to a **giant hole** for cycles with at least 7 vertices.

Berge conjectured in 1961 that perfect graphs are precisely those graphs that are (odd-hole, odd-antihole)-free; a proof took over four decades to discover.

**Theorem 5.30** ([28, 1.2], Strong Perfect Graph Theorem). *A graph is perfect if, and only if, it is (odd-hole, odd-antihole)-free.*

Perfect graphs are important in my work because of the following classical result.

**Theorem 5.31** ([76, Sect. 6]). *A maximum clique in a perfect graph can be found in polynomial time.*

Perfect graphs can also be recognised in polynomial time [27].

Table 5.1 summarises what is known about the complexity of IS for classes of graphs defined by excluding a single small induced subgraph. The graphs illustrated on the right hand side are complements of those illustrated on the left. For each entry a chain of inferences establishing its complexity is provided. Italicized blue entries are IS-hard, and those in normal typeface are IS-easy.

These results are folklore and quite simple to establish. However, as far as I can establish, no table such as this has been published. Enumerating the  $G$ -free classes of graphs for small  $G$  (as well as classifying them with respect to being IS-easy or IS-hard) is the first step toward constructing the similar table in Chapter 6 for richer structures.

$P_3$  is a path on three vertices. Its complement  $\text{co-}P_3$  is the graph on three vertices with a single edge. The classes of  $P_3$ -free graphs and  $\text{co-}P_3$ -free graphs are subclasses of perfect graphs. (Note that in a binary CSP instance with a  $\text{co-}P_3$ -free microstructure, every pair of distinct variables is either related by an anything-goes relation, or an empty relation (in the latter case, the instance has no solution). Either of these cases leads to a polynomial-time solution. Similarly, a  $P_3$ -free microstructure ensures that any solution forms a clique in the microstructure that has no vertex in common with the clique of any other solution, so the set of possible solutions can be enumerated and checked in polynomial time.)

A **house** is a graph consisting of  $C_4$  with an additional vertex connected to two vertices that are adjacent in the  $C_4$ . Many other graphs have special names that are frequently used: **diamond** is  $K_4$  with one edge removed;  $P$  is a  $C_4$  with an additional vertex connected to one of the vertices in the  $C_4$ ; the **paw** consists of a vertex connected by an edge to one of the vertices of a triangle. A **star**  $S_{i,j,k}$  consists of three disjoint paths  $P_i, P_j, P_k$  together with a central vertex joined to one endpoint of each path.  $S_{1,1,1} = K_{1,3}$  is also known as the **claw**, and  $S_{1,1,2}$  is also known as the **fork**. The graphs  $H$ ,  $X_{85}$ , house, fork,  $P$ , diamond, claw, and paw are illustrated in Figure 5.6.

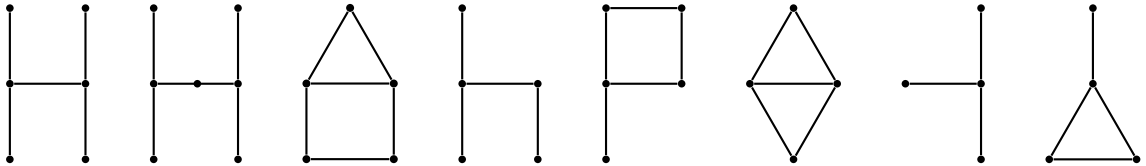


Figure 5.6: Graphs  $H$ ,  $X_{85}$ , house, fork,  $P$ , diamond, claw, paw.

I now summarize some previously published results from [46], which I will combine with Corollary 5.23. Let  $Q$  denote the class of  $(C_4, C_5, C_6, C_7, C_8, H, K_{1,4}, X_{85}, K_3)$ -free

graphs.

**Proposition 5.32.**

1.  $Q$  is IS-hard, so classes containing it are also IS-hard: these include  $K_3$ -free,  $(C_4, \text{co-claw})$ -free,  $(C_4, C_5)$ -free,  $(K_{1,4}, \text{diamond})$ -free, and  $P$ -free.
2. Classes  $(K_2 \cup \text{claw})$ -free, co-gem-free,  $(P, S_{1,2,5})$ -free, and  $(2+c)K_2$ -free for any fixed  $c = 0, 1, \dots$  are all IS-easy. Recently it was established that the  $P_5$ -free class is also IS-easy [113].

The complexity of INDEPENDENT SET for  $S_{1,2,5}$ -free graphs is not currently known.

### 5.3 Perfect microstructure

**Proposition 5.33.** *If  $H$  is a graph,  $s \geq |V(H)|$ , and the CSP instance  $(K_s, H)$  has perfect microstructure, then  $H$  is perfect.*

*Proof.* If  $H$  is not perfect then it contains an odd hole of size  $n$ , or  $\overline{H}$  does; enumerate the vertices of the cycle as  $v_1, v_2, \dots, v_n$  so that  $v_i$  is adjacent to  $v_{i+1}$  and  $v_n$  is adjacent to  $v_1$ . Now consider any subset  $X$  of  $n$  distinct vertices from  $V(K_s)$ , enumerated as  $X = \{u_1, u_2, \dots, u_n\}$ . The assignments  $\{(u_1, v_1), (u_2, v_2), \dots, (u_n, v_2)\}$  then induce an odd hole or antihole in the microstructure.  $\square$

The converse of Proposition 5.33 is not true; consider the microstructure of  $(K_4, \overline{P_5})$ . This contains an induced  $C_5$  (this cycle includes two different assignments to one of the variables), so is not perfect.

Combining constraints may fail to preserve the perfect microstructure of the individual constraints.

**Example 5.34** (combining constraints). Suppose  $G$  and  $H$  are graphs formed from  $C_5$  by adding one edge to  $C_5$ , with the edge added to form  $G$  different from the edge added to form  $H$ . Then

$$((V(C_5), E(C_5), E(K_5)), (\{1, 2\}, E(G), \{(2, 2)\}))$$

and

$$((V(C_5), E(C_5), E(K_5)), (\{1, 2\}, E(H), \{(2, 2)\}))$$

both have perfect microstructure, each consisting of a disjoint union of a 5-clique and a 5-cycle with a chord. Moreover, each of these CSP instances has a solution by mapping each vertex in the source structure to 2 in the target structure. However,

$$(V(C_5), E(C_5), E(K_5), E(C_5), E(K_5)), (\{1, 2\}, E(G), \{(2, 2)\}, E(H), \{(2, 2)\}))$$

does not have perfect microstructure, as it consists of a disjoint union of a 5-clique and a 5-cycle. Note that the two different chords fall away in the microstructure when the




















































	$\overline{K_2}$ -free = $\{K_i\} \subseteq$	perfect	$\supseteq \{\overline{K_i}\} = K_2$ -free	
	$\overline{K_3}$ -free $\subseteq$ claw-free $\subseteq (P, S_{1,2,5})$ -free		$K_3$ -free	
	$\overline{P_3}$ -free $\subseteq (\overline{C_{4+i}}, C_{5+i})$ -free $\subseteq$	perfect	$\supseteq (C_{4+i}, \overline{C_{5+i}})$ -free $\supseteq P_3$ -free	
	$\overline{K_4}$ -free $\subseteq (K_2 \cup \text{claw})$ -free		$K_3$ -free $\subseteq K_4$ -free	
	co-diamond-free $\subseteq (K_2 \cup \text{claw})$ -free		$K_3$ -free $\subseteq \text{diamond-free}$	
	co-paw-free $\subseteq (K_2 \cup \text{claw})$ -free		$K_3$ -free $\subseteq \text{paw-free}$	
	$2K_2$ -free $\subseteq (K_2 \cup \text{claw})$ -free		$(C_4, \text{co-claw})$ -free $\subseteq C_4$ -free	
	claw-free $\subseteq (P, S_{1,2,5})$ -free		$K_3$ -free $\subseteq \text{co-claw-free}$	
	$P_4$ -free $\subseteq$ weakly chordal $\subseteq$	perfect	[note: $P_4 = \overline{P_4}$ ]	
	$\overline{K_5}$ -free $\subseteq 5K_2$ -free		$K_3$ -free $\subseteq K_5$ -free	
	$(5K_1 + e)$ -free $\subseteq 4K_2$ -free		$K_3$ -free $\subseteq (K_5 - e)$ -free	
	$P_3 \cup 2K_1$ -free $\subseteq (P, S_{1,2,5})$ -free		$K_3$ -free $\subseteq \overline{P_3 \cup 2K_1}$ -free	
	co- $W_4$ -free $\subseteq 3K_2$ -free		$K_3$ -free $\subseteq W_4$ -free	
	$K_1 \cup \text{claw}$ -free $\subseteq (K_2 \cup \text{claw})$ -free		$K_3$ -free $\subseteq \overline{K_1 \cup \text{claw}}$ -free	
	$P_2 \cup P_3$ -free $\subseteq (K_2 \cup \text{claw})$ -free		$K_3$ -free $\subseteq \overline{P_2 \cup P_3}$ -free	
	co-gem-free $\subseteq (P, S_{1,2,5})$ -free		$K_3$ -free $\subseteq \text{gem-free}$	
	$(K_3 \cup 2K_1)$ -free $\supseteq$	$K_3$ -free	$\subseteq \text{co-}(K_3 \cup 2K_1)$ -free	
	$K_{1,4}$ -free $\supseteq (K_{1,4}, \text{diamond})$ -free $\supseteq$	$Q$	$\subseteq K_3$ -free $\subseteq (K_4 \cup K_1)$ -free	
	fork-free $\subseteq (P, S_{1,2,5})$ -free		$K_3$ -free $\subseteq \text{co-fork-free}$	
	co-dart-free $\supseteq$	$K_3$ -free	$\subseteq \text{dart-free}$	
	co-butterfly-free $\supseteq C_4$ -free $\supseteq$	$Q$	$\subseteq K_3$ -free $\subseteq \text{butterfly-free}$	
	$K_2 \cup K_3$ -free $\supseteq K_3$ -free $\supseteq$	$Q$	$\subseteq C_4$ -free $\subseteq K_{2,3}$ -free	
	$P_5$ -free		$K_3$ -free $\subseteq \text{house-free}$	
	$P$ -free $\supseteq C_4$ -free $\supseteq$	$Q$	$\subseteq K_3$ -free $\subseteq \text{co-P-free}$	
	cricket-free $\supseteq$	$K_3$ -free	$\subseteq \text{co-cricket-free}$	
	bull-free $\supseteq$	$K_3$ -free	[note: bull = co-bull]	
	$C_5$ -free $\supseteq$	$(C_4, C_5)$ -free	[note: $C_5 = \overline{C_5}$ ]	

Table 5.1:  $G$ -free graph classes for  $G$  with up to 5 vertices. Key: *IS-hard*, IS-easy.

intersection of the two corresponding subgraphs is taken. This instance still has the same solution.  $\square$

It is known that the product of perfect graphs is perfect precisely when either one of the graphs is bipartite, or both graphs are (odd-hole, paw)-free [133]. Note that this result does not directly assist the analysis of perfect microstructures, since the product of structures underlying the clause structure is not product of graphs, but a product of relational structures – the variable clauses are added in to the product of the underlying structures, even if these are graphs.

### 5.3.1 Classes related to trees

The **constraint graph** of a binary CSP instance with variables  $V$  and constraints  $C$  is the graph of the directed graph formed by the scopes of constraints, in other words  $\mathcal{G}((V, \{\sigma \mid (\sigma, R) \in C\}))$ .

**Definition 5.35.** A binary CSP instance has **tree structure** if its constraint graph is a tree. A CSP has tree structure if all its instances have tree structure.

Binary CSPs with tree structure are known to be tractable [140, Chapter 7].

**Lemma 5.36.** *If a graph  $G$  contains a cycle with at least 3 vertices as a subgraph, then it contains an induced cycle with at least 3 vertices.*

*Proof.* Suppose distinct vertices  $u_1, u_2, \dots, u_k, u_1$  form a cycle in  $G$ , so that  $k \geq 3$ ,  $\{u_i, u_{i+1}\}$  is an edge for every  $i$ , and  $\{u_k, u_1\}$  is an edge. I show that an induced cycle can be constructed. Let  $X_2 = \{u_1, u_2, u_k\}$ . If  $X_2$  induces a 3-cycle in  $G$ , then let  $X = X_2$  and halt. Otherwise,  $X_i$  is given for some  $i > 2$ , and  $X_i$  does not induce an  $(i+1)$ -cycle in  $G$ . Add  $u_{i+1}$  to  $X_i$  to form  $X_{i+1}$ . If  $u_{i+1}$  is connected to  $u_j$  for some  $j = 1, 2, \dots, i-1$  then let  $j'$  be the largest value in  $\{1, 2, \dots, i-1\}$  such that  $u_{i+1}$  is connected to  $u_j$ , let  $X = \{u_j, u_{j+1}, \dots, u_{i+1}\}$ , and halt. If  $u_{i+1}$  is connected to  $u_k$  then let  $X = \{u_1, u_2, \dots, u_{i+1}, u_k\}$  and halt. Otherwise increment  $i$  and continue. The algorithm will always halt, since  $u_{k-1}$  is connected to  $u_k$ . As the vertices  $u_i$  were distinct and  $k \geq 3$ ,  $X$  will form an induced cycle with at least 3 vertices, and the result follows.  $\square$

Note that there are several different notions of acyclicity in hypergraphs that in the non-binary case could be used to define an analogue of tree structure. A rather strict notion requires the Gaifman graph of the constraint hypergraph to be a tree. This means that there can be no edges containing 3 or more vertices; the instance must then be binary and its constraint graph must be a tree.

The microstructure of a tree-structured instance is not in general a tree, and nor is the clause structure.

**Example 5.37** (tree structure does not imply tree microstructure). If  $D$  contains at least two elements, then the clause structure of a single binary constraint relation  $\neq_D$  contains  $C_4$

as an induced subgraph, so is not a tree. (Note that it also is not chordal, see Section 5.3.2 for discussion of chordal graphs.)  $\square$

On the other hand, CSP instances with tree structure are perfect.

Let  $\text{TREE}$  denote the class of all binary CSP instances with tree structure.

**Proposition 5.38.**  $\text{CS}(\text{TREE}) \subset (\text{hole}, \text{odd-antihole})\text{-free} \subset \text{perfect}$ .

*Proof.* Let  $G = \text{CS}(\mathcal{P}) \in \text{CS}(\text{TREE})$ . Suppose  $G$  contains a cycle  $C_k$  with  $k \geq 5$ . The vertices of  $C_k$  must then involve at least 3 different variables, since assignments for the same variable are all connected in the clause structure. The structure of the instance  $\mathcal{P}$  therefore contains a cycle, contradicting the assumption that it is tree-structured. Hence  $G$  must be hole-free.

Since  $C_5$  is its own complement, only the giant antiholeholes need to be considered. Assume for contradiction that  $G$  contains a giant antihole  $\overline{C_k}$  with  $k \geq 7$  vertices  $(0, 1, \dots, k-1)$  (in that order around the cycle). This antihole contains an induced  $C_4$  on every 4 vertices  $(i, i+1, j+1, j)$  such that  $0 \leq i < j < k$  and  $i < i+2 < j < j+2 < i+k$  (all numbers modulo  $k$ ). As any induced 4-cycle in the clause structure of a tree-structured problem must involve exactly 2 variables, every set of 4 vertices of this kind in  $\overline{C_k}$  involves exactly 2 variables. This implies that the vertices  $(0, 1, 2, \dots, k-1)$  involve just two variables, which alternate around the cycle. If  $k$  is odd, these conditions are unsatisfiable. Hence  $G$  is odd-antihole-free.

This argument also shows that for every antihole  $\overline{C_{6+2n}}$  with an even number of vertices, there exist tree-structured CSP instances containing  $\overline{C_{6+2n}}$ , so the list of forbidden subgraphs cannot be enlarged by including more holes or antiholeholes.  $\square$

The following result then follows by taking complements.

**Corollary 5.39.**  $\text{MS}(\text{TREE}) \subset (\text{odd-hole}, \text{antihole})\text{-free} \subset \text{perfect}$ .

It is also worth observing that every bipartite graph can be obtained as the microstructure of a CSP instance with two variables, which is always tree-structured.

Note also that a bipartite graph is  $(C_3, C_5, C_7, \dots)$ -free, or equivalently, can be vertex coloured with 2 colours.

**Proposition 5.40.**  $\text{bipartite} = (C_3, \text{odd-hole}, \text{antihole})\text{-free} \subset \text{MS}(\text{TREE})$ .

*Proof.* The inclusion  $(C_3, \text{odd-hole}, \text{antihole})\text{-free} \subseteq \text{bipartite}$  follows from Proposition 5.19. Every antihole  $\overline{C_k}$  with  $k \geq 6$  contains a triangle: for instance, consider vertices  $0, 2, 4$ , when the vertices are labelled  $(0, 1, 2, \dots, k-1)$  in order around the cycle. Then by Proposition 5.19,  $\text{bipartite} \subseteq (C_3, \text{odd-hole}, \text{antihole})\text{-free}$ . Equality follows.

Now every bipartite graph can be 2-coloured. Given a bipartite graph with unique vertex labels, and a 2-colouring of its vertices, associate a CSP instance to the graph. The instance has two variables, one for each vertex colour. The domain of a variable consists of the vertex labels associated with its colour. The constraints specify which combinations



of assignments are allowed, and precisely encode the edges of the graph. This shows that every bipartite graph is in  $\text{MS}(\text{TREE})$ .

For strictness of the inclusion, consider the triangle,  $C_3$ . This graph is not bipartite. However, it is the microstructure of a CSP instance with 3 variables, each variable having a domain of size 1, and no constraints. This CSP instance has the disconnected graph with 3 vertices as its structure, and is therefore tree-structured.  $\square$

### 5.3.2 Classes related to chordal graphs

Recall that  $C_{4+i}$  denotes the infinite set  $\{C_4, C_5, C_6, \dots\}$  of all finite cycles with at least 4 vertices. A graph is called **chordal** or **triangulated** if it is  $(C_{4+i})$ -free. Such graphs may contain a cycle with at least 4 vertices as a subgraph, but not as an induced subgraph. Hence every cycle with at least 4 vertices must have an edge between two of its vertices that are not adjacent in the cycle. Such an edge is called a **chord**. A graph is called **co-chordal** if its complement is chordal. All chordal and co-chordal graphs are perfect [86].

A graph is **weakly chordal** if it is (hole, antihole)-free [80]. A weakly chordal graph may contain 3-cycles or 4-cycles.

Jégou noted that the class of binary constraint problems with chordal microstructure form a tractable class [97]. Cohen noted that binary constraint problems with chordal clause structure also form a tractable class, consisting of problems that are “permutably max-closed” [34, 72]. Both of these classes are perfect, and can be combined to obtain a larger tractable class.

Let  $\text{CC}$  denote the class of graphs that are either chordal or co-chordal, and let  $\text{CCMC}$  denote the class of all binary CSP instances with chordal microstructure or chordal clause structure.

**Proposition 5.41.**  $\text{MS}(\text{CCMC}) = \text{CS}(\text{CCMC}) = \text{CC}$

$\subset (\text{hole}, \text{antihole})\text{-free} = \text{weakly chordal}$

$\subset (\text{odd-hole}, \text{odd-antihole})\text{-free} = \text{perfect}.$

*Proof.* Any graph can be obtained as the microstructure of a trivial CSP instance. The domain of each variable contains exactly one value. There are binary constraints between every pair of distinct variables, with the constraint relation either empty or containing a single edge. By symmetry between edges and non-edges, any graph can then also be obtained as the clause structure of such an instance. This shows that  $\text{CC} \subseteq \text{MS}(\text{CCMC})$  and  $\text{CC} \subseteq \text{CS}(\text{CCMC})$ . On the other hand,  $\text{MS}(\text{CCMC})$  and  $\text{CS}(\text{CCMC})$  are contained in  $\text{CC}$  by definition, which proves the equalities between these classes.

