Some Results for FO-definable Constraint Satisfaction Problems Described by Digraph Homomorphisms

Patrick Aaron Moore

A Thesis
in
The Department
of
Mathematics and Statistics

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Science (Mathematics) at
Concordia University
Montréal, Québec, Canada

May 2018

© Patrick Aaron Moore, 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Patrick Aaron Moore**

Entitled: **Some Results for FO-definable Constraint Satisfaction Problems Described by Digraph Homomorphisms**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Mathematics)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Galia Dafni

______________________________ Examiner
Dr. Nadia Hardy

______________________________ Supervisor
Dr. Benoit Larose

______________________________ Co-supervisor
Dr. Chantal David

Approved by ______________________________
Chair of the Department of Mathematics and Statistics

__________ 2018 ______________________________
Dean of the Faculty of Arts and Science

ABSTRACT

Some Results for FO-definable Constraint Satisfaction Problems Described by Digraph Homomorphisms

Patrick Aaron Moore

Constraint satisfaction problems, or CSPs, are a naturally occurring class of problems which involve assigning values to variables while respecting a set of constraints. When studying the computational and descriptive complexity of such problems it is convenient to use the equivalent formulation, introduced by Feder and Vardi, that CSPs are homomorphism problems. In this context we ask if there exists a homomorphism to some target structure. Using this view many tools and ideas have been introduced in combinatorics, logic and algebra for studying the complexity of CSPs. In this thesis we concentrate on combinatorics and give characterization results based on digraph properties. Where previous studies focused on CSPs defined by a single digraph with lists we extend our relational structures to consist of many binary relations which each individually describe a distinct digraph on the structures universe. A majority of our results are obtained by using an algorithm introduced by Larose, Loten and Tardif which determines whether a structure defines a CSP whose homomorphism problem can be represented by first order logic. Using this tool we begin by completely classifying which of these structures are FO-definable when each of the relations defines a transitive tournament. We then generalize a characterization theorem, first given by Lemaître, to include structures containing any finite number of digraph relations and lists. We conclude with examples of obstructions and properties that can determine if a particular relational structure has a CSP which is FO-definable and how to construct such structures.

Acknowledgments

I would like to thank my supervisor Dr. Benoit Larose for his guidance, mentoring, patience and unwavering enthusiasm for these subjects without which this work would never have come into being.

Contents

List of Figures **vii**

1 Introduction **1**

2 Relational Structures **3**

2.1 Relational structures and their properties 3

2.2 Homomorphisms, retracts and cores 7

3 Digraphs **10**

3.1 Properties of digraphs 10

4 Constraint Satisfaction Problems **16**

4.1 The formal approach to CSPs 16

4.2 CSPs as homomorphism problems 18

5 Descriptive and Computational Complexity **21**

5.1 Efficient computation, efficient verification and the complexity classes P and NP . 22

5.2 L 26

5.3 NL 26

5.4 FO 27

6 Characterizations of Relational Structures That Define CSPs Which Are FO-definable With Lists **29**

6.1 The dismantling algorithm 29

6.2 A classification of digraphs with lists whose CSPs are FO-definable 32

6.3 CSPs that are FO-definable when the relations of $\mathbf{G}_L$ are transitive tournaments . . 38

6.4 A classification of relational structures with lists and n binary relations whose CSPs are FO-definable 50

6.5 Observations and examples 56

7 Conclusion **60**

7.1 Contribution 60

7.2 Outlook 61

Appendix A Dismantling Software **62**

A.1 Dismantle.java 62

A.2 Graph.java 70

A.3 RStruct.java 87

A.4 RStruct3.java 91

Bibliography **95**

List of Figures

Figure 2.1 $\mathbf{G}$ and its induced subdigraph $\mathbf{H}$ . . . 5
Figure 2.2 The tournament $\mathbf{T}_3$ is a retract of $\mathbf{G}$ . . . 8
Figure 3.1 Symmetric digraph . . . 11
Figure 3.2 We denote the complete graph on n vertex as $\mathbf{K}_n$. . . . 11
Figure 3.3 A Tournament . . . 11
Figure 3.4 A Transitive Tournament . . . 12
Figure 3.5 A reflexive digraph . . . 13
Figure 3.6 A bipartite graph . . . 13
Figure 3.7 A directed cycle (a, b, c). . . . 14
Figure 3.8 A connected graph . . . 14
Figure 3.9 A strongly connected digraph . . . 14
Figure 4.1 3-Colouring of a digraph with 6 vertices where no edge is shared between two vertices of the same colour. . . . 17
Figure 6.1 Digraph with separated edges (a, b) and (c, d). (The red dotted edge indicates that this edge is not present) . . . 33
Figure 6.2 An impeding bicycle . . . 34
Figure 6.3 Configuration from Proposition 10. . . . 35
Figure 6.4 Case (1) from Proposition 10. . . . 35
Figure 6.5 Case (2) from Proposition 10. . . . 36
Figure 6.6 The standard ordering of the transitive tournament on 4 vertices. . . . 43
Figure 6.7 The 4 tournament orderings that do not dismantle when paired with the standard ordering . . . 43
Figure 6.8 The square of the structure $\mathbf{T} = \langle \{0, 1, 2\}; E_1, E_2, E_3 \rangle$ does not dismantle to the diagonal . . . 44
Figure 6.9 $\mathbf{G}_F$ is the flip of $\mathbf{G}$. . . . 52
Figure 6.10 The generalized impeding bicycle . . . 52

Figure 6.11 Generalized impeding bicycle formed by separated edges in a flip closed structure. 53
Figure 6.12 The impeding bicycle found in the flip closed structure $\mathbf{T}_F$ from Example 6.5.2. 58
Figure 6.13 An obstruction found in the structure $\mathbf{T}$ from Example 6.5.2. 58

Chapter 1

Introduction

A Constraint Satisfaction Problem, abbreviated to CSP, consists of determining, given a finite set of variables with constraints, whether there exists a suitable assignment of values to these variables that satisfies all of the given constraints. The generality of this schema makes it an effective framework for many types of problems and this has made it a popular area of research for various branches of mathematics and computer science. In the latter it is natural to investigate such problems in terms of computational and descriptive complexity. In general the constraint satisfaction problem is known to be NP-*complete*. However, when certain restrictions on the form of the allowed constraints are added then one can find problems solvable in polynomial time. The well known dichotomy conjecture of Feder and Vardi states that any CSP is either in P or NP-*complete* [1]. Attacks on this conjecture have come from a combination of universal algebra, combinatorics and logic. In the process many tools and and ideas have been formulated in order to study the complexity of CSPs. In this text we will focus primarily on tractable CSPs using the combinatorial approach.

In [2] Feder and Vardi also offered the powerful observation that every CSP can be seen as a fixed target homomorphism problem which asks if there exists a homomorphism to a fixed relational structure. This is the standard view that we will take throughout this work. From this point of view it is also possible to discuss CSPs in terms of their descriptive complexity where we use sentences from different logics to describe a problem.

The dichotomy conjecture has been resolved for many types of CSPs, such as Boolean CSPs or those defined by graph homomorphisms, whose complexity has been completely classified. For example when it comes to (undirected) graph CSPs it was shown by Hell and Nešetril [3] that the

dichotomy can be characterized rather simply. If we have a graph $\mathbf{G}$, then the CSP defined by this graph is tractable if $\mathbf{G}$ is bipartite or if $\mathbf{G}$ has a loop and is in NP-*complete* otherwise. Such clean and concise results motivate researchers to find similar classifications for CSPs whose constraints are defined by various relational structures. Many of these problems still remain open. Feder and Vardi also showed the equivalence of resolving the dichotomy for digraphs and resolving the general dichotomy [1]. Bulatov proved this dichotomy in [4] for a particular type of relational structure which contains among its relations all possible unary relations on its universe. When these relations are present we say that this is a structure with lists. Bulatov's characterization of these structures with lists is purely algebraic and not graph theoretic. In this thesis we take a combinatorial and graph theoretic approach to classifying constraint satisfaction problems that are described by digraphs with lists as our template structures.