The antihole with 5 vertices is isomorphic to  $C_5$ , and all larger antiholes contain an induced 4-cycle. Chordal graphs must then be antihole-free, as they are  $C_4$ -free. The remaining inclusions follow from Proposition 5.19.

To see that the inclusions are strict, first consider the domino graph (illustrated in Figure 5.7), which is (hole, antihole)-free, but neither chordal nor co-chordal. Sec-



Figure 5.7: The domino graph and its complement.

ond, consider the 6-cycle: this is (odd-hole, antihole)-free, but it is clearly a hole so not (hole, antihole)-free.  $\square$

### 5.3.3 Classes related to gridline graphs

The vertices of a **gridline** graph can be embedded in the real plane so that there is an edge between two distinct vertices precisely when they both are part of the same horizontal line or the same vertical line. Gridline graphs form the class of (claw, diamond, odd-hole)-free graphs [128].

**Proposition 5.42.**  $CS(\text{ALL-DIFFERENT}) = \text{gridline}$

$\subset (\text{odd-hole}, \text{giant-antihole})\text{-free}$

$\subset (\text{odd-hole}, \text{odd-antihole})\text{-free}.$

*Proof.* Let  $G \in CS(\text{ALL-DIFFERENT})$ . Then  $G = CS(\mathcal{P})$  for some instance  $\mathcal{P}$  of ALL-DIFFERENT. Embed each vertex  $(x, a)$  of  $G$  in the real plane, with the variable  $x$  determining the horizontal position, and the value  $a$  determining the vertical position. The edges of  $G$  disallow multiple assignments of a value to any variable, and disallow the assignment of multiple variables to the same value. These are precisely those edges required by the definition of a gridline graph.

Conversely, for any gridline graph  $G$  there is an associated embedding in the real plane, so map each vertex to a pair  $(x, y)$ , and consider these to be the variable-value pairs of a CSP instance. The graph  $G$  is then the clause structure of an ALL-DIFFERENT instance. Its edges require each variable to be assigned a unique value, which is also different from the value assigned to every other variable. This shows that  $CS(\text{ALL-DIFFERENT}) = \text{gridline}$ .

Consider any giant antihole, with vertices labelled with  $\{0, 1, \dots, n+6\}$  so that there is no edge between vertices  $i$  and  $i+1 \pmod{n+7}$ . The vertices  $0, 1, 3, 5$  then induce a diamond. By Proposition 5.19, diamond-free  $\subset$  giant-antihole-free (strict inclusion follows since the diamond is  $\text{co-}C_{7+i}$ -free). Since gridline = (claw, diamond, odd-hole)-free, it follows that gridline  $\subset$  (odd-hole, giant-antihole)-free. Since  $C_5$  is its own complement, this class in turn is contained in (odd-hole, odd-antihole)-free. The graph  $\text{co-}C_8$  witnesses the strictness of this last containment.  $\square$

A symmetrical argument (replacing every graph  $G$  in the proof by  $\text{co-}G$ , and applying Proposition 5.19) yields complementary inclusions for co-gridline graphs.

**Corollary 5.43.**  $MS(\text{ALL-DIFFERENT}) = \text{co-gridline}$

$\subset (\text{odd-antihole}, \text{giant-hole})\text{-free}$

$\subset (\text{odd-hole}, \text{odd-antihole})\text{-free}.$

By Proposition 5.19,  $(\text{hole}, \text{odd-antihole})\text{-free} \subset (\text{odd-antihole}, \text{giant-hole})\text{-free}$  and  $(\text{odd-hole}, \text{antihole})\text{-free} \subset (\text{odd-hole}, \text{giant-antihole})\text{-free}$ . The graphs  $C_6$  and  $\overline{C_6}$  witness the strictness of these inclusions.

**Example 5.44** (3-variable, 2-value ALL-DIFFERENT). The clause structure of an ALL-DIFFERENT instance with 3 variables, with domains  $\{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_3\}$ , is the even hole  $C_6$ . This is shown in Figure 5.8.  $\perp$

**Example 5.45** (2-variable, 3-value ALL-DIFFERENT). The microstructure of an ALL-DIFFERENT instance with 2 variables and domains  $\{0, 1, 2\}$ , is illustrated in Figure 5.9.  $\perp$

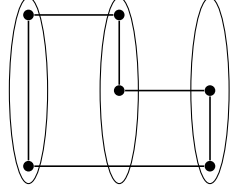


Figure 5.8: Clause structure of an  $(\text{odd-hole}, \text{antihole})\text{-free}$  ALL-DIFFERENT instance.

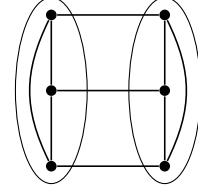


Figure 5.9: Clause structure of a  $(C_5, \text{giant-hole}, \text{giant-antihole})\text{-free}$  ALL-DIFFERENT instance.

The class inclusions of Theorem 5.30, Proposition 5.38, Proposition 5.41, and Proposition 5.42 are summarised in Figure 5.10. Several of the strict separations were discussed in the proofs. I now review the remaining separations.

1. For strict inclusion of  $(\text{hole}, \text{odd-antihole})\text{-free}$  in  $(\text{giant-hole}, \text{odd-antihole})\text{-free}$ , consider  $C_6$  (and  $\overline{C_6}$  separates the complement classes).
2. A tree is planar, so is  $K_{3,3}$ -free. Consider the graph  $2C_3$ , consisting of two disjoint triangles. This graph is the microstructure of a CSP instance with  $K_{3,3}$  structure, but is also  $(\text{hole}, \text{antihole})\text{-free}$ . This demonstrates that the containment of  $\text{MS}(\text{TREE})$  in  $(\text{odd-hole}, \text{antihole})\text{-free}$  is strict.
3.  $\text{CS}(\text{ALL-DIFFERENT})$  is not contained in  $(\text{odd-hole}, \text{antihole})\text{-free}$  as witnessed by  $\overline{C_6}$ , and  $C_6$  in turn witnesses that the class  $\text{CS}(\text{ALL-DIFFERENT})$  is not contained in the class  $(\text{hole}, \text{odd-antihole})\text{-free}$ .
4.  $\text{CS}(\text{ALL-DIFFERENT})$  does not contain the classes of bipartite and co-bipartite graphs, as witnessed by the claw and diamond, respectively.
5. Finally,  $C_4$  and  $\overline{C_4}$  separate chordal and co-chordal from CC (respectively).

Note that  $\text{MS}(\text{TREE})$  is a hereditary class, since removing a vertex from the microstructure will never introduce a new edge in the structure. Removing some vertices may change some existing proper constraints into anything-goes constraints, which would

remove the edges between those variables in the structure. Similarly,  $\text{CS}(\text{TREE})$  is also a hereditary class. The precise set of forbidden substructures remains to be determined: due to the inclusion of this class in the class  $(\text{odd-hole}, \text{antihole})\text{-free}$ , clearly  $\text{MS}(\text{TREE}) = (X, \text{odd-hole}, \text{antihole})\text{-free}$ , for some set  $X$  of graphs.

### 5.3.4 Summary of relationships

In this section I have discussed inclusions between some subclasses of perfect graphs, including several that are obtained as microstructures or clause structures of binary CSPs. The inclusions between these classes are illustrated in Figure 5.10, where lower position indicates smaller classes, the transitive strict subclass relation is indicated by lines, and the absence of a line between two classes indicates that they are incomparable.

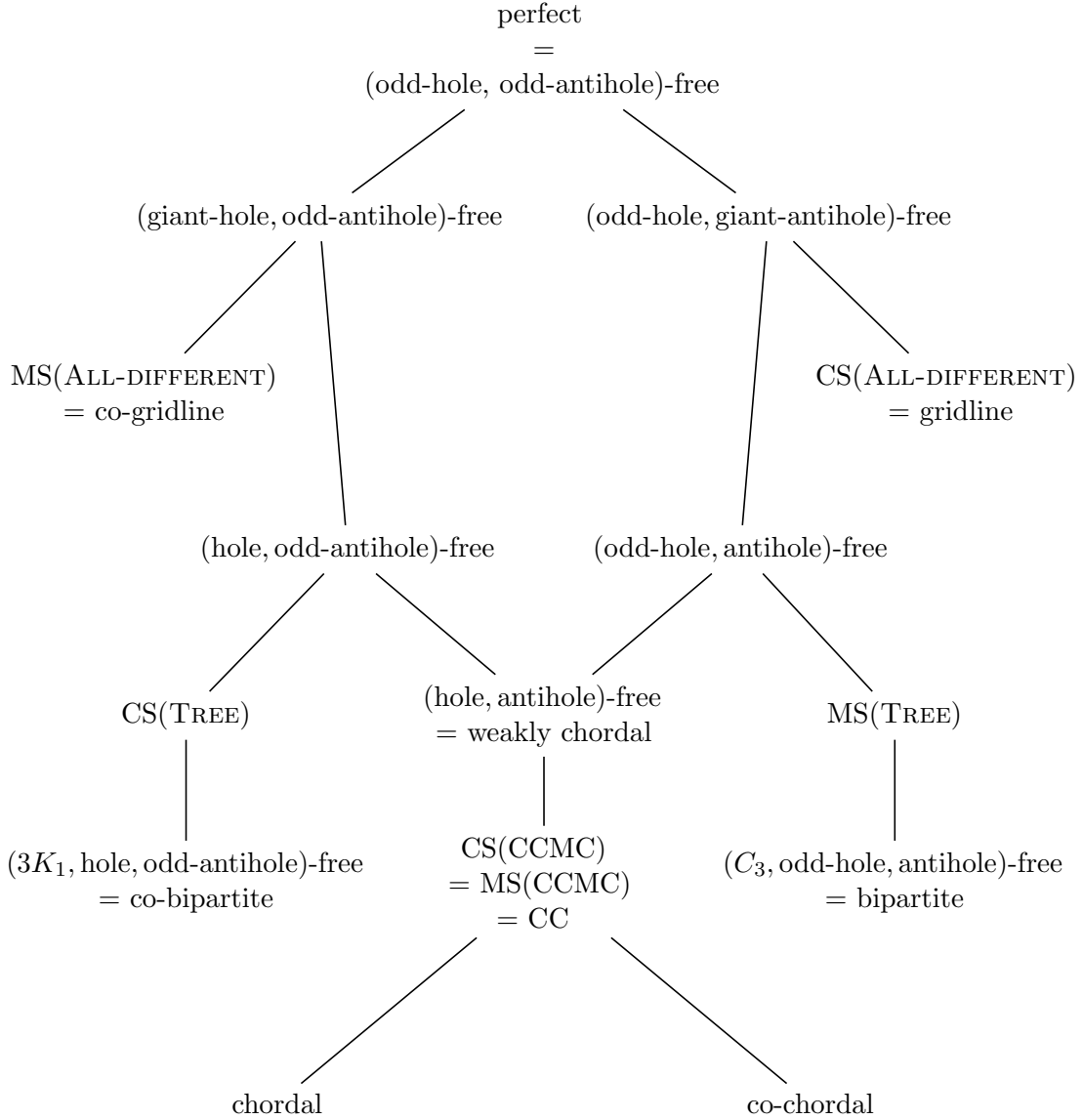


Figure 5.10: Strict inclusions among CSP-derived subclasses of perfect graphs.

A less detailed version of Figure 5.10 was published in [141]. In this chapter I have also shown that all inclusions in the figure are strict.

## 5.4 Summary and contribution

In Section 5.1, I have dealt with transformations of CSP instances to equivalent binary instances. In the constraints community, higher arity constraints are sometimes regarded as unnecessary. This is because a relation of arity greater than 2 can be expressed as a binary relation by introducing a new variable capturing the relatedness of the variables. Rossi et al. argued this point [139], providing a citation to Volume 3 of the collected works of Peirce. The citation seems intended to refer to Peirce’s critique of an earlier paper by Kempe, where Peirce acknowledges Kempe’s contention that higher arity relations can be expressed in terms of binary ones [127, 100]. I have used excerpts from these papers as quotations at the start of each chapter of this thesis.

Following Peirce, I believe that higher arity constraints are important. Higher arity constraints are useful for those global constraints which cannot naturally be decomposed into relations of fixed arity, for global constraints that can be so decomposed but where this requires introducing a large number of new variables, and for global constraints which cannot be decomposed while still preserving the ability to enforce GAC efficiently. Further, there may exist tractable classes of CSPs of unbounded arity which in any decomposition either lead to a class that is not tractable, or which superpolynomially increase the instance size. However, the focus in this and the next chapter is on the case where decompositions into binary constraints do exist, and are meaningful.

The microstructure of a binary CSP instance is a graph. This observation implicitly underlies Regin’s 1994 algorithm for establishing GAC of ALL-DIFFERENT+UNARY constraints [134]. Binary CSP instances employing just one constraint relation were also used in the dichotomy result of Hell and Nešetřil. They showed that CSPs with structures that are non-bipartite graphs are NP-complete. In contrast, a CSP with structures that are bipartite graphs is decidable in linear time: there is a solution precisely when the target graph is not completely disconnected [82]. The foundational work of Feder and Vardi, showing that constraint problems defined by constraint languages can be expressed in the MMSNP fragment of logic, also related the main relationships into questions about homomorphisms between graphs. The graph-theoretic point of view, and binary CSPs, are therefore central to much of the work in constraint satisfaction.

Montanari originally introduced the exact 2-section as a “minimal network” of an instance [122]. Constraints that are decomposable into their exact 2-sections have been called “network decomposable” [122, 47]. I have followed Gent et al. in using the term “decomposable” instead [62].

Copeland and Khoshafian discussed the Decomposition Storage Model for relational databases, in which the data relating to each attribute is stored in a separate table [45].

(This model is now usually referred to as a column-oriented database.) Each such table stores key-value pairs. The keys serve to tie together the different attribute values across tables: those attribute values associated with the same key belong together in a tuple identified by the key. This is conceptually similar to the hidden transformation, with attributes acting as variables, and the key attribute in each table acting as a single hidden variable corresponding to the table of solutions. However, the challenge in constraint satisfaction is to determine if the table of solutions is empty or not, without joining together all the individual constraints, whereas the Decomposition Storage Model starts with the table of solutions and is motivated by support for missing values in data records and improving performance for certain kinds of database queries.

Section 5.2 discusses the importance of hereditary properties of the microstructure. This leads to the conclusion that forbidden substructures in the microstructure are of fundamental importance as a way of capturing properties that are preserved during enforcement of arc-consistency.

Some previous results were important stepping stones for the results in Section 5.3. It is well-known that tree-structured CSP instances can be solved efficiently [48]. Also well-known is that the ALL-DIFFERENT constraint allows efficient solution [103, 134]. Furthermore, Gutin, Rafiey, and Yeo implicitly observed that perfect microstructure leads to a tractable class [78].

In an earlier published version of the work in this chapter, tree structure was shown to lead to perfect microstructure, and the microstructure of the ALL-DIFFERENT constraint was shown to be perfect [141]. Sellmann independently showed that the clause structure of tree-structured instances is (hole, odd-antihole)-free [144], and therefore perfect.

By a closer analysis of the classes of graphs formed by microstructures of several classes of CSPs, in Section 5.3 I showed that the relationships in Figure 5.10 hold between classes of perfect graphs obtained from CSPs. Note that Proposition 5.42 and Corollary 5.43 are slightly stronger than previous results. The main improvement is that the classes of (odd-antihole, giant-hole)-free and (odd-hole, giant-antihole)-free graphs now provide more specific upper bounds for the pairs of classes  $\text{CS}(\text{TREE})/\text{gridline}$  and  $\text{MS}(\text{TREE})/\text{co-gridline}$ . I also provide more detail in the proofs in this chapter.

*Mr. Kempe has, and must have, three kinds of elements in his diagrams, namely, one kind of spots, and two kinds of connections of spots.*

—C. S. Peirce, *The Critic of Arguments*, 1892. §423.

# 6

## Variable-coloured binary microstructures

The microstructure representation discussed in the previous two chapters captures the variable clauses associated with each variable by explicit edges added to the clause structure. The clause structure then contains cliques of edges for each variable.

The reason that variable clauses are present in the clause structure is that for each variable, only one literal involving that variable should be used to construct a solution. It is instead possible to enforce this intended semantics by adding additional features to the microstructure representation. The “plain” clause structure can be augmented with colours, where each colour represents one of the variables. Instead of looking for cliques in the microstructure or independent sets in the clause structure, one can then seek *rainbow* cliques or independent sets, where no two vertices have the same colour. It is also possible to order the vertices or each colour class, and to use such orders when specifying hereditary classes of CSP instances.

In this chapter, I examine the microstructure representation augmented with these additional features. The additional features of colours and order allow hereditary classes of structures to be defined with a greater degree of control than the plain microstructure representation. I review some of the foundational results regarding binary classes defined by forbidding small vertex-coloured structures, and provide an overview of CSPs defined by forbidding the variable-coloured microstructure representation from containing a single small induced vertex-coloured graph.

I then discuss a tractable class of CSPs for which a good vertex ordering can be found, and the vertex ordering can be used to solve the instance in polynomial time. This forms a novel tractable hybrid class of CSPs, and is defined by forbidding particular colour-ordered vertex-coloured graphs in the variable-coloured microstructure representation.

## 6.1 Vertex-coloured structures and logic

I now consider enriching graphs with the aim of using them to define hereditary classes that are larger than those that are possible to define just using graphs. Colours can be added to the vertices of a structure, the vertices in each colour class can be ordered, and the colours can also be ordered.

**Definition 6.1** (vc-graphs). A **vertex-coloured graph** or **vc-graph**  $(G, \kappa)$  is a pair consisting of a graph  $G$  and a colouring  $\kappa: V(G) \rightarrow [s]$ , for some positive integer  $s$ . In a vc-graph, the elements of  $[s]$  are also referred to as **colours**.  $G$  is obtained from  $(G, \kappa)$  by **forgetting** the colouring  $\kappa$ . The **colour class** associated with colour  $q \in [s]$  is the set of vertices  $\kappa^{-1}(q) \subseteq V(G)$ .  $\square$

Note that by definition, the colour classes form a partition of the vertices.

The focus in this chapter is on binary CSPs, so I will mostly use the term **vc-graph** instead of the more general terms **vc-structure** or **vc-hypergraphs** which would be relevant for CSPs of arbitrary arity. When vertex-colourings are of interest, I modify the meaning of forbidden substructure to allow one colouring to be used to represent all colourings that retain the same essential structure.

**Definition 6.2** (forbidding vc-graphs). When  $X$  is a vc-graph  $(G, \kappa)$ , I use  $X$ -free to mean  $\{(G, \kappa) \mid \kappa \text{ is any colouring of } G\}$ -free.  $\square$

By Fagin's Theorem, every language in NP can be defined by a finite formula of existential second-order logic [51]. The original proof encodes the nondeterministic Turing machine recognising the language as a relational structure, and then transforms the formula describing the existence of this machine into a formula of the right syntactic form. This line of enquiry was continued by Feder and Vardi, who showed that it was possible to restrict the logic even further [52, 53]. They introduced the class MMSNP of languages by making three syntactic restrictions, and conjectured that this class admits a dichotomy: every language in MMSNP is either NP-complete, or in PTIME. By Ladner's Theorem, if  $P \neq NP$  then there are infinitely many non-equivalent classes of languages between P and the NP-complete languages [110]. Feder and Vardi showed that NP is polynomial-time equivalent to the restricted class SNP (corresponding to formulae in which the existential second-order quantification appear first, and the remaining first-order subformula is universally quantified) together with any two out of the three further restrictions of all existential predicates being of arity one (monadicity), each input relation appearing only in negated form if the formula is written in conjunctive normal form (monotonicity), and not using the disequality predicate. Adding all three restrictions at once yields MMSNP, which is conjectured to admit dichotomy, and therefore to be strictly less powerful than unrestricted existential second-order logic, unless  $P = NP$ .