In Chapters 2, 3 and 4 we provide relevant results and properties for relational structures, digraphs and constraint satisfaction problems respectively. In chapter 5 we present a brief overview of computational and descriptive complexity where we define the idea of a first order definable problem. In Chapter 6 we present a dismantling algorithm, which was developed by Larose, Loten and Tardif in [5], for determining whether or not a particular CSP is FO-definable. Using the notion of relational domination introduced in this algorithm we present a full characterization, which was first established by Lemaître in [6], of which digraphs with lists have CSPs which are FO-definable. We then shift our attention to multi-relational binary structures where the set of relations contain more than one binary relation. An obvious first question is "If a structure has a set of binary relations which define digraphs that have CSPs that are FO-definable, then is the larger structure also FO-definable?" Starting with the fact that all transitive tournament digraphs with lists have CSPs which are FO-definable we construct larger structures with sets of transitive tournaments as the relations and show that this this larger structure may not have a CSP which is FO-definable. This implies that there are properties between these relations that make the overall CSP harder. Lastly, we discover these properties and generalize the characterization result of Lemaître in [6] for those structures with lists and many binary relations and provide some interesting examples from the culmination of our results.

Chapter 2

Relational Structures

We begin our investigation by defining some of our main objects of study. These include relational structures, digraphs, digraphs with lists and cores. We also define and describe some of the different types of mappings between these structures. The main result in this chapter is the proof that every relational structure has a unique core, up to isomorphism. This core is an interesting and important substructure because it captures the structure of the overall object in a homomorphism context. In later chapters we will show that we can define strong complexity results for problems described by cores. We in no way believe that this is a comprehensive or complete introduction to these objects and for that we would encourage the reader towards [7] or [8].

2.1 Relational structures and their properties

We begin by defining the components necessary to construct a relational structure.

Definition 2.1.1. Let A be a set. We define A^n as the set of all n-tuples $(x_0, \ldots, x_{n-1})$ where each $x_i \in A$.

Definition 2.1.2. An ***n-ary relation*** R on a set A is a subset of A^n.

When $n = 1$ we call R a unary relation, when $n = 2$ we call R a binary relation and so on.

Definition 2.1.3. Let σ be a finite set of n ***relation symbols*** $\{R_1^{r_1}, ..., R_n^{r_n}\}$ where each $r_i \in \mathbb{N}^+$. We say that each R_i has *arity* r_i and we call the set σ a ***vocabulary***.

Combining these we can define our main object of study, the relational structure.

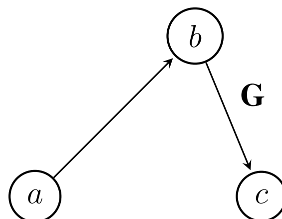
Definition 2.1.4. Let $\sigma = \{R_1^{r_1}, ..., R_n^{r_n}\}$ be a vocabulary and A a non-empty set. We call $\mathbf{A} = \langle A; R_1^{\mathbf{A}}, R_2^{\mathbf{A}} \cdots, R_n^{\mathbf{A}} \rangle$ a ***relational structure*** with ***universe*** A and vocabulary σ or, simply, a σ-***structure***, if $R_i^{\mathbf{A}} \subseteq A^{r_i}$ for each i.

We call two relational structures with the same vocabulary ***similar***.

Definition 2.1.5. A ***digraph***, or directed graph, is a relational structure $\mathbf{G} = \langle V; E \rangle$. The elements of its universe, V, are called ***vertices*** and it has one binary relation symbol, E, which defines its edges. We denote the set of vertices of $\mathbf{G}$ as $V(\mathbf{G})$ and the set of edges as $E(\mathbf{G})$. We call an element of E an ***edge***. For any edge $(x, y) \in E(\mathbf{G})$ we say that x is an ***in-neighbour*** of y and that y is an ***out-neighbour*** of x.

Throughout this study we will follow convention and denote our graphs and digraphs as $\mathbf{G} = (V, E)$ but we will use relational structure notation, $\mathbf{G} = \langle V; E \rangle$, when E denotes a list of relations.

Example 2.1.1. Let $\mathbf{G} = (\{a, b, c\}, \{(a, b), (b, c)\})$. Then the universe of $\mathbf{G}$ is $V(\mathbf{G}) = \{a, b, c\}$ and it has a single binary relation defined by $E(\mathbf{G}) = \{(a, b), (b, c)\}$.



The digraph $\mathbf{G}$ is a basic relational structure with a single binary relation.

Although many of the results and ideas that will be discussed in this work can be applied for all types of relational structures our primary focus will be on those structures that have only binary relations. Which means each relation will define a digraph on the universe of the structure. However it is also important to our study to define an extension of the digraph known as a digraph with lists.

Definition 2.1.6. Let $\mathbf{G}$ be a digraph whose universe is the set of vertices V. Let $U_H = \{U_1, \ldots, U_k\}$ denote the set of all non-empty subsets of V. Let σ be the vocabulary that consists of one binary symbol E and the unary symbols R_i, for $1 \leq i \leq k$. We call the σ-structure, $\mathbf{G}_L$, with the binary edge relation $E(\mathbf{G}_L)$ and unary relations $R_i(\mathbf{G}_L) = U_i$, a ***digraph with lists***.

If we have some digraph $\mathbf{G}$ we are able to build $\mathbf{G}_L$ by adding to it all of its unary relations. We now give an illustrative example.

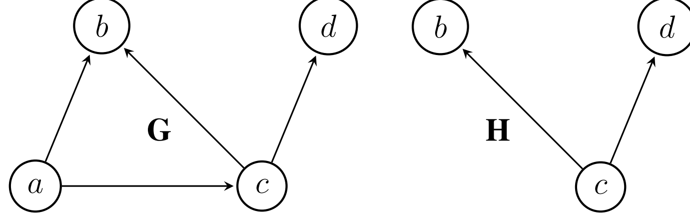


Figure 2.1: **G** and its induced subdigraph **H**

Example 2.1.2. Let $\mathbf{G}$ be the digraph from Example 2.1.1, where $\mathbf{G} = (\{a, b, c\}, \{(a, b), (b, c)\})$. Then $\mathbf{G}_L = (\{a, b, c\}, \{(a, b), (b, c)\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\})$ is $\mathbf{G}$ with lists.

We can similarly build a ***relational structure with lists*** from any relational structure by adding these unary relations. We can think of these unary relations as sets of vertex colourings. Each of these unary relations is a list that can be assigned to any vertex and their inclusion results in a more general structure which offers some useful properties to our study.

Definition 2.1.7. Let $\mathbf{G}$ be a digraph. We call the digraph $\mathbf{H}$ a ***subdigraph*** of $\mathbf{G}$ if $V(\mathbf{H}) \subseteq V(\mathbf{G})$ and $E(\mathbf{H}) \subseteq E(\mathbf{G})$ and we write $\mathbf{H} \subseteq \mathbf{G}$. We call $\mathbf{H}$ an ***induced subdigraph*** of $\mathbf{G}$ if $E(\mathbf{H}) = E(\mathbf{G}) \cap V(\mathbf{H}^2)$.