The traditional descriptive complexity approach has the limitation that the SNP restriction appears to be incompatible with CSPs with unbounded arity. Feder and Vardi



explicitly restricted their attention to languages where a single finite target structure is used to construct each instance in a language, thus avoiding this problem.

Continuing this line of investigation, forbidden pattern problems were introduced by Madelaine and Stewart [115]. Forbidden patterns capture the logical structure of problems in NP using techniques from combinatorial theory. Kun and Nešetřil have shown that this approach can be modified to use any of three different kinds of mappings, generalizing forbidden patterns [108].

These investigations all define classes of instances which are well-behaved, by using a restricted fragment of logic to define these classes. In this chapter I consider defining classes of instances via forbidden induced vertex-coloured structures. This can be thought of as a more restrictive kind of mapping than the three kinds that were studied by Kun and Nešetřil, and was originally motivated by such concerns, although I leave investigation of such mappings for further work.

A class defined by a single forbidden induced substructure  $S$  is equivalent to the intersection of several larger classes defined by coloured versions of  $S$ . Distinguishing vertices by means of colours therefore allows many more classes to be defined than can be defined by means of forbidden non-coloured structures. Further, forbidding ordered induced subgraphs defines hereditary classes of graphs that need infinitely many forbidden induced subgraphs to characterize [65]. Colouring and order are therefore powerful tools to define hereditary classes of structures.

I first define a rather simpler framework of terminology and notation in terms of vertex-coloured structures. As my motivation is different to forbidden patterns, this simpler machinery will suffice. I then apply this machinery to the microstructure.

Finally, I look at CSPs that are tractable by means of a backtrack-free variable ordering, where this ordering can be obtained in polynomial time by exploiting a forbidden substructure.

## 6.2 Transformations

The **variable-coloured microstructure** is the vc-structure obtained by assigning the variable of each literal in the microstructure as its colour. The **variable-coloured clause structure** is the vc-structure obtained by assigning the variable of each literal in the clause structure as its colour. In this chapter my focus is on transforming a binary CSP instance to its variable-coloured microstructure representation (either the variable-coloured microstructure or the variable-coloured clause structure).

A language characterized by a finite set of forbidden induced substructures is tractable. This generalizes a standard result for graphs.

**Proposition 6.3.** *Let  $L$  be a language in NP. If there is a finite set  $F$  of relational structures such that  $L$  contains precisely all  $F$ -free structures, then  $L \in P$ .*

*Proof.* Suppose  $L = F$ -free for some finite set  $F$  of forbidden substructures, that each structure in  $F$  has at most  $k$  vertices, and that there are  $q$  structures in  $F$ . Given an input structure  $S$  of size  $n$ , it is then possible in time at most  $O(q \cdot n^{k+1})$  to check that none of the structures in  $F$  are substructures of  $S$ . Note that  $q$  is constant, and  $k$  is fixed. This then constitutes an algorithm to recognise  $L$  in polynomial time.  $\square$

From Proposition 6.3 it follows that any class of CSP instances defined by means of finitely many forbidden induced substructures in their microstructures or clause structures is tractable.

Every NP language can be represented via the relational structure of the accepting computation [108]. Inspired by the notions of defining CSPs by means of the combinatorial arrangements of parts of the computation, in this chapter I consider CSPs defined by forbidden vertex-coloured induced substructures in the microstructures or clause structures associated with each instance. The microstructure representation of a CSP instance can be provided as a parameter to a generic Turing machine that checks whether an input partial assignment is consistent with the microstructure. The structure of such a “microstructure oracle” is then largely determined by the microstructure parameter, and it may then be possible to extend results about hereditary classes of microstructures to analyse the properties of such microstructure oracles. This forms a conceptual bridge between the previous work based on analysing the structure of computations and the ideas I explore in this chapter, but I do not explore the details of this link here.

If  $C$  is the class of CSP instances with  $G$ -free clause structures, then it is also the class of CSP instances with  $\mathcal{F}$ -free vertex-coloured clause structures, where  $\mathcal{F}$  contains all vertex-colourings of  $G$ . Using vertex-colouring therefore allows larger classes of hereditary structures to be defined than if only forbidden graphs are used.

**Lemma 6.4.** *Suppose  $\mathcal{F}$  contains all possible vertex colourings of  $S$ . Then the class obtained by forgetting the colourings of the  $\mathcal{F}$ -free vc-structures is identical to the class of  $S$ -free of structures.*

*Proof.* For the forward containment, consider a vc-structure  $G$  obtained by forgetting the colouring of some vc-structure  $(G, \kappa') \in \mathcal{F}$ -free and suppose  $G$  contains a copy of  $S$ . Then for every colouring  $\kappa$  of  $G$ ,  $(G, \kappa)$  contains at least one of the vc-structures in  $\mathcal{F}$ . Hence  $(G, \kappa')$  is not  $\mathcal{F}$ -free, a contradiction.

For the reverse containment, consider a structure  $G \in S$ -free, and suppose that there is no colouring  $\kappa$  such that  $(G, \kappa) \in \mathcal{F}$ -free. Then for every colouring  $\kappa$ , there is some  $(H, \kappa) \in \mathcal{F}$  such that  $(G, \kappa)$  contains  $(H, \kappa)$ ; this means that  $G$  contains  $S$ , a contradiction.  $\square$

It is possible to illustrate Lemma 6.4 graphically. In pictures of vc-graphs, I use the convention that distinct colours are denoted by distinct shapes oriented vertically, enclosing small circles that represent vertices of the structure  $\begin{pmatrix} \circ \\ \circ \\ \circ \end{pmatrix}$  and Table 6.1 illustrates many

small vc-graphs that all have the property that the vertices in each colour class induce a clique. I have chosen not to use actual colours to denote colour classes.

**Definition 6.5.** The *clique-completion* of a vc-graph is a vc-graph with the same vertices and colours, but with edges between any pair of distinct vertices in each colour class. The *IS-reduction* of a vc-graph is a vc-graph with the same vertices and colours, but with all edges between vertices in the same colour class removed.

The clique-completion also applies to general vc-structures, not just vc-graphs, but I will not pursue this possibility here. Applying the clique-completion to a class of vc-structures ensures that the class of structures corresponds precisely to the class of variable-coloured clause structures of some class of CSP instances. However, it is important to note that allowing colours in the forbidden vc-graphs used to define hereditary classes of vc-graphs gives rise to larger classes than can be defined if forbidden subgraphs are used.

Using clique-completions, in Table 6.1 I list each class of vertex-coloured clause structures of binary CSP instances that is defined by a forbidden subgraph in the clause structure of up to 4 vertices. For each graph  $G$ , a list of vc-graphs  $\mathcal{F}$  is given, so that the  $G$ -free clause structures are precisely the  $\mathcal{F}$ -free vertex-coloured clause structures. I assume that the clique-completion has been applied to each class of vertex-coloured clause structures before the vc-graphs containing the particular forbidden induced vc-subgraphs are removed.

$G$	$\mathcal{F}$	$\mathcal{F}$	$G$
		,	
		,  ,	
	,	,	
		,  ,  ,	
	,	,  ,  ,	
	,	,  ,  ,	
	,	,  ,  ,	
	,  ,  ,	,  ,  ,	
	,  ,  ,	,  ,  ,	
	,  ,  ,	,  ,  ,	

Table 6.1:  $G$ -free clause structure =  $\mathcal{F}$ -free vertex-coloured clause structure (with clique-completions assumed)

In Table 6.1, only one vc-graph is included in  $\mathcal{F}$  from each isomorphism class of vc-graphs. Moreover, vc-graphs that cannot occur as variable-coloured clause structures have been omitted from each  $\mathcal{F}$  in the table; when two vertices in vc-graph  $G'$  in  $\mathcal{F}$  are in the same colour class, but do not correspond to any edge in  $G$ , then  $G'$  can be omitted from  $\mathcal{F}$ .

### 6.3 Rainbow independent sets

Recall (see Example 2.20) that a proper colouring of a graph  $G$  is a vertex-colouring  $\kappa$  of  $G$  such that if  $\{u, v\} \in E(G)$  then  $\kappa(u) \neq \kappa(v)$ .

The colouring of a vc-structure does not have to be proper: there is no requirement that vertices appearing in a tuple in the structure must have at least two, or any specific number, of distinct colours. However, in a vc-structure it is natural to ask whether there exist independent sets with specific properties of the colours. For instance, it is possible to ask whether there exists an independent set of a given size, such that all vertices have the same colour. Here I only consider the problem of deciding whether a vc-structure contains an independent set of a given size, such that the vertices all have different colours.

**Definition 6.6** (rainbow set). A **rainbow set** in a vc-structure is a set of vertices with every vertex having a different colour.  $\square$

Rainbow subsets can be used to represent nogoods in variable-coloured clause structures, or props in variable-coloured microstructures.

The requirement that the independent set be a function, for it to be a solution, can be captured either by adding additional variable clauses to the hypergraph to form the clause structures considered in Chapter 4 and Chapter 5. This requirement can also be captured by means of requiring the independent set to be a rainbow set.

Two decision problems form the vertex-coloured analogues of INDEPENDENT SET for graphs and hypergraphs.

#### RAINBOW INDEPENDENT SET (**RIS**)

Input: vc-structure  $(S, \kappa)$ , integer  $k$

Question: is there a rainbow independent set with  $k$  elements in  $S$ ?

#### COMPLETE RAINBOW INDEPENDENT SET (**CRIS**)

Input: vc-structure  $(S, \kappa)$

Question: is there a rainbow independent set in  $S$  that uses all colours in  $\{\kappa(u) \mid u \in V(S)\}$ ?

Versions of these problems involving cliques instead of independent sets are denoted **RC** and **CRC**, respectively, but I will focus on the independent set versions.

To capture the semantics of CSPs, the edges in a colour class are irrelevant. If we are considering the MS semantics, then edges within any colour classes are implicitly forbidden by the rainbow condition and can therefore be ignored, while in the CS semantics, the same issue means that any missing edges can be added without changing the problem. Essentially, in the MS we can require the colour classes to be independent sets, and in the CS to form cliques, forming canonical vc-graphs in the class of instances of CRC and CRIS, respectively, which are equivalent to the CSP.

Consider CS semantics. Suppose a colouring  $c$  of graph  $G$  has some missing edges, so that  $(G, \kappa)$  differs from the clique-completion of  $(G, \kappa)$ . Then  $c$  can be thought of as containing new information.

There are two kinds of information contained in a colouring. The first is to restrict the vertices in a solution to one representative per colour class. The second is to introduce new edges.

In a CSP, if arc-consistency has been established, then there can be no assignments  $(u, a)$  that are connected to each allowed assignment of values to a different variable  $v$ . The cliques forming colour classes in the corresponding instance of CRIS are therefore maximal. Analogously, knowing which cliques form the colour classes therefore helps with solving RIS and CRIS, even if the colouring does not contain new information about which vertices cannot simultaneously participate in an independent set. Essentially, we can preprocess the graph with an AC-like procedure, possibly removing some vertices if they cannot participate in a solution, and then solve the problem for the smaller instance that remains. So, a colouring first allows an AC-like procedure to be used to preprocess the graph.

The second function of a colouring is to introduce new implied edges. Since an instance of RIS or CRIS will have the same solution as an instance that has been clique-completed, we could solve the clique-completed instance instead. For graphs this increases the size of the instance by a polynomial amount, in the extreme case of a completely disconnected graph with  $n$  vertices with a single colour class to represent  $n^2$  edges with only  $2n$  space. However, for hypergraphs the new edges can be superpolynomial in number, changing the underlying complexity of the problem. A vertex-coloured hypergraph may therefore represent an instance of RIS or CRIS more compactly than the corresponding IS instance, and a set of IS instances might be represented as RIS instances that are compact enough to require superpolynomially more time when expressed in terms of the instance size.

Instead of examining each of the  $\binom{n}{p}$  possible choices of a size  $p$  independent set out of  $n$  vertices, the presence of a colouring may reduce the time taken even for a brute-force approach. If  $k \geq p$  colours are used, and each colour class contains  $n/k$  vertices, then only  $\binom{k}{p}(n/k)^p$  possible choices need to be examined. Clearly in the degenerate case where every vertex is assigned a different colour, this is the same number of cases. However, even if a greedy domino tiling approach is used to select pairs of vertices connected by an edge to form colour classes, with the remaining vertices being singletons, there will be at least one colour class with two vertices (otherwise the graph is completely disconnected), and for the case of a star graph this is all that can be done. The number of cases to be examined where  $q$  two-element colour classes are created using tiling, is then  $2^q \binom{n-q}{p}$ , which is already a reduction. In general, enumerating the  $k$  colour classes as  $\{1, 2, \dots, k\}$ , there are  $\binom{k}{p} \prod_{i=1}^k |c^{-1}(i)|$  cases to be considered.

## 6.4 Variable ordering

The order in which variables are considered in a search procedure matters a great deal. It is possible to choose a variable ordering that is used throughout the search for solutions [63]. Alternately, the variable ordering can be changed as new information becomes available during search [15].

Finding a good variable ordering, and doing so quickly, are challenging problems in constraint programming, and in many cases finding a suitable variable ordering is NP-hard [123]. For some kinds of problems it is possible to find a good variable ordering. It was realised early on that instances with tree structure had this feature, and that this extended to bounded treewidth [49, 57]. It is possible to extend this to more general classes of problems. In particular, if the microstructure has a special structure then it is possible to find a good variable ordering efficiently. This variable ordering can then be used to obtain the solutions to the problem without requiring any backtracking during search. Freuder investigated some sufficient conditions for avoiding backtracking [55].

Finding an ordering of vertices in a graph to ensure “nice” properties is in general a difficult problem. For instance, many graph algorithms deal with the vertices in some particular order, and backtrack as necessary; for some kinds of problems having the “right” order available would allow the algorithm to find the solution efficiently, without doing any backtracking. A way to determine such orderings for an NP-complete graph problem is therefore either unlikely to exist, or finding a good ordering is going to be NP-hard itself. In some special cases it is possible to find good vertex orderings.

Perfect elimination orderings are available when the graph is chordal [137]. This leads to an efficient algorithm to solve CSPs with chordal microstructure complements [34, 155].

Perfectly orderable graphs are those graphs for which there is an **admissible ordering** of vertices, such that the resulting graph is obstruction-free. Here **obstruction** is the directed graph  $([4], \{(1, 2), (2, 3), (4, 3)\})$ . Recognizing perfectly orderable graphs is NP-complete [120]. Further, the complexity of finding admissible orderings is not known [135]. Note that perfectly orderable graphs are perfect, and generalize chordal graphs.

$b$ -perfect graphs are similarly well-behaved. These graphs do allow induced  $C_5$  subgraphs, so they are not perfect, but they can contain no larger odd holes or antiholes. The class of  $b$ -perfect graphs is characterized by 22 forbidden induced subgraphs [85].

For some classes of graphs, it is possible to find an ordering of the vertices which leads to an efficient greedy algorithm. For instance, an ordering by non-increasing number of neighbours is enough to guarantee that the greedy algorithm finds the largest independent set, for a class defined by six forbidden induced subgraphs [116].

However, for even quite restricted classes, it is not known how to find a good ordering other than by exhaustive search. For instance, perfectly orderable graphs form a subclass of perfect graphs defined by the existence of some order which forbids a particular ordered induced substructure, yet no efficient algorithm is known to determine such an ordering [84].

I will demonstrate in Section 6.7 that a good variable ordering does exist for a class of

CSPs with clause structures forming hereditary classes of vc-graphs.

## 6.5 Reductions

Clique-completion does not change the solutions of an instance of RIS, and the same holds for IS-reduction of RC instances.

**Lemma 6.7.** *Suppose  $(G_0, \kappa)$  is the clique-completion (IS-reduction) of vc-graph  $(G, \kappa)$ . Then the graph  $G_0$  contains an  $s$ -independent set ( $s$ -clique) if, and only if,  $(G, \kappa)$  contains a rainbow independent set (rainbow clique) with  $s$  elements.*

*Proof.* Suppose  $G_0$  contains an independent set  $X$  of the required size. Since  $(G_0, \kappa)$  is the clique-completion of  $(G, \kappa)$ , no two vertices in  $X$  can be in the same colour class in  $(G, \kappa)$ . Therefore  $X$  forms a rainbow set in  $(G, \kappa)$ . Moreover, since  $X$  is independent in  $G_0$ , and  $G$  is a subgraph of  $G_0$ ,  $X$  must also be an independent set in  $G$ , and hence in  $(G, \kappa)$ .

For the reverse implication, suppose  $X$  is a rainbow independent set with  $s$  elements in  $(G, \kappa)$ . Let  $u$  and  $v$  be distinct elements of  $X$ . It follows that  $\kappa(u) \neq \kappa(v)$ , so  $u$  and  $v$  are not affected by the clique-completion of  $(G, \kappa)$ . Since  $X$  is an independent set in  $G$ ,  $\{u, v\}$  is not an edge of  $G$ , and therefore is also not an edge of  $G_0$ . Hence  $X$  is an  $s$ -independent set in  $G_0$ .

The result for IS-reductions and rainbow cliques holds by complementation.  $\square$

This leads to a close relationship between binary CSP instances, CRIS instances, and RIS instances.

**Proposition 6.8.** *The classes of binary CSP instances, CRIS instances, and RIS instances are all polynomially equivalent.*

*Proof.* The class of binary CSP instances reduces to CRIS by the variable-coloured clause structure construction.

Clearly the class of CRIS instances reduces to the class of RIS instances by simply setting the integer threshold in the RIS instance to be the number of colour classes.

To reduce the class of RIS instances to the class of binary CSP instances, suppose the RIS instance has integer threshold  $s$  and input vc-graph  $(G, \kappa)$ . Construct a CSP instance  $(K_s, \overline{G_0})$ , where  $G_0$  is obtained by first taking the clique-completion of  $(G, \kappa)$  and then forgetting the colours. This CSP instance has a solution precisely when  $G_0$  contains an  $s$ -independent set, and by Lemma 6.7, precisely when  $(G, \kappa)$  contains a rainbow independent set with  $s$  elements.  $\square$

**Definition 6.9.** The vc-graph  $(G, \kappa)$  is a **vc-subgraph** of vc-graph  $(H, \lambda)$ , denoted  $(G, \kappa) \subseteq (H, \lambda)$ , if  $V(G) \subseteq V(H)$ ,  $E(G) \subseteq E(H)$ , and when  $x \in V(G)$  then  $\kappa(x) = \lambda(x)$ .  $(G, \kappa)$  is an **induced vc-subgraph** of  $(H, \lambda)$  if  $(G, \kappa)$  is a vc-subgraph of  $(H, \lambda)$ , and also whenever  $\{u, v\} \notin E(G)$  then  $\{u, v\} \notin E(H)$ . If  $(G, \kappa)$  is an induced vc-subgraph of  $(H, \lambda)$  then  $(H, \kappa)$  is **contained** in  $(G, \kappa)$ .

*Observation 6.10.* Proposition 5.19 holds without modification for vc-structures also.

*Observation 6.11.* If for every  $j = 1, 2, \dots, q$ , there is some  $i = 1, 2, \dots, p$  such that  $G_i$  is an induced subgraph of  $H_j$ , then  $(G_1, G_2, \dots, G_p)$ -free  $\subseteq (H_1, H_2, \dots, H_q)$ -free.