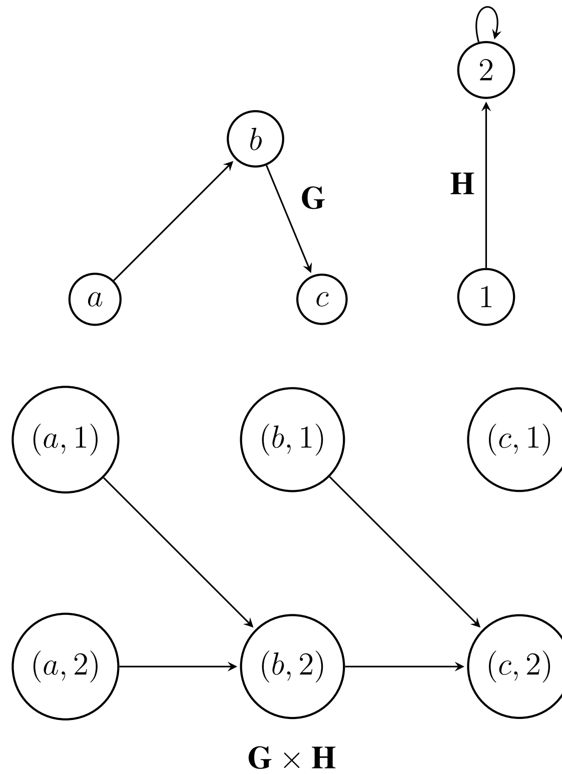
Example 2.1.3. Let $\mathbf{G} = (V, E)$ be a digraph with $V(\mathbf{G}) = \{a, b, c, d\}$ and $E(\mathbf{G}) = \{(a, b), (a, c), (c, b), (c, d)\}$. Then we can induce a substructure $\mathbf{H}$ by removing the vertex a of $\mathbf{G}$. Thus we have $V(\mathbf{H}) = \{b, c, d\}$ and $E(\mathbf{H}) = \{(c, b), (c, d)\}$. (Figure 2.1)

Like many other mathematical objects we are able to define some operations on relational structures. The most important of these operations for our study is that of the product. Which we define now.

Definition 2.1.8. Let $\mathbf{A}$ and $\mathbf{B}$ be similar relational structures such that $\mathbf{A} = \langle A; R_1^{\mathbf{A}}, \ldots, R_n^{\mathbf{A}} \rangle$ and $\mathbf{B} = \langle B, R_1^{\mathbf{B}}, \ldots, R_n^{\mathbf{B}} \rangle$ then we can define the ***product*** $\mathbf{A} \times \mathbf{B} = \langle A \times B; R_1^{\mathbf{A}\times\mathbf{B}}, \ldots, R_n^{\mathbf{A}\times\mathbf{B}} \rangle$. The elements in the universe of the product are the pairs (a_i, b_i) for all $a_i \in A$ and $b_i \in B$. We define the relations of the product as $R_i^{\mathbf{A}\times\mathbf{B}} = \{((a_1, b_1), \ldots, (a_n, b_n)) : (a_1, \ldots, a_n) \in R_i^{\mathbf{A}}, (b_1, \ldots, b_n) \in R_i^{\mathbf{B}}\}$.

Example 2.1.4. Let $\mathbf{G} = (\{a, b, c\}, \{(a, b), (b, c)\})$ and $\mathbf{H} = (\{1, 2\}, \{(1, 2), (2, 2)\})$ then the product of $\mathbf{G}$ and $\mathbf{H}$ can be described as:

$$V(\mathbf{G} \times \mathbf{H}) = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$$
$$E(\mathbf{G} \times \mathbf{H}) = \{((a, 1), (b, 2)), ((a, 2), (b, 2)), ((b, 1), (c, 2)), ((b, 2), (c, 2))\}$$

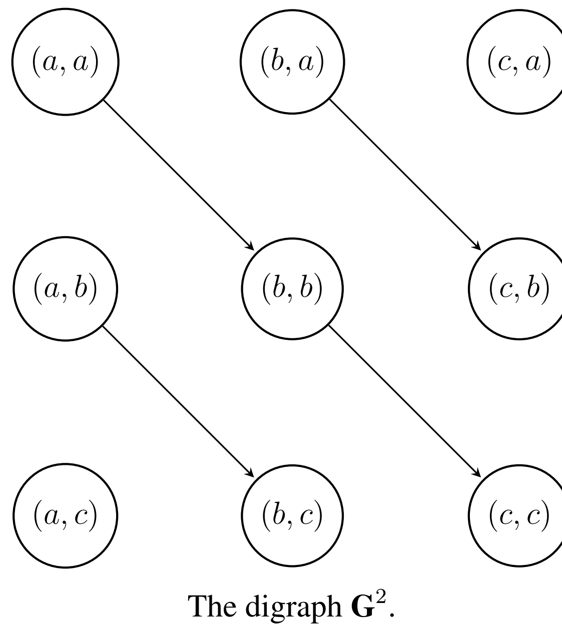


$\mathbf{G} \times \mathbf{H}$

We can also define the ***square*** of a relational structure as $\mathbf{A}^2 = \mathbf{A} \times \mathbf{A}$. Similarly we define the ***nth*** power of **A**, $\mathbf{A}^n$, as the product of **A** with itself n times.

Example 2.1.5. Let G be the digraph from Example 2.1.4. Then:

$$V(\mathbf{G}^2) = \{\{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c),(c,a),(c,b),(c,c)\}$$
$$E(\mathbf{G}^2) = \{((a,a),(b,b)),((a,b),(b,c)),((b,a),(c,b)),((b,b),(c,c))\}\}$$



The digraph $\mathbf{G}^2$.

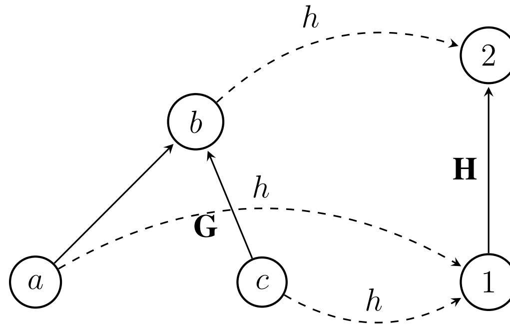
For the square of any digraph $\mathbf{G}$ we have that $|V(\mathbf{G}^2)| = |V(\mathbf{G})|^2$ and $|E(\mathbf{G}^2)| = |E(\mathbf{G})|^2$. Where $|V(\mathbf{G})|$ and $|E(\mathbf{G})|$ are the number of vertices and edges of $\mathbf{G}$ respectively. We call the substructure of $\mathbf{G}^2$ induced by the set of vertices $\{(x,x)|x \in V(\mathbf{G})\}$ the **diagonal** of $\mathbf{G}^2$.

2.2 Homomorphisms, retracts and cores

Another important concept for our study is that of homomorphisms between structures. Homomorphisms are simply relation-preserving mappings. In the case of digraphs we can think of these as edge-preserving functions.

Definition 2.2.1. A ***homomorphism*** from a relational structure $\mathbf{A}$ to a similar relational structure $\mathbf{B}$ is a mapping $h : \mathbf{A} \to \mathbf{B}$ such that for each n-ary relation $R_i^{\mathbf{A}}$ and every $(a_i, ..., a_n) \in R_i^{\mathbf{A}}$ we have $(h(a_i), ..., h(a_n)) \in R_i^{\mathbf{B}}$.

Example 2.2.1. Let $\mathbf{G}$ and $\mathbf{H}$ be a digraphs with $V(\mathbf{G}) = \{a, b, c\}$ and $E(\mathbf{G}) = \{(a, b), (c, b)\}$ and $V(\mathbf{H}) = \{1, 2\}$ and $E(\mathbf{H}) = \{(1, 2)\}$. Then $h : \mathbf{G} \to \mathbf{H}$ defined by $h(a) = h(c) = 1$ and $h(b) = 2$ is a homomorphism. We see that that both edges in $\mathbf{G}$ map to $(1, 2) \in E(\mathbf{H})$. $(h(a), h(b)) = (h(c), h(b)) = (1, 2) \in E(\mathbf{H})$.



A homomorphism h from $\mathbf{G}$ to $\mathbf{H}$.

If there exists a homomorphism from $\mathbf{G} \to \mathbf{H}$ and a homomorphism from $\mathbf{H} \to \mathbf{G}$ then we call $\mathbf{G}$ and $\mathbf{H}$ ***homomorphically equivalent***.

Definition 2.2.2. An ***endomorphism*** of a structure $\mathbf{G}$ is a homomorphism $h : \mathbf{G} \to \mathbf{G}$.

Definition 2.2.3. A homomorphism from $h : \mathbf{G} \to \mathbf{H}$ is called an ***isomorphism*** if h is a bijection and h^{-1} is a homomorphism from $\mathbf{H} \to \mathbf{G}$. If there exists an isomorphism between $\mathbf{G}$ and $\mathbf{H}$ then we say that $\mathbf{G}$ and $\mathbf{H}$ are ***isomorphic*** and we write $\mathbf{G} \cong \mathbf{H}$. An isomorphism between $\mathbf{G}$ and itself is called an ***automorphism***.

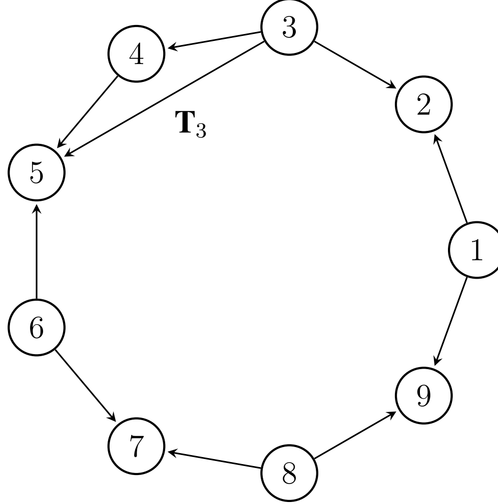


Figure 2.2: The tournament $\mathbf{T}_3$ is a retract of $\mathbf{G}$

Definition 2.2.4. We call a homomorphism $h : \mathbf{G} \to \mathbf{H}$ a ***retraction*** if $\mathbf{H} \subseteq \mathbf{G}$ and $h(x) = x$ for all $x \in \mathbf{H}$. We call a structure $\mathbf{H}$ a ***retract*** of $\mathbf{G}$ if there exists a retraction from $\mathbf{G}$ to $\mathbf{H}$ and we call this a ***proper retract*** if $\mathbf{H}$ is a proper substructure of $\mathbf{G}$.

Example 2.2.2. In Figure 2.2 the digraph $\mathbf{G}$ has $\mathbf{T}_3 = (\{3,4,5\}, \{(3,4),(3,5),(4,5)\})$ as a proper retract. We can construct the homomorphism $h : \mathbf{G} \to \mathbf{T}_3$ such that:

$$h(1) = h(3) = h(6) = h(8) = 3,$$
$$h(2) = h(5) = h(7) = h(9) = 5,$$
$$h(4) = 4$$

Since h fixes the image of $\mathbf{T}_3$ and $\mathbf{T}_3 \subset \mathbf{G}$ then $\mathbf{T}_3$ is a proper retract of $\mathbf{G}$.

Definition 2.2.5. We call a structure $\mathbf{C}$ a ***core*** if it has no proper retracts. We say that a structure $\mathbf{C}$ is a core of a structure $\mathbf{G}$ if $\mathbf{C}$ is a retract of $\mathbf{G}$ and $\mathbf{C}$ is a core.

A core, as its name suggests, is central in describing an overall structure. From the homomorphism point of view the core provides us with the DNA of a structure since all of its unique homomorphic information is contained in its cores. In Example 2.2 it should be obvious that $\mathbf{T}_3$ is a core of the digraph $\mathbf{G}$ because $\mathbf{T}_3$ does not have a proper retract. It should also be easy to see that all structures with lists are cores. This will be an important fact when we begin discussing the complexity of constraint satisfaction problems described by digraphs with lists.

Lemma 1. *Let* $\mathbf{G}$ *and* $\mathbf{H}$ *be two similar structures. If there exists surjective homomorphisms* $f : \mathbf{G} \to \mathbf{H}$ *and* $h : \mathbf{H} \to \mathbf{G}$ *then* $\mathbf{G}$ *and* $\mathbf{H}$ *are isomorphic.*

Proof. Since both f and h exist and are surjective we know that $|\mathbf{G}| = |\mathbf{H}|$. Thus both f and h are bijections and therefore $h \circ f$ is also a bijection. Therefore there exists a positive integer k such that $(h \circ f)^k$ is the identity mapping. Then we see that $(h \circ f)^{k-1} \circ h = f^{-1}$. Since f^{-1} is clearly a homomorphism we have that f is an isomorphism. □

Lemma 2. *Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be two cores. If $\mathbf{C}_1$ and $\mathbf{C}_2$ are homomorphically equivalent then $\mathbf{C}_1$ and $\mathbf{C}_2$ are isomorphic.*

Proof. Let $f : \mathbf{C}_1 \to \mathbf{C}_2$ and $h : \mathbf{C}_2 \to \mathbf{C}_1$ be homomorphisms. Then we have that $h \circ f$ is an endomorphism of $\mathbf{C}_2$ and since $\mathbf{C}_2$ is a core then this is an automorphism. Therefore both f and h are surjective and by Lemma 1 we have that $\mathbf{C}_1$ and $\mathbf{C}_2$ are isomorphic. □

Proposition 3. *Every relational structure $\mathbf{G}$ has a unique core $\mathbf{C}$, up to isomorphism, and $\mathbf{C}$ is a unique core to which $\mathbf{G}$ is homomorphically equivalent.*

Proof. First we show the existence of a core $\mathbf{C}$ of $\mathbf{G}$. Let $\mathbf{C}$ be a retract of $\mathbf{G}$ with minimal cardinality. Any proper retract of $\mathbf{C}$ would also be a retract of $\mathbf{G}$ and therefore $\mathbf{C}$ is a core.

Next, let $\mathbf{C}_1$ and $\mathbf{C}_2$ be two distinct cores of $\mathbf{G}$. Then by the definition of a core we have that there exist homomorphisms $h_1 : \mathbf{G} \to \mathbf{C}_1$ and $h_2 : \mathbf{G} \to \mathbf{C}_2$. Let r_1 define the restriction of h_1 to $\mathbf{C}_2$ and r_2 be the restriction of h_2 to $\mathbf{C}_1$. Then r_1 and r_2 show that the cores $\mathbf{C}_1$ and $\mathbf{C}_2$ are homomorphically equivalent and by Lemma 2 they are isomorphic. □

As we have stated throughout our investigation we will only be concerned with particular relational structures such as digraphs, digraphs with lists or structures with lists that only have binary relations (other than the set of allof its unary relations). Our study will be centred primarily on a combinatorial and graph theoretic approach, thus, for the sake of convenience, we will always denote the universe of all of our structures as V and the set of binary relations as E to emphasize this digraph connection. In the next chapter we present some properties of digraphs that will be useful to us throughout the remaining chapters of this exposition.

Chapter 3

Digraphs

In this chapter we will define many properties of the digraph. We will see in Chapter 4 that we can define a constraint satisfaction problem for every relational structure. It turns out that even though digraphs are rather simple structures they are sufficient for understanding these types of constraint satisfaction problems in general. Hence digraphs are an obvious choice for our investigation. The deep knowledge base pertaining to digraphs is extremely useful when attempting to understand their CSPs. For futher reference and more exhaustive studies of digraphs we suggest [9] or [7] to the reader.

3.1 Properties of digraphs

Definition 3.1.1. Let $\mathbf{G} = (V, E)$ be a digraph. If $(u, v) \in E$ implies that $(v, u) \in E$ then we call $\mathbf{G}$ a ***symmetric digraph***. A digraph which is symmetric is often simply called a ***graph***. (Figure 3.1).

Definition 3.1.2. Let $\mathbf{G} = (V, E)$ be a digraph such that for any $u, v \in V$ we have that $(u, v), (v, u) \in E$ and $u \neq v$ then we call $\mathbf{G}$ a ***complete graph***. (Figure 3.2).