*Observation 6.12.* If for every  $j = 1, 2, \dots, q$ , there is some  $i = 1, 2, \dots, p$  such that  $(G_i, \kappa_i) \subseteq (H_j, d_j)$ , then

$$((G_1, \kappa_1), (G_2, \kappa_2), \dots, (G_p, \kappa_p))\text{-free} \subseteq ((H_1, d_1), (H_2, d_2), \dots, (H_q, d_q))\text{-free}.$$

Proposition 5.22 holds without modification for vc-structures. I will rephrase this for vc-graphs, analogously to Corollary 5.23 for graphs. As with Definition 5.21, I use terminology to explicitly distinguish the decision problem for a class of vc-graphs from the promise problem of deciding CRIS relative to the promise that the input vc-graph is from a class of vc-graphs.

**Definition 6.13.** Class  $C$  of vc-graphs is **CRIS-hard** if COMPLETE RAINBOW INDEPENDENT SET is NP-complete when the input vc-graph is restricted to be from  $C$ , and **CRIS-easy** if COMPLETE RAINBOW INDEPENDENT SET can be decided in polynomial time when the input vc-graph is restricted to be from  $C$ .

**Corollary 6.14.** Let  $X \subseteq Y$  be classes of vc-graphs.

1. If  $X$  is CRIS-easy, then  $Y$  is CRIS-easy.
2. If  $Y$  is CRIS-hard, then  $X$  is CRIS-hard.

## 6.6 Forbidden vc-graphs

Graphs obtained by the microstructure complement mapping have special structure. Those assignments associated with a variable form a clique in the CS. Every graph can be obtained as the CS of some (generally unsolvable) CSP instance, but it is not clear whether every graph that has an independent set of size  $n$  can be obtained as the CS of a binary CSP instance with  $n$  variables (probably no).

A **hereditary class** of graphs is one that is closed under induced subgraphs (so if a graph  $G$  is in the class, then so is every induced subgraph of  $G$ ). Hereditary graph classes are the natural ones to consider when considering microstructures, since the basic CSP operation of **propagation** relies on imposing successive unary constraints when consistency information is discovered, and unary constraints correspond precisely to taking induced subgraphs of the microstructure or the microstructure complement. In some sense, then, a class of microstructures or microstructure complements associated with a CSP must be hereditary for the class to have relevance to constraint satisfaction.

In Table 6.2, Table 6.3, and Table 6.4 I classify as CRIS-easy or CRIS-hard each of the classes of CSPs with variable-coloured clause structures that are defined by forbidding a



single small induced vc-subgraph. *Moreover, I am assuming that clique-completion has been applied to each clause structure in the class before the substructures in the table are forbidden.* This is an important caveat and has a consequence that needs to be highlighted. The  $K_3$ -free graphs form an IS-hard class, so by consulting Table 6.1, each of the classes of vc-graphs  $\emptyset$ -free,  $\text{P}_2$ -free, and  $\text{P}_3$ -free are all CRIS-hard. However,  $\emptyset$ -free variable-coloured binary clause structures are actually CRIS-easy, since these are simply the clause structures of the tractable CSP 2-SAT. The discrepancy arises because clique-completion modifies the vc-graphs in the class. If the forbidden induced vc-subgraph is first excluded and the remaining vc-graphs are clique-completed, then many vc-graphs with large colour cliques will be present in the class. If the clique-completion is first performed, the resulting class vc-graphs has only cliques in each colour class, and forbidding a colour clique with 3 vertices will forbid vc-graphs with 3 or more vertices in each colour clique.
















































	CRIS-easy. Every instance is empty ( $s = 0$ ), so contains no non-trivial instances.
	CRIS-easy. All domains have at most 1 element, so solvable trivially.
	CRIS-easy. Each literal is incompatible with literals involving any other variable. Solvable trivially if $s = 1$ , has no solutions if $s \geq 2$ .
	CRIS-easy. All constraints are anything-goes, therefore every rainbow set is a prop, solvable trivially.
	CRIS-easy. All domains have at most 2 elements; this is 2-SAT.
	CRIS-easy. Each literal is compatible with at most one literal involving any other variable. Can be solved in polynomial time by assigning one variable to each of the possible domain values, and then checking whether this singleton partial assignment can be extended to a solution.
	CRIS-easy. Each literal is compatible with zero or all literals involving any other variable, so either has no solutions or every rainbow set is a prop, solvable trivially.
	CRIS-hard, since the disequality relation contains no such induced substructures, and hence CS(GRAPH $t$ -COLOURING) is contained in the class.
	CRIS-easy. This forbids any rainbow IS with 3 or more vertices. Hence there are no solutions if $s \geq 3$ ; for $s \leq 2$ exhaustive search is at most quadratic.
	CRIS-easy, by [44, Proposition 3.8].
	CRIS-easy, by [44, Corollary 3.7].
	CRIS-hard, since GRAPH 3-COLOURING over $K_3$ -free graphs is NP-complete, see [44, Proposition 3.10].

Table 6.2: Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 1, 2, or 3 vertices.

---

	CRIS-hard, since CS(GRAPH 3-COLOURING) is contained in this class.
	Not yet determined.
	CRIS-hard, since GRAPH 3-COLOURING is contained in this class.
	CRIS-easy. Each constraint relation is closed under the dual discriminator, see [95, Proposition 5.3].
	CRIS-easy by [39], since each assignment is compatible with zero, one, or all assignments to any other variable.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and Lemma 6.4 and $C_4$ -free being IS-hard.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-easy. Whenever $\phi$ is a prop with two literals, then there is precisely one way to extend it to a prop involving a third variable.
	Not yet determined.
	Not yet determined.
	CRIS-hard, by Corollary 6.14 and  -free.
	Not yet determined.
	Not yet determined.
	CRIS-easy, by Theorem 6.18 via any variable ordering.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and Lemma 6.4 and $C_4$ -free being IS-hard.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.


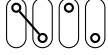


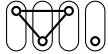










---

Table 6.3: Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 4 vertices and up to 3 colours.

These tables are new, but they are based on extended discussions with Martin Cooper, Peter Jeavons, Chris Jefferson, and Stanislav Živný. Several of the CRIS-easy entries have non-obvious proofs and references are provided for these.

Table 6.2 classifies the structures with up to 3 vertices. Table 6.3 provides a summary of what I know about structures with 4 vertices and up to 3 colours, and Table 6.4 covers structures with 4 vertices and 4 colours. Note that most of the known CRIS-hard entries are consequences of a few entries using Corollary 6.14.

---

	CRIS-easy. This forbids any rainbow IS with 4 or more vertices, so there is no solution if $s \geq 4$ ; for $s \leq 3$ exhaustive search takes at most cubic time.
	Not yet determined.
	Not yet determined.
	Not yet determined.
	CRIS-hard, by Corollary 6.14 and  -free.
	Not yet determined.
	Not yet determined.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and Lemma 6.4 and $C_4$ -free being IS-hard.
	CRIS-hard, by Corollary 6.14 and  -free.
	CRIS-hard, by Corollary 6.14 and  -free.

---

Table 6.4: Complexity of CRIS, clique colour classes. Forbidden induced vc-subgraphs with 4 vertices and 4 colours.

## 6.7 Broken-triangle property

I now apply the theory developed so far in this chapter to study a hybrid tractable class of CSPs. This class is defined in terms of a hereditary class of vertex-coloured graphs with ordered colours, where the forbidden induced subgraphs also specify an ordering on their colours.

Some of the results of this section have been published previously [41, 42], and constitute joint work with Martin Cooper and Peter Jeavons. My presentation here is somewhat different, and emphasizes the vertex-coloured framework.

The broken triangle property is a hybrid property that strictly generalizes tree structure. Moreover, CSPs with the broken triangle property can be solved in polynomial time.

In essence, the broken triangle property can be thought of as a kind of transitivity condition. By processing the vertices in the right order, the broken triangle property can

be reduced to a tree-like condition, and an algorithm akin to those used for solving tree structured CSPs can then be applied to find a solution. Moreover, a suitable such ordering of variables can be found efficiently.

The broken-triangle property is a variant of perfectly orderable graphs which forbids certain kinds of ordered obstructions only. The difference is that no ordering is required between the literals with the same variable; only the variables are ordered. This reduction in what is demanded of an ordering allows the BTP to be recognised in polynomial time. Due to the reduced requirements, an ordering of the variables can also be found, with the nice property that it avoids obstructions.

Given a variable ordering  $\leq$ , for every pair of variables  $u < v$  I denote by  $R_{uv}$  the constraint relation associated with scope  $(u, v)$  once normalisation has been performed (see Section 4.4.1). If there are multiple relations associated with variables  $u$  and  $v$ , then they are combined together; if variables  $u$  and  $v$  do not appear together in any scope, then then  $R_{uv}$  expresses the anything-goes constraint relation.

**Definition 6.15** (canonical constraint relation). Given a CSP instance  $(V, D, C)$  with an ordering  $\leq$  on  $V$ , for each pair  $u < v$  from  $V$  the **canonical constraint relation** for  $u$  and  $v$  is

$$R_{uv} = \bigcap (\{R \mid ((u, v), R) \in C\} \cup \{R \mid ((v, u), R) \in C\}),$$

where an empty intersection is defined to be  $D \times D$ . ┘

It greatly simplifies arguments to be able to refer to the canonical constraint relation  $R_{uv}$  without considering whether the scope involving  $u$  and  $v$  is  $(u, v)$ ,  $(v, u)$ , or doesn't appear at all in the instance. The canonical constraint relations assume that the explicit version of the instance is being used.

**Definition 6.16.** A binary CSP instance with  $s$  variables satisfies the **broken-triangle property** or **BTP** with respect to the variable ordering  $\leq$ , if whenever  $1 \leq u < v < w \leq s$  and for every  $a, b, c, d \in D$ , if  $(a, b) \in R_{uv}$ ,  $(a, c) \in R_{uw}$  and  $(b, d) \in R_{vw}$ , then either  $(a, d) \in R_{uw}$  or  $(b, c) \in R_{vw}$ . ┘

Note that the broken-triangle property must be satisfied for all triples  $u < v < w$ , even if the description of the instance does not specify a constraint between variables  $u$  and  $v$ . If there is no constraint specified in the instance between  $u$  and  $v$ , then  $R_{uv}$  does not restrict the allowed values.

The BTP can also be expressed as a forbidden induced vc-subgraph in the vertex-coloured microstructure or clause structure of the CSP instance, with respect to some fixed ordering of the variables of the instance. Figure 6.1 illustrates the forbidden BTP induced substructure in the microstructure and the clause structure. I also illustrate a combined representation, also called a “flat pattern” in subsequent work [36], in which dashed edges are edges in the variable-coloured clause structure and solid lines are edges in the variable-coloured microstructure. Such substructures are forbidden whenever  $u < v < w$ .

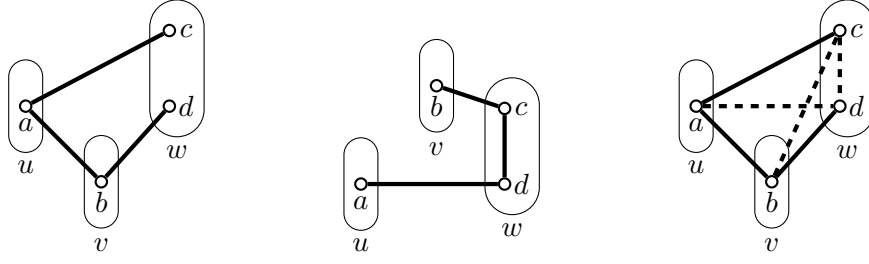


Figure 6.1: The BTP forbidden structure: in the vertex-coloured microstructure, in the vertex-coloured clause structure, combined diagram.

The following restatement of the BTP will be needed later. This shows that the BTP can be characterized as forcing a pattern of subset containments between images of the canonical relations in the BTP forbidden pattern.

**Lemma 6.17.** *A binary CSP instance satisfies the broken-triangle property with respect to variable ordering  $\leq$  if, and only if,*

$$\forall u < v < w \quad \forall (a, b) \in R_{uv} \quad R_{uw}(a) \subseteq R_{vw}(b) \vee R_{vw}(b) \subseteq R_{uw}(a).$$

The broken-triangle property is invariant under domain reductions, as it is a hereditary property. Note that establishing arc-consistency can be done by a sequence of domain reductions, removing each unsupported value in turn [59]. Therefore, if a CSP instance satisfies the BTP then it will still satisfy the BTP after arc-consistency has been established. This is key to the following result.

**Theorem 6.18.** *It is possible to determine, in polynomial time, if there exists a solution to a binary CSP instance satisfying the BTP with respect to a fixed variable ordering  $\leq$ .*

*Proof.* Colour the literals of the microstructure by a colouring  $\kappa$  that assigns to each literal  $(u, a)$  the name of its variable  $u$ . For convenience, suppose that the set of variables is the set  $[s]$  with the usual ordering  $\leq$  on integers.

If an instance has the BTP with respect to  $\leq$ , then establishing arc-consistency preserves the BTP. Arc-consistency can be established in polynomial time [16]. If the result of establishing arc-consistency is that some colour class is empty, then the instance has no solution. Assume therefore that the CSP instance is arc-consistent and that each colour class is non-empty.

For each variable  $u \in [s]$ ,  $\text{dom}(u) = \pi_1(\kappa^{-1}(u))$  is then the set of values in the colour class corresponding to colour  $u$ . Consider the first variable. Since  $\text{dom}(1) \neq \emptyset$ , the partial assignment  $\phi = \{(1, a)\}$  is a prop for any value  $a \in \text{dom}(1)$ .

It is then enough to show, for all  $w = 2, \dots, s$ , that any prop  $\phi$  involving the first  $w - 1$  variables can be extended to a prop involving the first  $w$  variables. Extending props to a complete assignment then results in a solution.

I now show that there is some literal  $(m, a)$  such that the literals that are its neighbours in the microstructure provide at least one candidate to extend  $\phi$ . In particular, each of the images  $R_{mu}(a)$  is non-empty, and any  $w$ -coloured neighbour of the literal  $(m, a)$  suffices to make progress.

By Lemma 6.17, if  $u < v < w$  then either  $R_{uw}(\phi(u)) \subseteq R_{vw}(\phi(v))$  or  $R_{vw}(\phi(v)) \subseteq R_{uw}(\phi(u))$ . Thus the set  $\{R_{uw}(\phi(u)) \mid u < w\}$  is totally ordered by subset inclusion, and hence has a minimal element  $R_{mw}(\phi(m)) = \bigcap_{u < w} R_{uw}(\phi(u))$  for some  $m < w$ . Further,  $R_{mw}(\phi(m)) \neq \emptyset$  by arc-consistency. By the definition of  $R_{uw}(\phi(u))$ , it follows that  $\phi \cup \{(w, a)\}$  is a prop, for any choice of  $a \in R_{mw}(\phi(m))$ .

Generating the next literal at each step can be performed in polynomial time, and there are  $s$  such steps, so the overall time is polynomially bounded.  $\square$

Theorem 6.18 is especially interesting because a suitable variable ordering can be determined efficiently, if it exists. This is important because a CSP instance may satisfy the BTP with respect to one variable ordering, but not another. Recall also from Section 6.4 that finding a “good” variable ordering (for various notions of “good”) is generally NP-complete. The proof that a “good” variable ordering exists proceeds by constructing an auxiliary CSP instance that determines a suitable variable ordering. The constraints of this instance are all max-closed, and hence a solution can be found in polynomial time.

**Theorem 6.19.** *Finding an ordering of the variables of a binary CSP instance which satisfies the broken-triangle property (or determining that no such ordering exists) can be done in polynomial time.*

*Proof.* Suppose the original instance has variables  $[s]$  and  $t$  values in its domain  $D$ . Construct an auxiliary CSP instance with the variables  $o_1, o_2, \dots, o_s$ . Each variable takes a value in  $[s]$ , which can be regarded as being ordered by the usual order on the integers. A solution  $\phi: \{o_1, o_2, \dots, o_s\} \rightarrow [s]$  represents the relative positions in the variable ordering. Define the ternary relation  $R \subseteq [s]^3$  as  $R = \{(u, v, w) \mid w < \max\{u, v\}\}$ .

The constraints of the auxiliary CSP instance have scopes that are triples of variables, and each constraint has the constraint relation  $R$ . For every distinct  $u, v, w \in [s]$ , the constraint  $\langle (o_u, o_v, o_w), R \rangle$  is imposed if there are  $a, b, c, d \in D$  such that the set of literals  $\{(u, a), (v, b), (w, c), (w, d)\}$  induces a vc-subgraph isomorphic to the forbidden BTP vc-subgraph in the variable-coloured microstructure of the original instance, with respect to the variable ordering  $u < v < w$ . This constraint forces either  $w < u$  or  $w < v$ , thus preventing the BTP vc-subgraph from appearing.

If in a solution to the auxiliary CSP instance,  $o_u = o_v$  for some  $u \neq v$ , then a topological sort of the solution still satisfies all the constraints. This can be done in linear time. A solution to the auxiliary CSP instance can then be taken as a variable ordering of the original instance with respect to which the original instance satisfies the BTP.

There are at most  $t^4$  possible choices of values  $a, b, c, d$  to check for each of the at most  $s(s-1)(s-2)$  ways of arranging the three distinct variables  $u, v, w$ , and these are the only

possible constraints in the auxiliary CSP instance. Therefore the size of the auxiliary CSP instance will be bounded by a polynomial in the size of the original instance.

Now suppose  $(u_1, v_1, w_1)$  and  $(u_2, v_2, w_2)$  are two tuples of  $R$ . By the definition of  $R$ ,  $w_1 < \max\{u_1, v_1\}$  and  $w_2 < \max\{u_2, v_2\}$ , so

$$\max\{w_1, w_2\} < \max\{\max\{u_1, u_2\}, \max\{v_1, v_2\}\}$$

and therefore  $(\max\{u_1, u_2\}, \max\{v_1, v_2\}, \max\{w_1, w_2\}) \in R$ . Hence  $R$  is max-closed by Definition 3.33, and therefore the auxiliary CSP instance can be solved in polynomial time, in the size of the auxiliary CSP instance.

It then follows that the auxiliary CSP instance can be solved in polynomial time in the size of the original instance. By construction, if it has a solution representing some variable ordering, then the original instance admits a variable ordering that satisfies the broken-triangle property with respect to this ordering, and if there is no such variable ordering, then the original instance does not have a variable ordering with respect to which it satisfies the BTP.  $\square$

I now demonstrate that if the structure of a binary CSP instance forms a tree, then it satisfies the BTP. This shows that the BTP is a non-trivial property of CSPs.

**Lemma 6.20.** *Suppose a binary CSP instance with domain  $D$  has a variable ordering  $\leq$  so that whenever  $u < v < w$  are variables, then either  $R_{uw}$  or  $R_{vw}$  is a complete relation. Then the instance has the BTP.*

*Proof.* Consider such a CSP instance. If  $R_{uw}$  is a complete relation, then  $R_{uw}(a) = D$  for any  $a \in D$ , while if  $R_{vw}$  is a complete relation, then  $R_{vw}(b) = D$  for any  $b \in D$ . In either case, by Lemma 6.17 the instance has the BTP.  $\square$

Recalling Definition 5.35, the structure of a binary CSP instance is a tree if the graph of the relational structure formed by the scopes is a tree. More precisely, for every pair of distinct variables  $u$  and  $v$ , if  $R_{uv}$  is a proper relation, then  $\{u, v\}$  is an edge of the graph of the relational structure formed by the scopes. However, if  $R_{uv}$  is a complete relation then  $\{u, v\}$  is still regarded as an edge of this graph, as long as there is a scope  $(u, v)$  or  $(v, u)$  in the instance. It is possible to broaden the class of instances to allow such anything-goes constraints.