Definition 3.1.3. We call a digraph $\mathbf{G} = (V, E)$ a ***tournament*** if for every $a, b \in V$, with $a \neq b$, either $(a, b) \in E$ or $(b, a) \in E$ but not both.

Definition 3.1.4. Let $\mathbf{G} = (V, E)$ be a tournament such that for any $a, b, c \in V$ where $(a, b), (b, c) \in E$ then $(a, c) \in E$. We call $\mathbf{G}$ a ***transitive tournament***. (Figure 3.4)

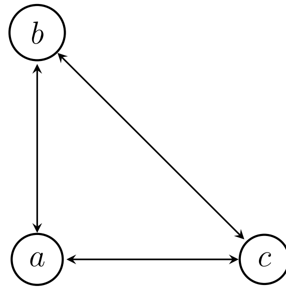


Figure 3.1: Symmetric digraph

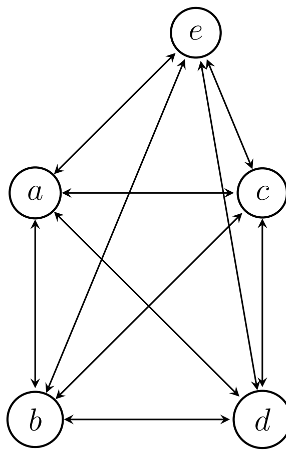


Figure 3.2: We denote the complete graph on n vertex as $\mathbf{K}_n$.

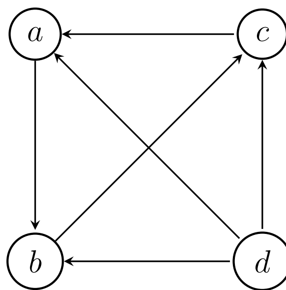


Figure 3.3: A Tournament

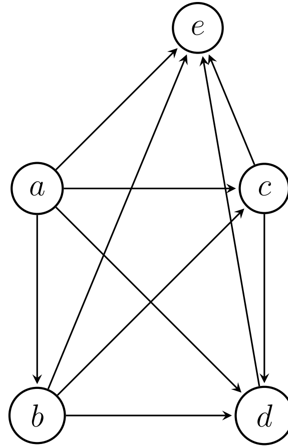


Figure 3.4: A Transitive Tournament

You can also think of a transitive tournament as a ***total strict ordering*** on the vertices of **G**. In Figure 3.4 the edges follow the alphanumeric ordering of the vertex names, $[a, b, c, d, e]$. Notice that all edges go out of a and all edges come into e and no two vertices have the same number of in and out edges. There are special names for the vertex a and the vertex e and we define those next.

Definition 3.1.5. We call the vertex $a \in V(\mathbf{G})$ a ***source*** of the digraph **G** if for all $b \in V(\mathbf{G})$ we have that $(b, a) \notin E(\mathbf{G})$.

Definition 3.1.6. We call the vertex $a \in V(\mathbf{G})$ a ***sink*** of the digraph **G** if for all $b \in V(\mathbf{G})$ we have that $(a, b) \notin E(\mathbf{G})$.

In every transitive tournament there is exactly one sink and one source. In Chapter 6 we will show how the very regular structure of transitive tournaments and the unique properties of their sink and their source offer us some non-trivial observations about the complexity of constraint satisfaction problems defined by these digraphs.

Definition 3.1.7. Let $\mathbf{G} = (V, E)$ be a digraph if for all $u \in V$ we have $(u, u) \in E$ we call **G** a ***reflexive digraph***. We call the edge (u, u) a ***loop***. (Figure 3.5)

A digraph that does not contain any loops is called ***irreflexive***.

Definition 3.1.8. Let $\mathbf{G} = (V, E)$ be a digraph if the set V can be partitioned into two non-empty sets U and W such that for any $(x, y) \in E$ either $x \in U$ and $y \in W$ or $x \in W$ and $y \in U$ we call **G** a ***bipartite digraph***. (Figure 3.6)

Definition 3.1.9. A ***walk*** is a sequence of vertices where each two consecutive vertices in the sequence have an edge between them. In a digraph the direction of the edge and the order of the

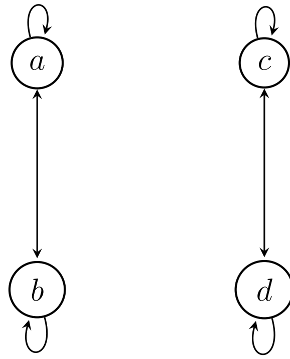


Figure 3.5: A reflexive digraph

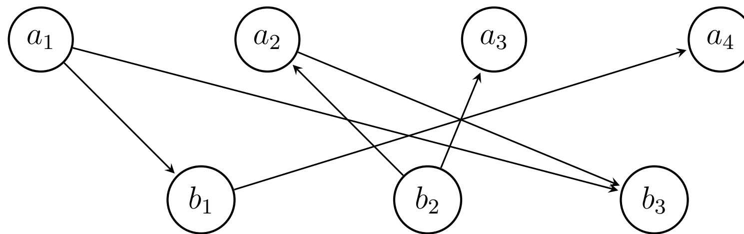


Figure 3.6: A bipartite graph

sequence must coincide. If the sequence starts and ends with the same vertex we call it a ***closed walk***.

Notice that the choice of starting vertex in a closed walk is irrelevant.

Definition 3.1.10. A ***path*** is a walk such that no vertex is repeated.

Definition 3.1.11. A ***directed cycle*** is a closed walk on a digraph such that no vertex is repeated in the sequence. (Figure 3.7)

Definition 3.1.12. Let $\mathbf{G} = (V, E)$ be a graph such that there is path between every pair of vertices. Then we call **G** a ***connected graph***. (Figure 3.8)

There is a different notion of connectedness when one is working with digraphs.

Definition 3.1.13. Let $\mathbf{G} = (V, E)$ be a digraph such that for any $u, v \in V$ there exists a walk from u to v. Then we call **G** a ***strongly connected digraph***. (Figure 3.9)

Definition 3.1.14. The ***graph neighborhood*** of a vertex v in an undirected graph **G** is the set of all vertices u such that $(v, u) \in E(\mathbf{G})$. We denote this set by $N(v)$.

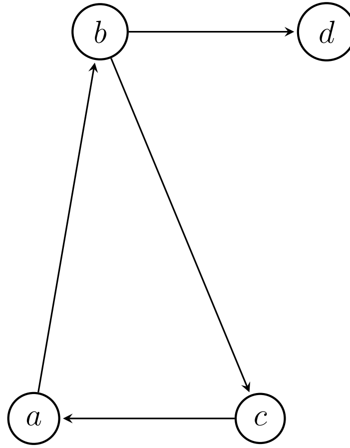


Figure 3.7: A directed cycle (a, b, c).

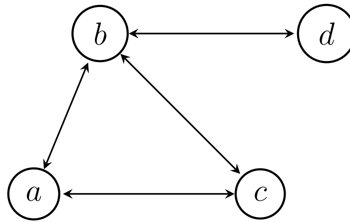


Figure 3.8: A connected graph

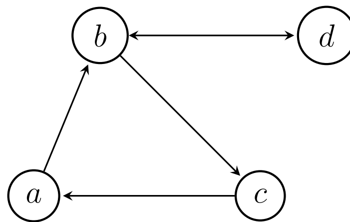


Figure 3.9: A strongly connected digraph

In a digraph we refine the notion of neighbourhoods to take into account the orientation of the edges.

Definition 3.1.15. Let $\mathbf{G} = (V, E)$ be a digraph with $v \in V$ we define the set $N_{out}(v)$ as the set of all vertex $a \in V$ such that $(v, a) \in E$ and the set $N_{in}(v)$ as the set of all vertex $a \in V$ such that $(a, v) \in E$. We call the set $N_{in}(x)$ the ***in-neighbourhood*** of x and the set $N_{out}(x)$ the ***out-neighbourhood*** of x. We say that the ***out-degree*** of x is $|N_{out}(x)|$ and the ***in-degree*** of x is $|N_{in}(x)|$.