**Definition 6.21.** A binary CSP instance has **weak tree structure** if its constraint graph has tree structure when anything-goes constraints are disregarded.  $\perp$

Note that if binary anything-goes constraints are added to a CSP instance with tree structure, the resulting instance may no longer have tree structure. However, the transformed instance will have weak tree structure.

**Proposition 6.22.** *If a binary CSP instance has weak tree structure, then it satisfies the BTP with respect to any variable ordering in which each variable precedes its children.*



*Proof.* In a tree, the vertices can be ordered from any vertex designated as the root to the leaves, for instance using depth-first search of the tree starting at the root. In a binary CSP instance with tree structure, suppose in variable ordering  $\leq$  each variable precedes its children in  $\leq$ . Then for every variable  $w$  there is at most one variable  $u$  with  $u < w$ , such that  $R_{uw}$  is not a complete relation: if  $R_{uw}$  is not a complete relation, then  $u$  must be the immediate ancestor of  $w$  in the tree. By Lemma 6.20 the CSP instance then satisfies the BTP with respect to  $\leq$ .  $\square$

It could be argued that binary CSPs with tree structure are already known to be tractable because they have bounded treewidth. However, there are other binary CSPs which do not have bounded treewidth structures, yet still satisfy the BTP.

**Definition 6.23** (right monotone relation). A binary relation  $R$  over  $D$  is **right monotone** with respect to the total order  $\leq$  on  $D$ , if for every  $a, b, c \in D$ , if  $b < c$  and  $(a, b) \in R$ , then  $(a, c) \in R$ .  $\square$

Right monotone relations are defined with respect to an ordering of the domain. This can be combined with an ordering of the variables to form a subclass of CSPs that satisfy the BTP.

**Definition 6.24.** A binary CSP instance is **renamable right monotone** with respect to a variable ordering  $\leq$  if for every variable  $w$  there is an ordering  $\leq_w$  on  $D$  such that  $R_{uw}$  is right monotone with respect to  $\leq_w$  for every  $u < w$ .  $\square$

**Proposition 6.25.** A CSP instance that is renamable right monotone with respect to a variable ordering  $\leq$  also satisfies the BTP with respect to  $\leq$ .

*Proof.* Suppose the CSP instance is renamable right monotone with respect to variable ordering  $\leq$ . Let  $w$  be any variable. There is then an ordering  $\leq_w$  of the domain  $D$ , such that for every variable  $u < w$ , the canonical constraint relation  $R_{uw}$  is right monotone with respect to  $\leq_w$ .

Now let  $u$  and  $v$  be any variables such that  $u < v < w$ ; I will show that the BTP condition is satisfied for this triple of variables. Each of  $R_{uw}$  and  $R_{vw}$  is right monotone with respect to  $\leq_w$ . Suppose  $(a, b) \in R_{uw}$ ,  $(a, c) \in R_{uw}$ , and  $(b, d) \in R_{vw}$ . If  $c <_w d$  then  $(a, c) \in R_{uw}$  by the right-monotonicity of  $R_{uw}$  with respect to  $\leq_w$ . On the other hand, if  $d <_w c$  then  $(b, c) \in R_{vw}$ , by the right-monotonicity of  $R_{vw}$  with respect to  $\leq_w$ . As the choice of variables was arbitrary, the BTP is satisfied by the instance.  $\square$

The CSP instance with clause structure in Figure 6.2 is not renamable right monotone with respect to any variable ordering. Moreover, each of the constraint relations is proper so this instance does not have weak tree structure. However, this instance does satisfy the BTP since it contains no induced BTP substructure (see Figure 6.1).

As binary CSPs with weak tree structure have the BTP, and moreover every renamable right monotone binary CSP instance has the BTP, the BTP is therefore a non-trivial hybrid property of CSPs that guarantees tractability.



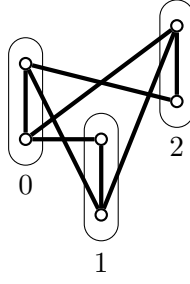


Figure 6.2: Variable-coloured clause structure of instance that is not renamable right monotone and does not have tree structure, yet satisfies the BTP w.r.t. variable ordering  $0 < 1 < 2$ .

## 6.8 Summary and contribution

In this chapter I developed a general framework of vertex-coloured structures. I showed that it is possible to obtain larger hereditary classes of structures by using vertex-colouring to be more precise in what structures are forbidden. Although the framework is general enough to encompass arbitrary classes of hypergraphs, I have phrased my work in terms of vertex-coloured graphs.

Using somewhat different terminology, I previously published the results in Section 6.7 with Martin Cooper and Peter Jeavons [41, 42]. I have also worked on further developments of these ideas, in collaboration with Dave Cohen, Martin Cooper, Páidí Creed, and Dániel Marx, leading to a further paper [36].

Vertex-coloured graphs have been deeply investigated but the focus has been on proper colourings and their hypergraph analogues [153]. In contrast, here I focus on the application of colourings to keep separate the variables in the microstructure representation of a CSP instance. I have demonstrated some progress toward a theory that is appropriate for this application. Further progress seems to require additional general results about classes of vc-structures that forbid vc-subgraphs where all colour classes are cliques. To the best of my knowledge, such clique-completed vc-graphs have not been studied before.

I showed that interesting classes of vertex-coloured graphs can be defined by the absence of some induced vc-subgraphs, where the vc-graphs in question may also have ordered vertices or colours. For the unordered case, I provided an overview of classes defined by single small vc-graphs of up to four vertices, and classified these classes by whether CRIS can be decided in polynomial time for such classes of vc-graphs, or whether CRIS is NP-complete even when the input vc-graph is restricted to be from one of these classes. Several of the four-vertex cases are still open, but all graphs up to three vertices have been classified.

I finally applied these notions of vc-graphs to binary CSPs to show that the broken-triangle property defines a non-trivial class of CSPs that are tractable for hybrid reasons. The tractability of this nontrivial class is guaranteed by the existence of a well-behaved variable ordering, that in turn can be determined efficiently.

*It can readily be shown that every form which admits of representation by means of graphical units and lines of different sorts, can, by the introduction of additional or auxiliary graphical units, be represented by means of graphical units and links only. It will be convenient for the purpose of description to suppose that the lines of different sorts are of different colours. In the place of any red line substitute a red unit (i.e., one coloured red, instead of having a letter inside), this unit being connected by two links to the two units which were joined by the red line ...  $\lambda$  — **RED** —  $\mu$  ... similarly in the place of blue, yellow, &c., lines, substitute blue, yellow, &c., units, joined by links to the units which were connected by the several coloured lines.*

—A. B. Kempe, *A Memoir on the Theory of Mathematical Form*, 1886. §68.



## Complete representation

In this chapter I briefly discuss the **complete representation**, a further representation of CSP instances. The aim of this representation is for every scope of a given arity to appear in the instance, but to restrict the set of variables in each scope to appear in precisely one constraint. The complete representation is a special case of the homomorphism representation where all constraints have the same arity. In the complete representation, the hypergraph of the source structure  $S$  in the instance  $(S, T)$  contains every possible edge of the chosen arity. This is achieved by choosing as scopes the lexicographically smallest tuples from each class of tuples that are permutations of each other.

When discussing width measures of CSP instances, the width of an instance is by convention the width of the hypergraph of the instance.

### 7.1 Complete structures

**Definition 7.1.** Let  $V$  be an ordered  $s$ -element set, let  $2 \leq r \leq s$  be a positive integer, and let  $I = \binom{V}{r}$  be the set of increasing  $r$ -tuples of  $V$ . The **complete**  $(r, s)$ -**structure** with base set  $V$  is the relational structure  $CS(r, s) = (V, (Q_\sigma)_{\sigma \in I})$ , where  $Q_\sigma = \{\sigma\}$ .  $\square$

In a complete structure, the specific names for elements of the base set are not important, so all complete  $(r, s)$ -structures are isomorphic, and the base set can be assumed to be  $[s]$  with the usual order on integers. Note that the hypergraph of a complete  $(r, s)$ -structure is  $([s], \binom{[s]}{r})$ , containing all  $r$ -edges of the base set.

**Definition 7.2.** The **complete representation** of a CSP instance with maximum arity  $r \geq 2$  and  $s$  variables is a special case of the homomorphism representation, of the following

form: the source is a complete  $(r, s)$ -structure, and the target is a relational structure containing  $\binom{s}{r}$  (not necessarily distinct) relations, all of arity  $r$ .  $\square$

Every CSP instance of maximum arity  $r$  can be transformed to an instance in the complete representation that has the same infrastructure, other than possibly some differences in the  $r$ -sections of their infrastructures. For each constraint with arity less than  $r$ , pick any  $r$ -subset of variables that includes its scope, add the remaining variables in the subset to the scope, and expand the constraint relation to allow any value to be assigned to the added variables. This means also that anything-goes constraints are added to all scopes of arity  $r$  that are not constrained. If multiple constraints in the original instance end up with the same scope in the modified instance, replace these constraints with a single constraint, its constraint relation being the intersection of the constraint relations of the modified constraints with that scope.

This transformation leads to a new CSP instance with width  $s/r \geq 1$ , even if the original instance had width strictly lower. Hence care needs to be taken when applying structural measures of tractability.

Measures of hypergraph width capture how close a hypergraph is to being tree-like: low width hypergraphs are tree-like, and those with high width are far from tree-like. This is important for constraint satisfaction since CSP instances with structures that are sufficiently tree-like can be solved in polynomial time. More precisely, if the width of all structures of instances in a CSP is bounded, then the tree-like structure of each instance can be exploited to decide whether an instance in the class has a solution in polynomial time, and the width parameter determines the degree of the polynomial. Width measures studied to date form a hierarchy, with the most general being the submodular width. For each known width measure, if the measure is bounded for a class of CSP instances, then so is the submodular width [118]. Adaptive width is within a constant factor of the submodular width, so is closely related [119]. For CSP instances in the complete representation, there is in fact no difference between adaptive width, submodular width, or fractional hypertree width (see Corollary 7.12).

## 7.2 Decompositions

I now briefly discuss the fractional hypertree width of CSPs in the complete representation, and in Proposition 7.11 I compute the exact fractional hypertree width for a CSP instance in the complete representation. A worst-case analysis of tractable classes that is purely structural must therefore be able to deal with recovering any hidden low-width structure in a CSP instance, even when the instance has been transformed to the complete representation and its width has potentially been increased by such a transformation.

**Definition 7.3.** A *tree decomposition* of a relational structure  $S = (V, (Q_i)_{i \in I})$  is a tuple  $(T, B)$ , where  $T = (J, F)$  is a tree and  $B = (B_j)_{j \in J}$  is a family of subsets of  $V$ , each referred to as a *bag*, such that

1.  $\bigcup_{j \in J} B_j = V$ ,
2. for every  $i \in I$ , if  $(v_1, v_2, \dots, v_r) \in Q_i$  then there is some  $j \in J$  with  $\{v_1, v_2, \dots, v_r\} \subseteq B_j$ , and
3. for every  $v \in V$ , the set of vertices  $\{j \in J \mid v \in B_j\}$  induces a subtree of  $T$ .  $\square$

Informally, a tree decomposition contains every element of the base set in some bag, the elements of each tuple in the structure are contained in some bag, and the bags that contain each element of the base set of the structure form a subtree.

The **width** of a tree decomposition is one less than the largest cardinality of a bag in the decomposition. The **tree-width** of a structure is the minimal width across all of its tree decompositions.

The third condition of tree decompositions guarantees a useful property.

**Definition 7.4** ([68, p. 92]). A collection  $\{S_i \mid i \in J\}$  of subsets of a set  $X$  has the **Helly property** if whenever  $I \subseteq J$  such that  $\bigcap_{i \in I} S_i = \emptyset$ , then there exist  $j, k \in I$  such that  $S_j \cap S_k = \emptyset$ .  $\square$

**Lemma 7.5** ([68, Proposition 4.7]). *Any collection of induced subtrees of a tree has the Helly property.*

**Corollary 7.6.** *If  $T = (J, F)$  is a tree and  $(T, (B_j)_{j \in J})$  is a tree decomposition of a hypergraph  $(V, E)$ , then  $\{\{B_j \mid v \in B_j\} \mid v \in V\}$  has the Helly property.*

It is often more convenient to use the contrapositive form of the Helly property: given a collection of sets with the Helly property, for any pairwise intersecting subcollection there must be some element that is common to every set in the subcollection.

The Helly property means that tree decompositions cannot separate cliques.

**Proposition 7.7** ([19, Lemma 3.1]). *In any tree decomposition of a graph  $G$ , if  $V \subseteq V(G)$  induces a clique in  $G$ , then there exists a bag of the decomposition that contains  $V$ .*

Proposition 7.7 also applies to relational structures containing complete  $(r, s)$ -structures as substructures. The proof is essentially the same as Bodlaender's proof for graphs.

**Proposition 7.8.** *In any tree decomposition of a relational structure that contains a complete structure with base set  $V$  as a substructure, there exists some bag that contains  $V$ .*

*Proof.* For any tree decomposition of a relational structure with bags  $\{B_j \mid j \in J\}$ , the collection  $\{\{j \in J \mid v \in B_j\} \mid v \in V\}$  of subsets of  $J$  has the Helly property, by the definition of tree decomposition and by Lemma 7.5.

Since the structure contains a complete structure,  $\{j \in J \mid u \in B_j\}$  and  $\{j \in J \mid v \in B_j\}$  must intersect for every distinct  $u$  and  $v$ . By the Helly property, there is then some  $i \in J$  such that  $i \in \{j \in J \mid v \in B_j\}$  for all  $v \in V$ , so  $B_i$  contains  $V$ .  $\square$

Note that Proposition 7.8 relies on complete structures having arity at least 2. If a complete  $(1, s)$ -structure were admissible by Definition 7.1, it would permit a tree decomposition (really a forest decomposition) in which the bags were all disjoint singletons.

The proof of Proposition 7.8 only requires that for every pair  $u$  and  $v$  of distinct vertices from  $V$ , there is some edge in the hypergraph of the structure containing both  $u$  and  $v$ . The result therefore extends to cliques in the Gaifman graph of a relational structure.

**Corollary 7.9.** *In any tree decomposition of a relational structure  $T$  that contains a clique  $V$  in its Gaifman graph, there exists some bag that contains  $V$ .*

Hypertree decompositions reduce bag size by considering edges as single elements in a bag [71]. This can make a large difference when the object being decomposed has large edges. Crucially, a hypertree decomposition consists of a tree decomposition together with a different measure of its width. The width of a hypertree decomposition is calculated in terms of the number of edges required to cover any bag, instead of the number of vertices. Proposition 7.8 then extends also to hypertree decompositions, so that in any hypertree decomposition of a relational structure  $T$  that contains a clique  $V$  in its Gaifman graph, there exists some bag that contains a set of edges that together contain every vertex in  $V$ .

Fractional hypertree decompositions provide an even finer notion of decomposition.

**Definition 7.10** ([73]). For hypergraph  $H = (V, E)$  and mapping  $\gamma: E \rightarrow [0, 1]$ , let

$$B(\gamma) = \{v \in V \mid \sum_{e \in E, v \in e} \gamma(e) \geq 1\}.$$

A fractional hypertree decomposition of a hypergraph  $H = (V, E)$  consists of a tuple  $(T, B, \Phi)$ , where  $T = (J, F)$  is a tree,  $B = \{B_j \subseteq V \mid j \in J\}$  is a set of bags, and  $\Phi = (\gamma_j)_{j \in J}$  is a family of fractional guards consisting of mappings  $\gamma_j: E \rightarrow [0, 1]$  for each  $j \in J$ , such that  $(T, B)$  is a tree decomposition of  $H$ , and  $B_j \subseteq B(\gamma_j)$  for every  $j \in J$ . The weight of a fractional hypertree decomposition  $(T, B, \Phi)$  of  $(V, E)$  is  $w(T, B, \Phi) = \max\{\sum_{e \in E} \gamma(e) \mid \gamma \in \Phi\}$ . The **fractional hypertree width** of  $H$  is the minimum weight of a fractional hypertree decomposition of  $H$ .  $\square$

The fractional hypertree width of a structure bounds its hypertree width from below, and also bounds one greater than its tree-width from below [75]. I now establish the fractional hypertree width of complete structures.

**Proposition 7.11.** *The fractional hypertree width of a complete  $(r, s)$ -structure is  $s/r$ .*

*Proof.* Since  $r \geq 2$ , by Proposition 7.8, any tree decomposition of a complete  $(r, s)$ -structure must contain a bag that includes every vertex, say at tree vertex  $j$ .

Hence  $\sum_{X \subseteq V, |X|=r, v \in X} \gamma_j(X) \geq 1$  for every  $v \in V$ . Since  $|V| = s$ ,

$$\sum_{X \subseteq V, |X|=r} \gamma_j(X) = \frac{1}{r} \sum_{v \in V} \sum_{X \subseteq V, |X|=r, v \in X} \gamma_j(X) \geq \frac{s}{r},$$

so the width of any fractional hypertree decomposition is at least  $s/r$ . The fractional hypertree width is then also at least  $s/r$ . This value can be achieved by setting  $\gamma_j(e) = \binom{s-1}{r-1}^{-1}$  for every edge  $e$  of the complete  $(r, s)$ -structure, so the bound is exact.  $\square$

Adaptive width is also defined over tree decompositions, so Proposition 7.8 applies to adaptive width also. The adaptive width differs from the fractional hypertree width in allowing the tree decomposition to vary, but these width measures are the same when there is only one possible tree decomposition [119]. Moreover, the submodular width for CSP instances is between the fractional hypertree width and the adaptive width [118]. Hence the adaptive width, submodular width, and fractional hypertree width are identical for CSP instances in the complete representation.

**Corollary 7.12.** *For a CSP instance with a complete  $(r, s)$ -structure as the structure, the adaptive width, submodular width, and fractional hypertree width are all  $s/r$ .*

In general  $s/r$  is not bounded. On the other hand, classes of CSPs where  $r \geq s/c$  for some constant  $c > 1$  do have bounded fractional hypertree width. Such a class was considered in [75, Example 8], with  $r = s/2$ .

Unbounded fractional hypertree width for complete structures can be guaranteed by making  $r$  a function of  $s$  that grows slowly in  $s$ , such that  $\lim_{s \rightarrow \infty} s/r = \infty$ .

The significance of Corollary 7.12 is that every CSP instance with  $s$  variables and maximum arity  $r$  can be transformed to a solution-equivalent instance in the complete representation, which has fractional hypertree width  $s/r$ . The class of these complete representations may not have bounded fractional hypertree width, even if the original class did. Hence an analysis of the structural tractability of a CSP has to account for the possibility that the class has unbounded width only because it contains constraints that increase the fractional hypertree width of each instance, but do not otherwise contribute to determining the solutions.

### 7.3 Summary and contribution

I have shown that the fractional hypertree width and two related measures of width of a complete  $(r, s)$ -structure are all  $s/r$ . Since every CSP instance with maximum arity  $r$  and  $s$  variables can be transformed into one in the complete representation, with its structure being a complete  $(r, s)$ -structure, care must be taken when analysing CSPs in terms of their structural width.

Transformations to the complete representation may be useful in exploring the limits of using structural restrictions to define tractable CSPs. However, a limitation of the complete representation is the overhead of transforming to it. If the arity of instances is not bounded, then the reduction is unlikely to be possible to polynomially bound, yet proving such lower bounds is a major challenge. Hence the complete representation may only have restricted applicability.