Example 3.1.1. Consider the digraph $\mathbf{G} = (\{0,1,2\}, \{(0,2), (1,0), (1,2), (2,0), (2,2)\})$. Then we have the sets:

$$N_{out}(0) = \{2\}\ N_{in}(0) = \{1,2\}$$
$$N_{out}(1) = \{0,2\}\ N_{in}(1) = \emptyset$$
$$N_{out}(2) = \{0,2\}\ N_{in}(2) = \{0,1,2\}$$

We will appreciate the usefulness of defining these neighbourhoods when we present the idea of vertex domination in Chapter 6.

In the next chapter we will define the constraint satisfaction problem and we will show the natural connection between these decision problems and the digraph homomorphisms we discussed in Chapter 2.

Chapter 4

Constraint Satisfaction Problems

Constraint satisfaction problems, or CSPs, are mathematical problems where one must find values which satisfy a set of constraints or criteria. They can be used to represent many real world problems such as those of scheduling or the allotment of resources, they can be used to define and frame many different games and puzzles, and are frequently found in various areas of computer science such as the study of Artificial Intelligence. The goal when working with CSPs is to determine an assignment of values given to variables from a specific domain which satisfy the given constraints. This can also be framed as a decision problem by asking whether not such an assignment exists. This framework is obviously quite versatile and can be used to describe many types of problems. This is why CSPs have been actively studied for many decades and through the lens of many different disciplines. We begin with the formal definition of a constraint satisfaction problem and then present some alternate views using homomorphisms.

4.1 The formal approach to CSPs

Definition 4.1.1. A ***constraint satisfaction problem*** is a triplet (V, D, C) Where $V = \{V_1, \ldots, V_n\}$ is a set of ***variables***, D is a non-empty domain, and $C = \{C_1, \ldots, C_m\}$ is a set of ***constraints***. A ***constraint***, with arity k, is a pair (t, U) where $t = (t_1, \ldots, t_k)$ is a tuple of variables such that each $t_i \in V$ and $U = \{u_1, \ldots, u_s\}$ is a set of tuples with each $u_i \in D^k$.

Definition 4.1.2. A ***valuation*** of a CSP is a mapping $f : V \to D$. We say that a valuation ***satisfies*** a constraint $C = (t, U)$ if $(f(t_1), \ldots, f(t_k)) \in U$, where k is the arity of the constraint. We call a valuation a ***solution*** to a constraint satisfaction problem if it is satisfies all constraints.

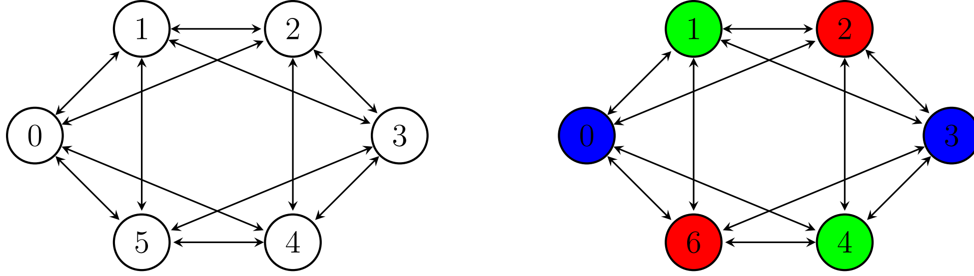


Figure 4.1: 3-Colouring of a digraph with 6 vertices where no edge is shared between two vertices of the same colour.

Constraint satisfaction problems can be found in our every day experiences in the games of sudoku or newspaper crosswords. In computer science they can be found in the canonical example of graph colouring. To illustrate these problems further we provide a few examples of CSPs below.

Example 4.1.1. Let $\mathbf{G} = (V, E)$ be a graph. A k-colouring of the graph $\mathbf{G}$ is an assignment of one of k possible colours to each vertex $v \in V$ such that no two adjacent vertices receive the same colour. In the language of a constraint satisfaction problem the k-colouring problem can be seen as the triplet (V, D, C) where V consists of the vertices of the graph $\mathbf{G}$, the domain, D, is the set $\{1, \ldots, k\}$ of k colours. The binary relation consists of all pairs of distinct colours, $R_{\neq} = D^2 \setminus \{(1,1), (2,2), \ldots, (k,k)\}$. For any edge $(v_i, v_j) \in E(\mathbf{G})$ we have the constraint $C = ((v_i, v_j), R_{\neq})$.

In Figure 4.1 we see that a solution to this 3-colouring CSP with domain $D = \{R, G, B\}$ is given by:

$$f(0) = B = f(3),$$
$$f(1) = G = f(4),$$
$$f(2) = R = f(5)$$

Example 4.1.2. Sudoku is a puzzle game that involves placing the numbers from the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ into various empty spaces on a 9×9 grid. Each row and column can only have one instance of each of the numbers 1 through 9 and each 3×3 grid marked by the thicker lines can also only contain one instance of those numbers as well.

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

Unsolved Sudoku

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

Solved Sudoku

In the language of CSPs we would consider each cell of the grid as a variable. In this case we could label each variable as x_{ij} where i denotes the row and j the column of the particular cell. So $V = \{x_{11}, x_{12}, \ldots, x_{98}, x_{99}\}$. The domain for these variables is the possible values they can take on which is simply the numbers 1 through 9, thus we have $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The constraints that describe the restrictions on the rows, columns and grids are relations of arity 9 and we use unary relations to describe the cells that are already filled at the problems outset. Let D_{per} be the set of tuples defined by all possible permutations of the numbers 1 to 9 then if we take the top row the constraint that each variable must be assigned a different value from our domain can be expressed as $C = (\{x_{11}, x_{12}, \ldots, x_{19}\}, D_{per})$. We can also give a unary constraint for the the values that are already present at the puzzles outset. From the example above we have $C_{32} = (\{x_{32}\}, (3))$ and $C_{67} = (\{x_{67}\}, (7))$.

4.2 CSPs as homomorphism problems

Constraint satisfaction problems can also be framed in another interesting and useful way. Feder and Vardi observed that we can consider each CSP as a ***homomorphism problem*** in [2]. Informally we can imagine that the set of variables defines one structure, **G**, and the domain defines another structure, **H**, and the valuation h is a map between the two. If we define our structures properly and the mapping h defines a homomorphism between our structures then this homomorphism provides a solution to the CSP. To convey this more formally we offer the following definition.

Definition 4.2.1. Given a constraint satisfaction problem defined by the triplet (V, D, C) we define two relational structures **G** and **H** such that $V(\mathbf{G}) = V$ and $V(\mathbf{H}) = D$. Using each set of tuples U occurring in some constraint $(t, U) \in C$, with $t = \{t_1, \ldots, t_k\}$, we define a k-ary relation S_i on $V(\mathbf{H})$ for which the corresponding relation R_i on $V(\mathbf{G})$ consists of all k-tuples t' of variables with a constraint defined as $C' = (t', U)$. Then a ***solution*** to CSP(**H**) is a homomorphism $h : \mathbf{G} \to \mathbf{H}$.

This definition is incredibly useful when we turn our gaze to the theoretical and view this through the lens of the complexity classification of various CSPs. In Example 4.1 we saw how we could represent the k-colouring problem as a CSP. Now we give an illustrative example of how this CSP can be represented as a homomorphism problem.

Example 4.2.1. It is well known that the problem of k-colouring graphs is closely linked to the complete graph on k vertices. It is easy to see that one can transform the question of k-colouring a particular graph to that of finding a homomorphism from that graph to the complete graph on k vertices. In Figure 4.1 we showed a 3 colouring of an undirected graph, $\mathbf{G}$, where $\mathbf{G} = (\{0,1,2,3,4,5\}, \{(0,1),(0,2),(0,4),(0,5),(1,2),(1,3),(1,5),$
$(2,3),(2,4),(3,4),(3,5),(4,5)\})$ and $\mathbf{K}_3 = \{(a,b,c), \{(a,b),(a,c),(c,b)\}\}$. This implies that there exists a homomorphism h that goes from $\mathbf{G}$ to $\mathbf{K}_3$.

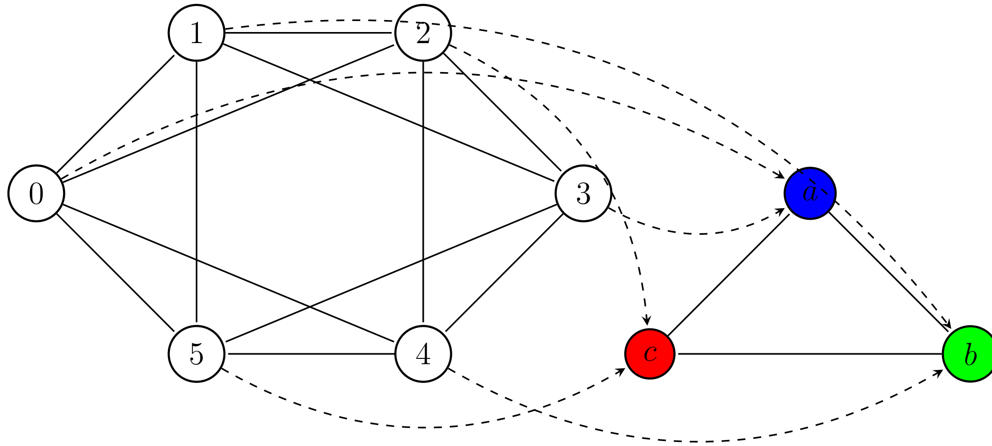
We can define the homomorphism h as:

$$\begin{aligned} h(0) &= a, \\ h(1) &= b, \\ h(2) &= c, \\ h(3) &= a, \\ h(4) &= b, \\ h(5) &= c \end{aligned}$$

We can see that this maintains the edge relations of $\mathbf{G}$ in $\mathbf{K}_3$

$$\begin{aligned} h((0,1)) &= (a,b), \\ h((0,2)) &= (a,c), \\ h((0,4)) &= (a,b), \\ h((0,5)) &= (a,c), \\ h((1,2)) &= (b,c), \\ h((1,3)) &= (b,a), \\ h((1,5)) &= (b,c), \\ h((2,3)) &= (c,a), \\ h((2,4)) &= (c,b), \\ h((3,4)) &= (a,b), \\ h((3,5)) &= (a,c), \\ h((4,5)) &= (b,c) \end{aligned}$$

Since these are undirected edges we see that each of the relations is preserved by h and therefore h defines a homomorphism from $\mathbf{G}$ to $\mathbf{K}_3$. the following diagram gives the visual interpretation of the homomorphism h:



The homomorphism h from $\mathbf{G}$ to $\mathbf{K}_3$.

In general a CSP has $\mathbf{G}$ and $\mathbf{H}$ as inputs but we can fix the target structure $\mathbf{H}$ and ask "which structures $\mathbf{G}$ have a homomorphism to $\mathbf{H}$?" Using this fixed target idea it is natural to define a particular constraint satisfaction problem in terms of a set of structures.

Definition 4.2.2. Let $\mathbf{H}$ be a relational structure then CSP($\mathbf{H}$) is defined as the set of all similar relational structures $\mathbf{G}$ such that there exists a homomorphism $h : \mathbf{G} \to \mathbf{H}$. We call the structure $\mathbf{H}$ the ***target*** structure.

Example 4.2.2. Looking back at Example 4.2.1 we would say that $\mathbf{G} \in$ CSP($\mathbf{K}_3$) because we have a homomorphism $h : \mathbf{G} \to \mathbf{K}_3$.

Notice that by Proposition 3, in Chapter 2, we have that if $\mathbf{C}$ is the core of some structure $\mathbf{G}$ then CSP($\mathbf{C}$) = CSP($\mathbf{G}$). This nice property is one of the reasons cores are helpful in the study of constraint satisfaction problems and one of the reasons digraphs with lists, which are themselves cores, are central in this work.

Chapter 5

Descriptive and Computational Complexity

Constraint satisfaction problems arise naturally in the study of computational complexity. This is because so many everyday and scientific problems can be easily translated into the language of CSPs. This motivates computer scientists and mathematicians alike to study and understand the difficulty or ”hardness” in solving general forms of these problems. Since the concepts of computational complexity were formalized in the 1960s and 1970s, by the likes of Hartmans and Stearns [10], Levin [11] and Cook [12] and others, CSPs were among some of the earliest problems studied in this new context.

In this chapter we will briefly discuss the well-known complexity classes P and NP as well as the methods for ordering problems in these classes in terms of their relative hardness. We will also define some subclasses of P, such as the classes L and NL as well as problems that are known as first order definable (FO). We make no pretense of this being a complete or comprehensive study of this subject but more of a primer to show the underlying motivations for the work presented in the rest of this thesis. We direct the reader to [13] or [14] for a more comprehensive reference. We will present the famous dichotomy of Feder and Vardi found in [1] for the complexity of CSPs and explain how this motivates our current study. Finally, we conclude with an illustrative example of a problem that is first order definable. We will link this descriptive complexity to our study of digraph defined constraint satisfaction problems in the next chapter.

5.1 Efficient computation, efficient verification and the complexity classes P and NP

The complexity classes of P and NP are very familiar to anyone who has a basic knowledge of complexity theory. They have also been made infamous in popular science culture because of the well known open question ”Does P equal NP?” whose proof offers a million dollar prize and endless fame to the researchers who finally crack it. Motivated by the elegance and simplicity found in the forthcoming text by Avi Wigderson [15] we will present what we believe are the simplest constructions of these very important classes.

The easiest way to understand complexity is to think of it as the study of what can be ***efficiently*** computed. Of course the word efficient can be defined in many different ways from many different viewpoints but very generally we think of it as computation that uses a reasonable amount of time and a reasonable amount of space in order to arrive at some conclusion. One class of problems that arise naturally in the study of Computer Science are ***decision problems***. These problems can simply be thought of as questions with a yes or no answer. For example we could ask if the number 1993 is a prime number?, or if a particular graph, **G**, 3-colourable?, or if one can solve a Rubik's cube with a particular configuration in 21 moves or less? and so on. A convenient way to frame a question into a decision problem is to consider whether a particular object has a specific property and then divide these objects into sets which have this property and those that do not. We can ask does the number 1993 belong to the set of primes? or does it belong to the complement set of composite numbers? Does this particular Rubik's cube configuration belong to the set of configurations, R_{21}, of configurations that can be solved in 21 moves or less? To illustrate these points we formalize the famous 3-colourable graph problem in the following example:

Example 5.1.1. Let C_3 be the class of 3-colourable graphs and let $\mathbf{G} = (V, E)$ be a graph then the framework for a decision problem is:

Problem : C_3

Input: the graph **G**

Output: True if $\mathbf{G} \in C_3$ and false if $\mathbf{G} \notin C_3$.

This example illustrates how we can set up a decision problem for a graph **G** and some characteristic C. By describing the problem in this way we are able to try to describe a solution-finding algorithm for a Turing machine to perform in order to determine whether the input graph **G** has

the characteristic C. Note that every decision problem has an obvious complement, $\bar{C}$ which represents the set of all structures not in C. In this case $\bar{C_3}$ would contain all graphs which are not 3-colourable.