*[T]o say that A gives B to C is to say more than that A gives something to C, and gives to somebody B, which is given to C by somebody. . . . my algebra is perfectly adequate to expressing that A gives B to C . . . This is accomplished by adding to the universe of concrete things the abstraction “this action.” But I remark that the diagram fails to afford any formal representation of the manner in which this abstract idea is derived from the concrete ideas. Yet it is precisely in such processes that the difficulty of all difficult reasoning lies.*

—C. S. Peirce, *The Critic of Arguments*, 1892. §423.

# 8

## Conclusions

### 8.1 Main contributions

In this thesis I have investigated the tractability of classes of constraint satisfaction problems that are not tractable for either language or structural reasons. By reducing such a hybrid CSP to a problem on a different structure, it is possible to make progress for some classes that have eluded analysis via the theory of structural tractability or the theory of language tractability. Both these theories are quite mature, but cannot explain the tractability of the ALL-DIFFERENT constraint.

In Chapter 4 and Chapter 5 I showed that the microstructure and the clause structure can be represented as products of relational structures that are derived directly from the input instance.

I also showed in Chapter 4 that the clause structure is also the structure that is obtained by applying the direct encoding of a CSP instance into SAT, with the resulting domain clauses represented in the clause structure or variable-coloured clause structure by requiring a solution to be sufficiently large. Hence the results in this thesis about the structure of the microstructure representation apply also to SAT instances obtained from CSP instances via the direct encoding. This is likely to be useful in any future analysis of SAT instance structure.

In Chapters 5 to 6 I linked hereditary microstructures and clause structures with classes of CSPs closed under domain reduction operations such as enforcing 1-consistency and arc-consistency. This provided useful new classes of CSPs that are tractable for reasons that cannot be explained by purely language or structural reasons.

I demonstrated in Chapter 5 how the microstructure representation unifies several

tractable classes as subclasses of those CSPs that have microstructures that are perfect graphs. This includes the ALL-DIFFERENT constraint and CSPs with tree structure.

In Chapter 6 I showed how adding ordering of the colours and vertices of the variable-coloured microstructure and clause structure allows finer grained definitions of hereditary classes. For instances that are broken-triangle-free, the additional structure can be used to find a variable ordering which allows a solution to be found efficiently. The class of instances satisfying the broken-triangle property includes all CSPs with weak tree structure and a CSP with unbounded treewidth, so serves as a non-trivial hybrid tractable class to illustrate the applicability of the variable-coloured microstructure representation.

In Chapter 7 I considered another representation of CSPs, which contains every possible scope of some arity, and attains the largest possible fractional hypertree width for a given number of variables and arity.

The techniques introduced in this thesis should assist further development of the theory of hybrid tractable classes of CSPs.

## 8.2 Further work

I now outline several issues for further research that arise from the investigation I have presented in this thesis. Most of these topics have been mentioned in passing in the text, in which case more details are provided here. The presentation is roughly in order that the questions arise in the text.

### 8.2.1 Precise complexity of instance equivalences

In Chapter 3, I discussed the worst-case complexity of infrastructure-equivalence and nogood-equivalence. However, I do not have a lower bound on the worst-case complexity of infrastructure-equivalence: is it co-NP-hard?

Also, the precise worst-case complexity of nogood-equivalence was left open, with this problem being co-NP-hard as a lower bound and in  $\Sigma_2^P$  as an upper bound. Is it actually in co-NP, or is it  $\Sigma_2^P$ -hard?

### 8.2.2 Products have large width

As shown in Chapter 4 and Chapter 6, the clause structure of CSP instances can be expressed as an augmented product. The augmentation consists of adding variable clauses forming cliques, or colours for each variable. The variable clauses ensure high width when arbitrarily large domains are used for instances in the CSP. However, even when large cliques of variable clauses are avoided in the variable-coloured microstructure representation, direct products tend to have large width.

For the binary case the direct product of two paths is a grid, which has treewidth that grows linearly with the number of vertices in the shorter of the paths. This rules out some obvious approaches to defining tractable classes by means of low width clause structures.



It may therefore be worth investigating when large width clause structures can be avoided, or alternately it may be possible to show that the clause structure unavoidably has high width. When the domains are small, then neither the product nor the variable cliques necessarily create high width clause structures, so there may be possibilities for exploiting low width. Conversely, it may be possible to rule out many kinds of approaches by showing that for the case of unbounded domain size the CSP instances tend to have unbounded width clause structures.

### 8.2.3 Applying constraints techniques when microstructure is implicit

In Section 4.3 I mentioned examples of work in computer vision and robotics where systems of mutually compatible assignments were implicitly used, without using the framework of constraint satisfaction explicitly. It seems possible to consider such problems as constraint satisfaction problems, by starting with the microstructure representation that is implicit in such work. The key challenge is to obtain useful CSP instances from individual microstructures. This appears to be a non-trivial challenge but making progress on it may help to provide tools to automate part of the process of constraint modelling.

### 8.2.4 Refining hereditary clause structures

Which literals can be removed during search? In practice, only those literals that participate in no solution are removed by constraint solvers. It would be useful to develop a theory that refines the notion of hereditary clause structures, to incorporate the notion that the only literals to be removed are those that do not form a part of a maximum independent set in the clause structure. There are also links to be made between symmetry breaking and hereditary CSPs: usually the choice of which symmetries to remove is based on specific criteria, and these influence the substructures that can be taken into account.

### 8.2.5 SAT and the clause structure

The clause structure of a CSP instance contains as edges the nogoods that are provided in the instance description. Each of these nogoods corresponds to a clause in the SAT instance obtained by using the direct encoding. SAT clauses are usually regarded as unstructured sets of literals, and a SAT instance is an unstructured set of clauses. However, every SAT instance can be viewed as the clause structure of an underlying CSP. As demonstrated in this thesis, the clause structure has a lot of structure: it is amenable to analysis based on forbidden induced substructures.

Some obvious questions then arise. If one lays bare the structure that underlies the SAT instance, is the SAT solver actually using this structure? Further, is a SAT solver able to implicitly solve the semidefinite programming problem to find the independent set in a perfect graph without needing to use the machinery of mathematical programming and relaxation of a discrete problem to a continuous one?

Viewing a SAT instance as the clause structure of a CSP instance also provides an immediate reduction of SAT to the independent set problem in hypergraphs, or to the vertex-coloured independent set problem in vertex-coloured hypergraphs with at most two vertices in each colour class. This motivates the study of classes of hypergraphs and vc-hypergraphs which are IS-easy or CRIS-easy. Little work has been done in this area, but the prospect of better understanding the behaviour of SAT solvers might provide motivation for further work.

### 8.2.6 Hyperresolution and hereditary classes

Hulme suggested a notion of “extended consistency” for constraint satisfaction problems [88]. Given the direct encoding of a CSP instance into SAT, new variables are introduced for parts of clauses whenever a hyperresolution deduction step can be applied. If new variables are introduced during hyperresolution, then it is possible to bound the arity of clauses to the greater of the size of the largest domain, and the largest cardinality of a clause in the instance. In contrast, applying hyperresolution directly can increase the cardinality of clauses up to the number of variables in the instance (if an assignment of two values to one variable is generated, the clause can immediately be discarded).

Extended consistency can be seen as a transformation of the microstructure that keeps the structure in a well-behaved class. It would therefore be interesting to investigate whether the techniques developed in this thesis could be applied to investigate extended consistency.

Further, Petke and Jeavons related the cardinality of clauses generated by a SAT solver that uses hyperresolution as its basic mechanism to the level of consistency of the CSP instance [129]. It would be interesting to apply the extended consistency technique of introducing new variables to bound the width of clauses, to consistency.

### 8.2.7 Classes beyond hereditary

In Chapter 5 and Chapter 6 I argued that hereditary classes of hypergraphs (or for binary CSPs, of graphs) are a convenient and natural notion for capturing the process of constraint solving, as it usually proceeds by employing domain reduction. Hereditary classes are closed under taking of all induced substructures. However, domain reduction is usually combined with heuristics related to variable ordering, how frequently speculative search is restarted, and which search options to choose. These lead to only some induced substructures being encountered during search, not all.

A more precise model of constraint solving would therefore consider not just classes of microstructures or clause structures that are closed under taking of any induced substructures, but more general classes that are not necessarily “fully” hereditary. It may, for instance, be possible to study a closure operation of classes of structures other than hereditary closure. Given a class of instances with associated clause structures forming class  $C$ , consider the class  $C'$  of clause structures encountered during constraint solving by

a given constraint solver implementation when presented with instances in  $C$  as input. It is possible that  $C \subseteq C'$ , for instance with simple domain reduction, but if the solver may introduce new variables then  $C$  and  $C'$  may be incomparable. The requirement that  $C$  be hereditary can be seen as ensuring that  $C = C'$ . It may be worthwhile investigating more general classes  $C$  that are closed under a stricter notion of substructure that relates to the kinds of transformations of instances that have been found to be useful in practical constraint solver implementations.

### 8.2.8 Incidence graph of microstructure

One can also consider the incidence graph of the microstructure representation. Each clause in the clause structure is a vertex in the bipartite incidence graph, as is each assigning literal. Literals are connected to the partial assignments in which they occur. It would be interesting to investigate how this representation relates to the dual transformation, and which CSPs have hereditary classes in the class of incidence graphs of their clause structures.

### 8.2.9 Binarizing instances

In Section 5.1 I discussed binarizing transformations, that yield a binary instance that is equivalent in some sense to the original. There are many interesting questions related to binarization which are beyond the scope of this thesis. One such question is whether there exists a binary transformation that preserves substructures and is of polynomial size. Such a transformation would allow desirable properties of hereditary classes to be preserved, while allowing the focus of theoretical results to be restricted to binary CSPs.

### 8.2.10 Booleanizing instances

Kolaitis and Vardi have argued that there may be transformations that turn a CSP instance  $\mathcal{P}$  into an equivalent Boolean instance  $\text{Boolean}(\mathcal{P})$ , such that the treewidth of the structure of  $\text{Boolean}(\mathcal{P})$  is low even if the hypergraph of the structure of  $\mathcal{P}$  is far from tree-like [104]. However, it is not clear how to do this in general, or how far it is possible to reduce the treewidth of  $\text{Boolean}(\mathcal{P})$ . Kolaitis and Vardi explicitly raised the question of whether it is possible to find in polynomial time an encoding of  $\text{Boolean}(\mathcal{P})$  achieving a given treewidth  $k$ . The work I presented in this thesis then suggests a further question: can Booleanization be used to transform a class of CSP instances into a class of Boolean instances which is tractable for reasons other than having structures of bounded treewidth? Such an approach might help to avoid some of the difficulties in working with a notion such as treewidth, which has an NP-complete associated decision problem.

### 8.2.11 IS-easy classes of hypergraphs

In Chapter 5 I mentioned the contrast between the large number of interesting graph classes which are known to be IS-easy and the scarcity of hypergraph classes which have been studied in any depth. Due to my focus on finding independent sets in the microstructure representation, this has meant that I have had to focus on the case of binary CSPs. I did find one possibly interesting hypergraph class, which might begin an extension of the techniques in this thesis to higher arity CSPs.

If the clause structures of a class of binary CSPs are  $2K_2$ -free, then the class is tractable and solutions can be found in quadratic time [8]. The argument used to establish this result has been extended to hypergraphs which forbid large induced “hypergraph matchings”, and which therefore form IS-easy classes of hypergraphs [20]. It is not clear whether these classes of hypergraphs correspond to interesting CSPs, but this may be worth investigating.

### 8.2.12 Generalized constraint satisfaction

Consider the generalized constraint satisfaction problem of finding a homomorphism between two relational structures. When the structures can be infinite, then this can model problems that are PSPACE-complete and EXPTIME-complete [81] or even undecidable [131]. Can such a generalized view of constraint satisfaction be used to provide a unified notation for computability theory as well as complexity theory outside NP, the way that constraint satisfaction has provided an important conceptual toolkit for NP? Note that Bodirsky’s investigation of infinite target structures still requires the source structure to be finite [17].

### 8.2.13 Microstructure oracles

In Chapter 6 I remarked that one can consider a parameterized Turing machine that checks whether an input partial assignment is a prop, by comparing it against the parameter of the machine that encodes the microstructure representation of a CSP instance. Such a comparison can be done in polynomial time in the size of the input as long as the cardinality of domains is bounded by a polynomial in the number of variables. Such Turing machines will have much of their structure in common, since they essentially all involve iterating over the literals in the microstructure representation and checking to see whether the input partial assignment is a down-clique or an independent set in the hypergraph. Differences between these machines will be confined to differences between their parameters.

If the class of parameters is strongly structured, for instance by forming a hereditary class, it then seems possible that the theory in Chapter 6 can be translated into results about the combinatorial structure of a single second-order formula capturing the CSP, via the class of parameterized Turing machines. Such a result would potentially give a general reason for dichotomies that have been established [36], perhaps by showing that a formula of MMSNP is sufficient to express such a CSPs.

### 8.2.14 Property testing

Austin and Tao have shown that hereditary properties of structures can be tested with one-sided error [6]. More precisely, classes with hereditary properties can be recognised in polynomial time, with high probability. By iterating, it is possible to trade off a larger degree of the polynomial in return for lower error in the answer, and for some classes of structures it is possible that even better performance is possible than by simple iteration.

It is not yet clear how to turn this non-constructive result into an algorithm for recognition. This appears to be a promising area for further work, as it would have many applications for CSPs based on the links with hereditary classes built in this thesis.

### 8.2.15 Extensions to soft constraints

Hybrid tractable classes can also be studied for constraint satisfaction problems where a weighted notion is used for membership of tuples in relations. This allows many kinds of optimisation problems to be represented within the constraints framework. Classes of CSPs defined by forbidden subproblems were introduced recently [43], and further work along these lines appears promising.

### 8.2.16 Forbidden subgraphs of tree-structured CSP instances

In Section 5.3.1, I showed that the class  $\text{MS}(\text{TREE})$  was strictly contained in the class of (odd-hole, antihole)-free graphs, and itself strictly contains the  $(C_3, \text{odd-hole, antihole})$ -free graphs. Since  $\text{MS}(\text{TREE})$  is hereditary, it is therefore the class of  $(X, \text{odd-hole, antihole})$ -free graphs, for some set  $X$  of graphs. The precise set  $X$  remains to be determined, although it includes  $2C_3$  (the graph formed by the union of two disjoint copies of  $C_3$ ) as an element.

### 8.2.17 Canonical constraint relations for non-binary CSPs

In Section 6.7, I used the canonical constraint relation for binary CSPs. This simplifies many of the arguments involving variable ordering, since it in effect allows the structure of the CSP instance to be treated as an undirected graph. Constructing canonical constraint relations can be done in polynomial time if the CSP has bounded arity.

It would be interesting to determine if using the canonical constraint relation for non-binary CSPs is equally useful. Essentially this allows the structure of the CSP instance to be treated as a hypergraph instead of as a relational structure, thus removing much potentially superfluous symmetry. Such a tool may be especially useful in arguments involving variable ordering. It would also be interesting to explore if it is possible to modify the notion of canonical constraint relation to well-behaved unbounded arity CSPs, avoiding the potentially superpolynomial work required to construct the canonical constraint relations when arity is not bounded.

### 8.2.18 Complete representation, for analysis of width measures

The edge relation in graphs is often represented as a 2-dimensional matrix. It is also possible to represent the hyperedges in a hypergraph using a  $k$ -dimensional structure, if every hyperedge contains precisely  $k$  vertices. The complete representation can then be seen as a higher-dimensional adjacency matrix, specifying for every set of  $k$  variables which combinations of values they may take. If the arity of constraints in a CSP is bounded, then the complete representation has size at most polynomial in the size of the usual representations, and provides a useful and regular representation for tractability results. In particular, it provides worst-case examples for the various structural width measures. It is therefore important when studying structural tractability to consider what would happen if the class of instances were to be transformed to the complete representation. Such a transformation would realise the worst-case width even if the instances started off having low width, thus obfuscating the original reason for tractability. A simple example of such a transformation is the addition of binary anything-goes constraints to a binary CSP instance with tree structure. This may destroy the tree structure of the instance, although the transformed instance will still have weak tree structure.

Given an instance in the complete representation which was obtained by adding constraints to an instance with low width, such that the set of solutions is not changed, is it possible to reverse the process? This problem appears to be intractable, and proving or disproving this would be a useful step in understanding the practical applicability of approaches based on bounded width structures. If inessential changes in representation such as adding constraints that have no impact on the solutions can change a tree-like instance into one with worst-case treewidth, then it would be important to know when this could be undone, so that a brittle property of the instance representation is not relied on to obtain efficient algorithms. Further investigation appears to be warranted.

### 8.2.19 Linear Space Hypothesis and the complete representation

The Exponential Time Hypothesis (ETH) in computational complexity was introduced by Impagliazzo and Paturi [90]. The ETH essentially states that at least exponential time is required for any deterministic Turing machine to decide 3-SAT. The ETH therefore implies  $P \neq NP$ , but also has a range of even stronger consequences. As the only known lower bounds for 3-SAT are linear, since its introduction the ETH and its associated sparsification lemmas [91] have formed an important technique for investigating the structure of NP. On the other hand, disproving the ETH would immediately yield breakthroughs in several long-standing algorithmic challenges [126].

An analogous Linear Space Hypothesis (LSH) can also be defined, essentially stating that any Turing machine (deterministic or nondeterministic) that decides 3-SAT requires at least linear space. ETH trivially implies LSH, but the reverse implication does not appear to hold. LSH implies that 3-SAT is not in NL, and hence  $NL \neq NP$ . This is also implied by  $P \neq NP$ , but  $NL \neq NP$  is not known to imply  $P \neq NP$ . LSH therefore appears

to be weaker than ETH, but still has interesting consequences for complexity lower bounds. Assuming the LSH, it appears possible to construct an infinite hierarchy of CSPs using the complete representation. Each CSP in the sequence can be reduced to the next using logarithmic space, but no element of the sequence can be reduced to the previous element in sub-linear space, unless the LSH fails. This general idea may provide a construction of a strict hierarchy of languages, where strictness of the inclusion is contingent on LSH. This appears to be more direct than Ladner's classical delayed diagonalization technique, which he used to show that if  $NL \neq NP$  then there exists an infinite sequence of languages of strictly increasing hardness between NL and NP-complete [110, Theorem 6]. It is not known whether  $NL \neq NP$  implies the LSH, and it may be that LSH is a strictly stronger hypothesis than  $NL \neq NP$ , so the price to pay for a more constructive proof may be the requirement for a stronger hypothesis. It seems worthwhile pursuing such an explicit construction of a hierarchy of languages.

## References

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] A. P. Ambler, H. G. Barrow, C. M. Brown, R. M. Burstall, and R. J. Popplestone. [A versatile computer-controlled assembly system](#). *IJCAI 1973: Proceedings of the 3rd international joint conference on Artificial intelligence*, 298–307. Morgan Kaufmann, 1973.
- [3] A. P. Ambler, H. G. Barrow, C. M. Brown, R. M. Burstall, and R. J. Popplestone. A versatile system for computer-controlled assembly. *Artificial Intelligence*, **6**(2), 129–156, 1975. [doi:10.1016/0004-3702\(75\)90006-5](#).
- [4] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- [5] A. Atserias, A. Bulatov, and V. Dalmau. On the power of  $k$ -consistency. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *ICALP 2007: Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **4596**, 279–290. Springer-Verlag, 2007. [doi:10.1007/978-3-540-73420-8\\_26](#).
- [6] T. Austin and T. Tao. Testability and repair of hereditary hypergraph properties. *Random Structures and Algorithms*, **36**(4), 373–463, 2010. [doi:10.1002/rsa.20300](#).
- [7] F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, **140**(1–2), 1–37, 2002. [doi:10.1016/S0004-3702\(02\)00210-2](#).
- [8] E. Balas and C. S. Yu. On graphs with polynomially solvable maximum-weight clique problem. *Networks*, **19**(2), 247–253, 1989.
- [9] L. Barto and M. Kozik. Constraint satisfaction problems of bounded width. *FOCS 2009: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, 595–603. IEEE Computer Society, 2009. [doi:10.1109/FOCS.2009.32](#).



- [10] L. Barto and M. Kozik. New conditions for Taylor varieties and CSP. *LICS 2010: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, 100–109, 2010. doi:[10.1109/LICS.2010.34](https://doi.org/10.1109/LICS.2010.34).
- [11] F. Benhamou, editor. *CP 2006: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **4204**. Springer-Verlag, 2006.
- [12] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*. Elsevier, 1989.
- [13] J. Berman, P. L. Idziak, P. Marković, R. McKenzie, M. Valeriote, and R. Willard. Varieties with few subalgebras of powers. *Transactions of the American Mathematical Society*, **362**(3), 1445–1473, 2010. doi:[10.1090/S0002-9947-09-04874-0](https://doi.org/10.1090/S0002-9947-09-04874-0).
- [14] C. Bessière. Constraint propagation. In Rossi et al. [140], chapter 3, 29–83. doi:[10.1016/S1574-6526\(06\)80007-6](https://doi.org/10.1016/S1574-6526(06)80007-6).
- [15] C. Bessière, A. Chmeiss, and L. Saïs. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. *CP 2001: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **2239**, 565–569. Springer-Verlag, 2001. doi:[10.1007/3-540-45578-7\\_40](https://doi.org/10.1007/3-540-45578-7_40).
- [16] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. *IJCAI 2001: Proceedings of the 17th International joint conference on Artificial intelligence*, 309–315, 2001.
- [17] M. Bodirsky. *Constraint Satisfaction with Infinite Domains*. PhD thesis, Humboldt-Universität zu Berlin, November 2004.
- [18] M. Bodirsky and M. Grohe. Non-dichotomies in constraint satisfaction complexity. *ICALP 2008: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, Lecture Notes in Computer Science **5126**, 184–196. Springer-Verlag, 2008. doi:[10.1007/978-3-540-70583-3\\_16](https://doi.org/10.1007/978-3-540-70583-3_16).
- [19] H. L. Bodlaender and R. H. Möhring. The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, **6**(2), 181–188, 1993. doi:[10.1137/0406014](https://doi.org/10.1137/0406014).
- [20] E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan. Extending the Balas–Yu bounds on the number of maximal independent sets in graphs to hypergraphs and lattices. *Mathematical Programming, Series B*, **98**(1–3), 355–368, 2003. doi:[10.1007/s10107-003-0408-4](https://doi.org/10.1007/s10107-003-0408-4).
- [21] S. Bova, H. Chen, and M. Valeriote. Generic expression hardness results for primitive positive formula comparison. *Information and Computation*, **222**, 108–120, 2013. doi:[10.1016/j.ic.2012.10.008](https://doi.org/10.1016/j.ic.2012.10.008).

- [22] A. Bulatov, P. Jeavons, and A. Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing*, **34**(3), 720–742, 2005. doi:[10.1137/S0097539700376676](https://doi.org/10.1137/S0097539700376676).
- [23] A. A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of the ACM*, **53**(1), 66–120, 2006. doi:[10.1145/1120582.1120584](https://doi.org/10.1145/1120582.1120584).
- [24] A. A. Bulatov and P. Jeavons. An algebraic approach to multi-sorted constraints. In Rossi [138], 183–198. doi:[10.1007/978-3-540-45193-8\\_13](https://doi.org/10.1007/978-3-540-45193-8_13).
- [25] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC 1977: Proceedings of the ninth annual ACM symposium on Theory of computing*, 77–90, 1977. doi:[10.1145/800105.803397](https://doi.org/10.1145/800105.803397).
- [26] Y. Chen, M. Thurley, and M. Weyer. Understanding the complexity of induced subgraph isomorphisms. *ICALP 2008: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part I*, Lecture Notes in Computer Science **5125**, 587–596. Springer-Verlag, 2008. doi:[10.1007/978-3-540-70575-8\\_48](https://doi.org/10.1007/978-3-540-70575-8_48).
- [27] M. Chudnovsky, G. Cornuéjols, X. Liu, P. Seymour, and K. Vušković. Recognizing Berge graphs. *Combinatorica*, **25**(2), 143–186, 2005. doi:[10.1007/s00493-005-0012-8](https://doi.org/10.1007/s00493-005-0012-8).
- [28] M. Chudnovsky, N. Robertson, P. Seymour, and R. Thomas. The strong perfect graph theorem. *Annals of Mathematics*, **164**(1), 51–229, 2006. doi:[10.4007/annals.2006.164.51](https://doi.org/10.4007/annals.2006.164.51).
- [29] M. Chudnovsky and P. Seymour. Excluding induced subgraphs. In A. Hilton and J. Talbot, editors, *Surveys in Combinatorics 2007*, volume 346 of *London Mathematical Society Lecture Note Series*, 99–119. Cambridge University Press, 2007.
- [30] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, **4**(4), 305–337, 1973. doi:[10.1016/0012-365X\(73\)90167-2](https://doi.org/10.1016/0012-365X(73)90167-2).
- [31] D. Cohen, M. Cooper, M. Green, and D. Marx. On guaranteeing polynomially bounded search tree size. In Lee [111], 160–171. doi:[10.1007/978-3-642-23786-7\\_14](https://doi.org/10.1007/978-3-642-23786-7_14).
- [32] D. Cohen and P. Jeavons. The complexity of constraint languages. In Rossi et al. [140], chapter 8, 245–280. doi:[10.1016/S1574-6526\(06\)80012-X](https://doi.org/10.1016/S1574-6526(06)80012-X).
- [33] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, **11**(2–3), 115–137, July 2006. doi:[10.1007/s10601-006-8059-8](https://doi.org/10.1007/s10601-006-8059-8).

- [34] D. A. Cohen. A new class of binary CSPs for which arc-consistency is a decision procedure. In Rossi [138], 807–811. doi:[10.1007/978-3-540-45193-8\\_57](https://doi.org/10.1007/978-3-540-45193-8_57).
- [35] D. A. Cohen. Tractable decision for a constraint language implies tractable search. *Constraints*, **9**(3), 219–229, 2004. doi:[10.1023/B:CONS.0000036045.82829.94](https://doi.org/10.1023/B:CONS.0000036045.82829.94).
- [36] D. A. Cohen, M. C. Cooper, P. Creed, D. Marx, and A. Z. Salamon. The tractability of CSP classes defined by forbidden patterns. *Journal of Artificial Intelligence Research*, **45**, 47–78, September 2012. doi:[10.1613/jair.3651](https://doi.org/10.1613/jair.3651).
- [37] D. A. Cohen and M. J. Green. Typed guarded decompositions for constraint satisfaction. In Benhamou [11], 122–136. doi:[10.1007/11889205\\_11](https://doi.org/10.1007/11889205_11).
- [38] C. J. Colbourn. The complexity of completing partial Latin squares. *Discrete Applied Mathematics*, **8**(1), 25–30, 1984. doi:[10.1016/0166-218X\(84\)90075-1](https://doi.org/10.1016/0166-218X(84)90075-1).
- [39] M. Cooper, D. Cohen, and P. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, **65**, 347–361, 1994. doi:[10.1016/0004-3702\(94\)90021-3](https://doi.org/10.1016/0004-3702(94)90021-3).
- [40] M. Cooper and S. Živný. Tractable triangles. In Lee [111], 195–209. doi:[10.1007/978-3-642-23786-7\\_17](https://doi.org/10.1007/978-3-642-23786-7_17).
- [41] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Hybrid tractable CSPs which generalize tree structure. In M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *ECAI 2008, Proceedings of the 18th European Conference on Artificial Intelligence, July 21–25, Patras, Greece*, Frontiers in Artificial Intelligence and Applications **178**, 530–534. IOS Press, 2008. doi:[10.3233/978-1-58603-891-5-530](https://doi.org/10.3233/978-1-58603-891-5-530).
- [42] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination. *Artificial Intelligence*, **174**(9–10), 570–584, June 2010. doi:[10.1016/j.artint.2010.03.002](https://doi.org/10.1016/j.artint.2010.03.002).
- [43] M. C. Cooper and S. Živný. Hybrid tractability of valued constraint problems. *Artificial Intelligence*, **175**(9–10), 1555–1569, 2011. doi:[10.1016/j.artint.2011.02.003](https://doi.org/10.1016/j.artint.2011.02.003).
- [44] M. C. Cooper and S. Živný. Tractable triangles and cross-free convexity in discrete optimisation. *Journal of Artificial Intelligence Research*, **44**, 455–490, 2012. doi:[10.1613/jair.3598](https://doi.org/10.1613/jair.3598).
- [45] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD 1985: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, 268–279. ACM, 1985. doi:[10.1145/318898.318923](https://doi.org/10.1145/318898.318923).
- [46] H. N. de Ridder and other. *Information System on Graph Classes and their Inclusions (ISGCI)*. <http://www.graphclasses.org/>.

- [47] R. Dechter. [On the expressiveness of networks with hidden variables](#). *AAAI 1990: Proceedings of the Eighth National Conference on Artificial Intelligence*, 556–562, 1990.
- [48] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, **34**(1), 1–38, 1987. [doi:10.1016/0004-3702\(87\)90002-6](#).
- [49] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, **38**(3), 353–366, 1989. [doi:10.1016/0004-3702\(89\)90037-4](#).
- [50] R. Dechter and J. Pearl. [Directed constraint networks: A relational framework for causal modeling](#). *IJCAI 1991: Proceedings of the 12th international joint conference on Artificial intelligence*, volume 2, 1164–1170. Morgan Kaufmann, 1991.
- [51] R. Fagin. [Generalized first-order spectra and polynomial-time recognizable sets](#). In R. Karp, editor, *Complexity of Computation. SIAM-AMS Proceedings*, volume 7, 43–73. American Mathematical Society, 1974.
- [52] T. Feder and M. Y. Vardi. Monotone monadic SNP and constraint satisfaction. *STOC 1993: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 612–622, 1993. [doi:10.1145/167088.167245](#).
- [53] T. Feder and M. Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through Datalog and group theory. *SIAM Journal on Computing*, **28**(1), 57–104, 1998. [doi:10.1137/S0097539794266766](#).
- [54] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, **21**(11), 958–966, 1978. [doi:10.1145/359642.359654](#).
- [55] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, **29**(1), 24–32, 1982. [doi:10.1145/322290.322292](#).
- [56] E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, **32**(4), 755–761, 1985. [doi:10.1145/4221.4225](#).
- [57] E. C. Freuder. [Complexity of  \$k\$ -tree structured constraint satisfaction problems](#). *AAAI 1990: Proceedings of the Eighth National Conference on Artificial Intelligence*, 4–9, 1990.
- [58] E. C. Freuder. [Eliminating interchangeable values in constraint satisfaction problems](#). *AAAI 1991: Proceedings of the Ninth National Conference on Artificial Intelligence*, 227–233, 1991.
- [59] E. C. Freuder and A. K. Mackworth. Constraint satisfaction: An emerging paradigm. In Rossi et al. [140], chapter 2, 13–27. [doi:10.1016/S1574-6526\(06\)80006-4](#).

- [60] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA., 1979.
- [61] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, **1**(3), 237–267, 1976. doi:[10.1016/0304-3975\(76\)90059-1](https://doi.org/10.1016/0304-3975(76)90059-1).
- [62] I. Gent, K. Stergiou, and T. Walsh. Decomposable constraints. *Artificial Intelligence*, **123**(1–2), 133–156, 2000. doi:[10.1016/S0004-3702\(00\)00051-5](https://doi.org/10.1016/S0004-3702(00)00051-5).
- [63] I. P. Gent, C. Jefferson, and I. Miguel. [MINION: A fast, scalable, constraint solver](#). *ECAI 2006: Proceedings of the 17th European conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications* **141**, 98–102, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.
- [64] I. P. Gent, I. Miguel, and P. Nightingale. Generalised arc consistency for the AllDifferent constraint: An empirical survey. *Artificial Intelligence*, **172**(18), 1973–2000, 2008. doi:[10.1016/j.artint.2008.10.006](https://doi.org/10.1016/j.artint.2008.10.006).
- [65] M. Ginn. Forbidden ordered subgraph vs. forbidden subgraph characterizations of graph classes. *Journal of Graph Theory*, **30**(2), 71–76, 1999. doi:[10.1002/\(SICI\)1097-0118\(199902\)30:2<71::AID-JGT1>3.0.CO;2-G](https://doi.org/10.1002/(SICI)1097-0118(199902)30:2<71::AID-JGT1>3.0.CO;2-G).
- [66] O. Goldreich. On promise problems: A survey. In O. Goldreich, A. L. Rosenberg, and A. L. Selman, editors, *Theoretical Computer Science: Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, 254–290. Springer-Verlag, 2006. doi:[10.1007/11685654\\_12](https://doi.org/10.1007/11685654_12).
- [67] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, **12**(4), 516–524, 1965. doi:[10.1145/321296.321300](https://doi.org/10.1145/321296.321300).
- [68] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Annals of Discrete Mathematics **57**. Elsevier, 2nd edition, 2004.
- [69] G. Gonthier. [Formal proof—the four-color theorem](#). *Notices of the AMS*, **55**(11), 1382–1393, 2008.
- [70] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *PODS 1999: Proceedings of the eighteenth ACM SIGACT-SIGMOD symposium on Principles of database systems*, 21–32. ACM, 1999. doi:[10.1145/303976.303979](https://doi.org/10.1145/303976.303979).
- [71] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, **64**(3), 579–627, 2002. doi:[10.1006/jcss.2001.1809](https://doi.org/10.1006/jcss.2001.1809).

- [72] M. J. Green and D. A. Cohen. Domain permutation reduction for constraint satisfaction problems. *Artificial Intelligence*, **172**(8–9), 1094–1118, 2008. doi:[10.1016/j.artint.2007.12.001](https://doi.org/10.1016/j.artint.2007.12.001).
- [73] M. Grohe. The structure of tractable constraint satisfaction problems. *MFCS 2006: Proceedings of the 31st Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **4162**, 58–72. Springer-Verlag, 2006. doi:[10.1007/11821069\\_5](https://doi.org/10.1007/11821069_5).
- [74] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, **54**(1), 1–24, 2007. doi:[10.1145/1206035.1206036](https://doi.org/10.1145/1206035.1206036).
- [75] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *SODA 2006: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, 289–298, 2006. doi:[10.1145/1109557.1109590](https://doi.org/10.1145/1109557.1109590).
- [76] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, **1**(2), 169–197, 1981. doi:[10.1007/BF02579273](https://doi.org/10.1007/BF02579273).
- [77] D. Gusfield. A graph theoretic approach to statistical data security. *SIAM Journal on Computing*, **17**(3), 552–571, 1988. doi:[10.1137/0217034](https://doi.org/10.1137/0217034).
- [78] G. Gutin, A. Rafiey, and A. Yeo. Minimum cost and list homomorphisms to semicomplete digraphs. *Discrete Applied Mathematics*, **154**(6), 890–897, 2006. doi:[10.1016/j.dam.2005.11.006](https://doi.org/10.1016/j.dam.2005.11.006).
- [79] R. Hayward and B. A. Reed. Forbidding holes and antiholes. In Ramírez Alfonsín and Reed [132], chapter 6.
- [80] R. B. Hayward, J. P. Spinrad, and R. Sritharan. Improved algorithms for weakly chordal graphs. *ACM Transactions on Algorithms*, **3**(2), 14:1–19, 2007. doi:[10.1145/1240233.1240237](https://doi.org/10.1145/1240233.1240237).
- [81] R. A. Hearn and E. D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, **343**(1–2), 72–96, 2005. doi:[10.1016/j.tcs.2005.05.008](https://doi.org/10.1016/j.tcs.2005.05.008).
- [82] P. Hell and J. Nešetřil. On the complexity of  $H$ -coloring. *Journal of Combinatorial Theory, Series B*, **48**, 92–110, 1990. doi:[10.1016/0095-8956\(90\)90132-J](https://doi.org/10.1016/0095-8956(90)90132-J).
- [83] P. Hell and J. Nešetřil. *Graphs and Homomorphisms*, volume 28 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, 2004.

- [84] C. T. Hoàng. Perfectly orderable graphs: A survey. In Ramírez Alfonsín and Reed [132], chapter 7, 139–166.
- [85] C. T. Hoàng, F. Maffray, and M. Mechebbek. A characterization of b-perfect graphs. *Journal of Graph Theory*, **71**(1), 95–122, September 2012. doi:10.1002/jgt.20635.
- [86] S. Hougardy. Classes of perfect graphs. *Discrete Mathematics*, **306**, 2529–2571, 2006. doi:10.1016/j.disc.2006.05.021.
- [87] C. Houghton, D. Cohen, and M. J. Green. The effect of constraint representation on structural tractability. In Benhamou [11], 726–730. doi:10.1007/11889205\_59.
- [88] D. J. Hulme. *The Path to Satisfaction: Polynomial Algorithms for SAT*. PhD thesis, Department of Computer Science, University College London, 2008.
- [89] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *Journal of the ACM*, **31**(4), 761–791, 1984. doi:10.1145/1634.1886.
- [90] R. Impagliazzo and R. Paturi. On the complexity of  $k$ -SAT. *Journal of Computer and System Sciences*, **62**, 367–375, 2001. doi:10.1006/jcss.2000.1727.
- [91] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, **63**(4), 512–530, 2001. doi:10.1006/jcss.2001.1774.
- [92] W. Imrich and S. Klavžar. *Product Graphs: Structure and Recognition*. Wiley, 2000.
- [93] A. Ion, J. Carreira, and C. Sminchisescu. Image segmentation by figure-ground composition into maximal cliques. *ICCV 2011: Proceedings of the 13th IEEE International Conference on Computer Vision*, 2110–2117, 2011. doi:10.1109/ICCV.2011.6126486.
- [94] R. Irving and M. Jerrum. Three-dimensional statistical data security problems. *SIAM Journal on Computing*, **23**(1), 170–184, 1994. doi:10.1137/S0097539790191010.
- [95] P. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, **200**(1–2), 185–204, 1998. doi:10.1016/S0304-3975(97)00230-2.
- [96] P. G. Jeavons and M. C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, **79**(2), 327–339, 1995. doi:10.1016/0004-3702(95)00107-7.
- [97] P. Jégou. [Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems](#). *AAAI 1993: Proceedings of the Eleventh National Conference on Artificial Intelligence*, 731–736, 1993.
- [98] W. W. Johnson and W. E. Story. [Notes on the “15” puzzle](#). *American Journal of Mathematics*, **2**(4), 397–404, 1879.



- [99] R. M. Karp. [Reducibility among combinatorial problems](#). In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Communications*, The IBM Research Symposia Series, 85–103. Plenum, 1972. doi:[10.1007/978-1-4684-2001-2\\_9](#).
- [100] A. B. Kempe. [A memoir on the theory of mathematical form](#). *Philosophical Transactions of the Royal Society of London*, **177**, 1–70, 1886.
- [101] S. C. Kleene. [Recursive predicates and quantifiers](#). *Transactions of the American Mathematical Society*, **53**(1), 41–73, 1943.
- [102] C. Knessl and J. Keller. Partition asymptotics from recursion equations. *SIAM Journal on Applied Mathematics*, **50**(2), 323–338, 1990. doi:[10.1137/0150020](#).
- [103] D. E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal on Discrete Mathematics*, **5**(3), 422–427, 1992. doi:[10.1137/0405033](#).
- [104] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, **61**(2), 302–332, 2000. doi:[10.1006/jcss.2000.1713](#).
- [105] R. E. Korf. Recent progress in the design and analysis of admissible heuristic functions. *SARA 2002: Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*, Lecture Notes in Computer Science **1864**, 45–55, London, UK, 2000. Springer-Verlag. doi:[10.1007/3-540-44914-0\\_3](#).
- [106] D. Kozen. A clique problem equivalent to graph isomorphism. *SIGACT News*, **10**(2), 50–52, 1978. doi:[10.1145/990524.990529](#).
- [107] O. Kullmann. [Constraint satisfaction problems in clausal form: Autarkies and minimal unsatisfiability](#). *Electronic Colloquium on Computational Complexity (ECCC)*, **14**(055), 2008. Revision 1.
- [108] G. Kun and J. Nešetřil. Forbidden lifts (NP and CSP for combinatorialists). *European Journal of Combinatorics*, **29**(4), 930–945, 2008. doi:[10.1016/j.ejc.2007.11.027](#).
- [109] C. Kuratowski. Sur la notion de l’ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, **2**(1), 161–171, 1921.
- [110] R. E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, **22**(1), 155–171, 1975. doi:[10.1145/321864.321877](#).
- [111] J. Lee, editor. *CP 2011: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **6876**. Springer-Verlag, 2011. doi:[10.1007/978-3-642-23786-7](#).



- [112] J. Lehel. Covers in hypergraphs. *Combinatorica*, **2**(3), 305–309, 1982. doi:[10.1007/BF02579237](https://doi.org/10.1007/BF02579237).
- [113] D. Lokshtanov, M. Vatshelle, and Y. Villanger. Independent set in  $P_5$ -free graphs in polynomial time. *SODA 2014: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2014. (accepted).
- [114] L. Lovász. Normal hypergraphs and the perfect graph conjecture. *Discrete Mathematics*, **2**, 253–267, 1972. doi:[10.1016/0012-365X\(72\)90006-4](https://doi.org/10.1016/0012-365X(72)90006-4).
- [115] F. Madelaine and I. A. Stewart. Constraint satisfaction, logic and forbidden patterns. *SIAM Journal on Computing*, **37**(1), 132–163, 2007. doi:[10.1137/050634840](https://doi.org/10.1137/050634840).
- [116] N. V. R. Mahadev and B. A. Reed. A note on vertex orders for stability number. *Journal of Graph Theory*, **30**(2), 113–120, 1999. doi:[10.1002/\(SICI\)1097-0118\(199902\)30:2<71::AID-JGT1>3.0.CO;2-G](https://doi.org/10.1002/(SICI)1097-0118(199902)30:2<71::AID-JGT1>3.0.CO;2-G).
- [117] D. Marx. Tractable structures for constraint satisfaction with truth tables. In S. Albers and J.-Y. Marion, editors, *STACS 2009: Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science*, 649–660, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. doi:[10.4230/LIPIcs.STACS.2009.1807](https://doi.org/10.4230/LIPIcs.STACS.2009.1807).
- [118] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *STOC 2010: Proceedings of the 42nd ACM symposium on Theory of computing*, 735–744. ACM, 2010. doi:[10.1145/1806689.1806790](https://doi.org/10.1145/1806689.1806790).
- [119] D. Marx. Tractable structures for constraint satisfaction with truth tables. *Theory of Computing Systems*, **48**(3), 444–464, 2011. doi:[10.1007/s00224-009-9248-9](https://doi.org/10.1007/s00224-009-9248-9).
- [120] M. Middendorf and F. Pfeiffer. On the complexity of recognizing perfectly orderable graphs. *Discrete Mathematics*, **80**(3), 327–333, 1990. doi:[10.1016/0012-365X\(90\)90251-C](https://doi.org/10.1016/0012-365X(90)90251-C).
- [121] D. G. Mitchell. *The resolution complexity of constraint satisfaction*. PhD thesis, University of Toronto, Toronto, Canada, 2002.
- [122] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, **7**, 95–132, 1974. Also Technical Report, Carnegie-Mellon University, 1971. doi:[10.1016/0020-0255\(74\)90008-5](https://doi.org/10.1016/0020-0255(74)90008-5).
- [123] N. Narodytska and T. Walsh. Breaking symmetry with different orderings. In C. Schulte, editor, *CP 2013: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **8124**, 545–561. Springer-Verlag, 2013. doi:[10.1007/978-3-642-40627-0\\_41](https://doi.org/10.1007/978-3-642-40627-0_41).

- [124] S. D. Nikolopoulos and L. Palios. [Hole and antihole detection in graphs](#). *SODA 2004: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 850–859, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.
- [125] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [126] M. Pătraşcu and R. Williams. On the possibility of faster SAT algorithms. *SODA 2010: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1065–1075, 2010. [doi:10.1137/1.9781611973075.86](#).
- [127] C. S. Peirce. The critic of arguments (1892). In C. Hartshorne and P. Weiss, editors, *Collected Papers of Charles Sanders Peirce*, volume III. Exact Logic (Published Papers), 250–265. Harvard University Press, 1933.
- [128] D. Peterson. Gridline graphs: a review in two dimensions and an extension to higher dimensions. *Discrete Applied Mathematics*, **126**, 223–239, 2003. [doi:10.1016/S0166-218X\(02\)00200-7](#).
- [129] J. Petke and P. Jeavons. Local consistency and SAT-solvers. *CP 2010: Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **6308**. Springer-Verlag, 2010. [doi:10.1007/978-3-642-15396-9\\_33](#).
- [130] E. L. Post. [Formal reductions of the general combinatorial decision problem](#). *American Journal of Mathematics*, **65**(2), 197–215, 1943.
- [131] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, **52**, 264–268, 1946. [doi:10.1090/S0002-9904-1946-08555-9](#).
- [132] J. L. Ramírez Alfonsín and B. A. Reed, editors. *Perfect Graphs*. John Wiley & Sons, 2001.
- [133] G. Ravindra and K. Parthasarathy. Perfect product graphs. *Discrete Mathematics*, **20**, 177–186, 1977. [doi:10.1016/0012-365X\(77\)90056-5](#).
- [134] J.-C. Régin. [A filtering algorithm for constraints of difference in CSPs](#). *AAAI 1994: Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, 362–367, 1994.
- [135] F. S. Roberts. T-colorings of graphs: recent results and open problems. *Discrete Mathematics*, **93**(2-3), 229–245, 1991. [doi:10.1016/0012-365X\(91\)90258-4](#).
- [136] N. Robertson, D. P. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. *STOC 1996: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 571–575, New York, NY, USA, 1996. ACM. [doi:10.1145/237814.238005](#).

- [137] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, **5**(2), 266–283, 1976. doi:[10.1137/0205021](https://doi.org/10.1137/0205021).
- [138] F. Rossi, editor. *CP 2003: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **2833**. Springer-Verlag, 2003. doi:[10.1007/b13743](https://doi.org/10.1007/b13743).
- [139] F. Rossi, C. Petrie, and V. Dhar. *On the Equivalence of Constraint Satisfaction Problems*. Technical Report ACT-AI-222-89, MCC, 1989.
- [140] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.
- [141] A. Z. Salamon and P. G. Jeavons. Perfect constraints are tractable. In P. J. Stuckey, editor, *CP 2008: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **5202**, 524–528. Springer-Verlag, 2008. doi:[10.1007/978-3-540-85958-1\\_35](https://doi.org/10.1007/978-3-540-85958-1_35).
- [142] T. J. Schaefer. The complexity of satisfiability problems. *STOC 1978: Proceedings of the tenth annual ACM symposium on Theory of computing*, 216–226, 1978. doi:[10.1145/800133.804350](https://doi.org/10.1145/800133.804350).
- [143] A. Schutt, T. Feydy, P. Stuckey, and M. Wallace. Explaining the cumulative propagator. *Constraints*, **16**(3), 250–282, 2011. doi:[10.1007/s10601-010-9103-2](https://doi.org/10.1007/s10601-010-9103-2).
- [144] M. Sellmann. The polytope of tree-structured binary constraint satisfaction problems. In L. Perron and M. A. Trick, editors, *CPAIOR 2008: Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science **5015**, 367–371. Springer-Verlag, 2008. doi:[10.1007/978-3-540-68155-7\\_39](https://doi.org/10.1007/978-3-540-68155-7_39).
- [145] W. E. Singletary. [The equivalence of some general combinatorial decision problems](#). *Bulletin of the American Mathematical Society*, **73**, 446–451, 1967.
- [146] R. M. Stallman and G. J. Sussman. *Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis*. Technical Report Memo No. 380, Massachusetts Institute of Technology Artificial Intelligence Laboratory, September 1976.
- [147] J. Thapper and S. Živný. The complexity of finite-valued CSPs. *STOC 2013: Proceedings of the 45th annual ACM symposium on Theory of computing*, 695–704. ACM, 2013. doi:[10.1145/2488608.2488697](https://doi.org/10.1145/2488608.2488697).
- [148] E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

- [149] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, **s2-42**(1), 230–265, 1937. [doi:10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230).
- [150] N. Ueda and T. Nagao. *NP-completeness Results for NONOGRAM via Parsimonious Reductions*. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, 1996.
- [151] P. van Beek and R. Dechter. On the minimality and decomposability of row-convex constraint networks. *Journal of the ACM*, **42**(3), 543–561, 1995. [doi:10.1145/210346.210347](https://doi.org/10.1145/210346.210347).
- [152] W.-J. van Hoeve and I. Katriel. Global constraints. In Rossi et al. [140], chapter 6, 169–208. [doi:10.1016/S1574-6526\(06\)80010-6](https://doi.org/10.1016/S1574-6526(06)80010-6).
- [153] V. I. Voloshin. *Coloring Mixed Hypergraphs: Theory, Algorithms and Applications*. Fields Institute Monographs **17**. American Mathematical Society, 2002.
- [154] T. Walsh. SAT v CSP. *CP 2000: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science **1894**, 441–456. Springer-Verlag, 2000. [doi:10.1007/3-540-45349-0\\_32](https://doi.org/10.1007/3-540-45349-0_32).
- [155] R. Weigel and C. Bliet. On reformulation of constraint satisfaction problems. *ECAI 1998: Proceedings of the 13th European Conference on Artificial Intelligence*, 254–258, 1998.

# Index of terms

- 1-consistent, [34](#)
- active domain, [26](#)
- admissible ordering, [109](#)
- antihole, [90](#)
- antisymmetric, [13](#)
- anything-goes
  - constraint, [20](#)
- applied to relation, [29](#)
- arc-consistent, [65](#)
- arity
  - of CSP instance, [29](#)
  - of relational structure, [14](#)
  - of tuple, [12](#)
- assigning literal, [52](#)
- assignment, [9](#)
- avoiding literal, [52](#)
- bag, [122](#)
- binary, [13](#)
  - CSP instance, [23](#)
- Boolean, [23](#)
- Boolean queries, [28](#)
- bounded
  - log-space, [12](#)
  - polynomial-time, [11](#)
- bounded arity, [76](#)
- broken-triangle property, [115](#)
- BTP, [115](#)
- canonical constraint relation, [115](#)
- Cartesian product, [13](#)
- CC, [96](#)
- CCMC, [96](#)
- chord, [96](#)
- chordal, [96](#)
  - co-chordal, [96](#)
  - weakly, [96](#)
- CHROMATIC NUMBER, [17](#)
- chromatic number, [17](#)
- class, [10](#)
  - of structures, [45](#)
- clause, [53](#)
- clause structure, [54](#)
  - variable-coloured, [104](#)
- claw, [91](#)
- CLIQUE, [17](#)
- clique, [17](#)
- clique-completion, [106](#)
- closed-world assumption, [54](#)
- co- $G$ , [16](#)
- co-chordal, [96](#)
- coarser, [14](#)
- colour class, [103](#)
- colouring
  - proper, [17](#)
- colours, [103](#)
- complement, [72](#)
  - graph, [16](#)
  - hypergraph, [15](#)
  - of relation, [18](#)
  - of relational structure, [18](#)
- complete  $(r, s)$ -structure, [121](#)
- complete assignment, [24](#)
- complete graph, [16](#)
- COMPLETE RAINBOW INDEPENDENT SET,  
[107](#)
- complete relation, [13](#)
- complete representation, [121](#)
- completely disconnected, [17](#)

- conformal, 78
- conjunctive queries, 28
- consistency
  - arc, 65
- consistent partial assignment, *see* prop
- constraint, 23
  - anything-goes, 20
  - global, 19
  - graph, 29
  - hypergraph, 29
  - network, *see* constraint hypergraph
  - relation, 23
    - canonical, 115
  - scope, 23
  - violation, 24
- constraint graph, 94
- contained, 110
- CRC, 107
- CRIS, 107
- CRIS-easy, 111
- CRIS-hard, 111
- $CS(\mathcal{C})$ , 88
- $CS(\mathcal{P})$ , 54
- CSP, 9
- CSP instance, 9
  - arity of, 29
  - binary, 23
- cycle, 16
- decision problem, 11
- decomposed, 76
- diamond, 91
- direct encoding, 68
- directed graph, 15
- DIRECTED GRAPH ACYCLICITY, 4
- disequality, 42
- domain, 23
  - reduction, 88
- domain clause, 68
- down-clique, 58
- downward-closed, 89
- dual graph, 84
- dual transformation, 84
- edge, 15
- edges
  - of hypergraph, 15
- equality, 42
- equivalence class, 14
- equivalence relation, 14
- equivalent
  - log-space, 12
- exact 2-section
  - of CSP instance, 77
- explicit
  - instance, 53
  - nogood, 25
- explicit version, 53
- extended, 24
- extensional, 32
- finer, 14
- finite, 14
- first-order logic representation, 27
- forgetting, 103
- fork, 91
- fractional hypertree width, 124
- free
  - w.r.t. class of structures, 85
  - w.r.t. sets, 85
  - w.r.t structures, 85
- FUTOSHIKI, 21
- GAC, 65
- Gaifman graph, 15
- generalized arc-consistent, 65
- giant hole, 90
- global constraint, 19, 76
- graph, 15
  - bipartite, 17
  - chromatic number, 17
  - complement, 16
  - completely disconnected, 17

- directed, 15
- independent set in, 17
- proper colouring, 17
- vertex-colouring, 17
- GRAPH  $t$ -COLOURING, 43, 112
- graph isomorphism, 16
- graph of, 15
- gridline, 97
- Helly property, 123
- hereditary, 89
- hereditary class, 111
- hidden transformation, 84
- hidden variable transformation, 84
- hole, 90
- homomorphism representation, 28
- house, 91
- hybrid, 46
- hyperedges
  - of hypergraph, 15
- hypergraph, 15
  - complement, 15
  - edges, 15
  - hyperedges, 15
  - intersection, 66
  - on a set, 15
  - union, 66
  - vertices, 15
- hypergraph of CSP instance, 29
- hypergraph of relational structure, 18
- image, 13
- incidence graph, 84
- independent set, 17, 58
- induced
  - subgraph, 16
  - subhypergraph, 15
- induced substructure, 16
- induces, 16
- infrastructure, 34
- infrastructure-equivalent to, 34
- INJECTIVE, 20
- instance, 9
  - explicit, 53
  - nogood, 25
  - of CSP, 9
  - prop, 25
- instance description
  - first-order logic, 27
  - homomorphism, 28
  - variable-value, 23
- intensional, 32
- intersection, 66
  - hypergraph, 66
- inverse, 13
- irreflexive, 13
- IS-easy, 87
- IS-hard, 87
- IS-reduction, 106
- isomorphic, 16
- $k$ -section, 15
- language
  - defined by class, 45
  - of structure, 14
- language class, 42
- language restriction, 42
- LATIN SQUARE COMPLETION, 21
- literal, 52
- log-space
  - bounded, 12
  - equivalent, 12
  - reduction, 12
- logarithmically bounded, 11
- many-one reduction, 12
- max-closed
  - CSP instance, 45
  - relation, 45
- microstructure, 52
  - variable-coloured, 104
- microstructure complement, 54

- microstructure representation, 54
- $MS(\mathcal{C})$ , 88
- $MS(\mathcal{P})$ , 52
- negative literal, 67
- network decomposable, 77
- nogood, 25
  - explicit, 25
  - implicit, 25
  - instance, 25
- NP, 12
- NP-complete, 12
- obstruction, 109
- open-world assumption, 55
- ordered pair, 12
- P, 12
- pair
  - ordered, 12
- partial assignment, 24
  - consistent, *see* prop
- partial hypergraph, 15
- partial order, 13
- path, 16
- paw, 91
- perfect, 17
- PLANAR GRAPH  $k$ -COLOURING, 2
- polynomial time, 11
- polynomial-time
  - bounded, 11
  - decidable, 11
  - reduction, 12
- polynomially bounded, 11
- positive literal, 67
- preorder, 13
- primal graph, 15
- primitive positive, 27
- problem, 11
  - log-space equivalent, 12
- product
  - Cartesian, 13
  - relational structures, 69
  - relations, 69
- projection, 14
- promise problem, 11
- prop, 25
  - instance, 25
- propagation, 111
- proper colouring, 17
- proper relation, 13
- RAINBOW INDEPENDENT SET, 107
- rainbow set, 107
- RC, 107
- reduction
  - log-space, 12
  - many-one, 12
  - polynomial-time, 12
- reduction, domain, *see* domain reduction
- reflexive, 13
- related, 13
- relates, 13
- relation, 13
  - complement of, 18
  - complete, 13
  - disequality, 42
  - equality, 42
  - proper, 13
  - right monotone, 119
- relational structure, 14
  - arity of, 14
  - complement of, 18
  - vertices, 14
- relational structure homomorphism, 29
- renamable right monotone, 119
- representation
  - complete, 121
  - first-order logic, 27
  - homomorphism, 28
  - variable-value, 23
- respect the constraints, 24
- restriction



- language, 42
- structural, 42
- right monotone, 119
- RIS, 107
- s*-clique, 17
- s*-cycle, 16
- s*-down-clique, 58
- s*-independent set, 58
- SAT clause, 67
- SAT instance, 67
- satisfying assignment, 28
- signature, 14
- solution, 9, 25
- solving, 12
- source structure, 28
- star, 91
- strict substructure, 88
- strict total order, 13
- strictly totally ordered set, 14
- structural class, 42
- structural restriction, 42
- structures
  - class of, 45
- subgraph, 16
  - induced, 16
- subhypergraph, 15
  - induced, 15
- subproblem, 34
- substructure, 16, 88
- SUDOKU, 21
- SUM, 22
- supported, 35
- symmetric, 13
- target structure, 28
- total order, 13
  - strict, 13
- tractable, 41
- transitive, 13
- tree, 86
- tree decomposition, 122
- tree structure, 94
  - weak, 118
- tree-width, 123
- triangulated, 96
- tuple
  - arity of, 12
  - increasing, 14
- unary, 13
- underlying set of a tuple, 18
- union, 66
  - hypergraph, 66
- values, 23
- values in the instance, 24, 29
- variable clause, 54
- variable-coloured clause structure, 104
- variable-coloured microstructure, 104
- variable-value representation, 23
- variables, 23
  - in a constraint, 24
  - in an instance, 24
  - of partial assignment, 24
- variables in the instance, 24, 29
- vc-graph, 103
- vc-hypergraphs, 103
- vc-structure, 103
- vc-subgraph, 110
  - induced, 110
- vertex-coloured graph, 103
- vertex-colouring, 17
- vertices
  - of hypergraph, 15
  - of relational structure, 14
- VICTORIAN LETTERS, 3
- violates a constraint, 24
- weak tree structure, 118
- weakly chordal, 96
- width, 123