An algorithm, by definition, must at some point stop for every input and give an output to the decision problem, but different algorithms can take different amounts of time to complete their tasks. An algorithm is thought to be efficient if it stops in a reasonable amount of time, but how can we define this notion of reasonable? It has been agreed upon, for many reasons, that polynomial time, as opposed to exponential time, is a good definition for reasonable. This implies that the runtime for a particular input of length n is bounded by a polynomial function in n. Problems that have such a solution are said to be contained in the complexity class P. We will use the definition found in [15] to formally define this class. Let I define the set of all binary sequences. This is the set of all possible inputs to our decision problem. Now let I_n be the set of all binary sequences of size n, eg. $I_n = \{0,1\}^n$ then we can define the complexity class P as follows:

Definition 5.1.1. A function $f : I \to I$ is in the ***complexity class*** P if there is an algorithm computing f and positive constants A and c, such that for every n and every $x \in I_n$ the algorithm computes $f(x)$ in at most An^c steps.

We consider each of these steps to be a basic operation. Notice that this definition works well with decision problems since the output bit can be either a 1 or a 0, or in other words "yes" or "no". There are many non-trivial and natural problems that are known to belong to P such as instances of factoring polynomials, finding greatest common divisors, prime checking and more. Often extensive study is needed in order to discover an elegant polynomial time algorithm which solves problems in P, but once an algorithm is presented it is often the case that these algorithms can be made more and more efficient than their first formulations. For every problem that is found to be in P there are many more which have yet to find their polynomial time algorithm. Are we naive to think that every problem can have a polynomial time algorithm which solves it? How do we know we are not wasting our time in searching for one? Are there properties inherent in problems that can point to which are more likely to be found in P and which are not?

An interesting fact of a decision problem is that it is often easier to determine whether a particular input satisfies some property than it is to come up with an algorithm to check all general inputs. These "acts of verification" have varying degrees of complexity themselves. This begs the question: is it easier to find algorithms that determine whether a general input has a particular property if it is easier to verify that a particular input has that property? For example, if you are given a graph that is 3-coloured it is far easier to check if the 3-colouring is proper (ie. no

vertices with an edge between them have the same colour) then to find such a colouring. We are not concerned with colouring the graph but simply verifying that the given colouring is proper or not. These problems are connected but different. It is well know that the verification problem of 3-colouring can be done efficiently whereas the problem of finding a 3-colouring might not be. So what does efficient verification say about a problem in general?

The class NP is born from this idea of verification. NP contains all the properties C where the question "is $x \in C$?" has an efficient verification process. Again, we use polynomials to define the idea of efficient. In this case we say that the verification is efficient if the candidate proof, y, can be defined in a length that is polynomial to the length n of the input x and it can be checked using a verification algorithm V_C within a time which is also polynomial to this length n.

Definition 5.1.2. The set C is in the ***complexity class*** NP if there is a function $V_C \in \mathsf{P}$ and a constant k such that:
(i) If $x \in C$ then there exists a y with $|y| \leq k|\dot{x}|^k$ and $V_C(x, y) = 1$.
(ii) If $x \notin C$ then for all y we have $V_C(x; y) = 0$.

Problems in the class P are defined by the efficiency of the algorithms designed to classify them whereas problems in the class NP are defined by the efficiency in which one can verify that a given input has a property or not. Two problems known to be in NP are the problem of finding Hamiltonian paths in graphs and the general problem of finding graph isomorphisms [15]. It's clear that $\mathsf{P} \subset \mathsf{NP}$ but the other way around is a far different story. The idealist in us may hope that by finding an efficient verification for a problem we might be closer to an efficient algorithm for solving the problem but the majority of computer scientists do not agree and that is why we have the famous conjecture $\mathsf{P} \neq \mathsf{NP}$.

Arguably the most important and powerful tool in the study of complexity is that of reductions. A reduction is a function that allows a complexity comparison of two problems that may, at first glance, seem completely disjoint. Basically, if we are able to translate one problem C, into another problem D in an efficient way, then this means that C is at most as difficult as D.

Definition 5.1.3. Let $C, D \subset I$ be two classification problems. We call $f : I \to I$ an ***efficient reduction*** from C to D if $f \in \mathsf{P}$ and for every $x \in I$ we have that $x \in C$ if and only if $f(x) \in D$. We write $C \leq D$ if there exists an efficient reduction from C to D.

The usefulness of reductions should be immediately apparent. Our first obervation is that because P is closed under composition thr relation $\leq$ is transitive and therefore defines a partial

ordering on classification problems. Our second observation is that if one is able to efficiently reduce a problem C to D and there already exists an efficient algorithm for D then we can construct an efficient algorithm for C. This offers researchers another route for classifying particular problems. Instead of looking for a direct algorithm to solve a particular problem one could find a reduction to a different problem that has been already classified. This partial ordering also allows us to define the hardest problems in any given complexity class.

Definition 5.1.4. A problem D is called C-$hard$ if for all $C \in \mathsf{C}$ we have that $C \leq D$ and if $D \in \mathsf{C}$ then we call D C-$complete$.

These complete problems, which are the most difficult problem in any particular complexity class, are very important because they capture the complexity of their entire class. For example we know that the problem of boolean satisfiability known as SAT is NP-$complete$. SAT is the set of all satisfiable boolean statements. It was an incredibly important result of both Cook [12] and Levin[11] to show that this was in fact NP-$complete$. Returning to example 5.1.1 it is well known that this particular decision problem, known as the 3-colouring problem, is NP-$complete$. We know this because there exists a reduction from SAT to 3-colouring. We will not provide this reduction here because it is outside the scope of this work but it can be found in [15].

There are obvious similarities between boolean satisfiability and the general constraint satisfaction problem. They are both NP-$complete$. Even though the problem of CSPs is hard in general it is possible to find CSPs with certain characteristics that are tractable. This fact is captured in the famous dichotomy conjecture of Feder and Vardi.

Conjecture 4. *[1]For all relational structures, **A**, the problem CSP(**A**) is either in* P *or* NP-$complete$.

This means that a constraint satisfaction problem is either tractable or it is as hard as SAT and the other hardest problem in NP. In this study we will focus on CSPs that are tractable and belong to a simple form of descriptive complexity, associated with first order logic, known as FO. Problems that are in FO are those which can be captured by first order logical sentences. It is know that problems with first order definability are contained in P. In order to better understand FO we will describe two different classes also contained in P. These are L and NL. By defining these classes and their inclusions we hope to give a greater understanding of these tractable problems.

In order to understand these complexity classes it is important to understand that Turing machines use two resources, time and space. In order to classify a problem we must focus on the limits of these resources.

Definition 5.1.5. Let the function $f : \mathbb{N} \to \mathbb{R}^+$ then:

(1) $\mathsf{TIME}(f(n))$ is the class of problems that can be solved by a deterministic Turing machine in $O(f(n))$ steps.

(2) $\mathsf{NTIME}(f(n))$ is the class of problems that can be solved by a non-deterministic Turing machine in $O(f(n))$ steps.

(3) $\mathsf{SPACE}(f(n))$ is the class of problems that can be by a deterministic Turing machine using $O(f(n))$ blocks of memory.

(4) $\mathsf{NSPACE}(f(n))$ is the class of problems that can be solved by a non-deterministic Turing machine using $O(f(n))$ blocks of memory.

5.2 L

The class L contains those problems which can be solved by a deterministic Turing machine using only a logarithmic amount of memory space relative to the input. Thus we have that $\mathsf{L} = \mathsf{SPACE}(log(n))$. This class can be called ***deterministic log space***. An example problem in this class is the decision problem determining whether a graph is 2-colourable.

5.3 NL

The class NL is the non-deterministic counterpart to the class L and therefore contains those problems which can be solved by a non-deterministic Turing machine using only a logarithmic amount of memory space relative to the input. Thus we have that $\mathsf{NL} = \mathsf{NSPACE}(log(n))$. This class can also be called ***non-deterministic log space***. An example problem in this class is the decision problem for determining whether there exists a directed path between two vertices in a digraph. It is clear that L is contained in NL but like P and NP it is not known whether $\mathsf{L} = \mathsf{NL}$.

5.4 FO

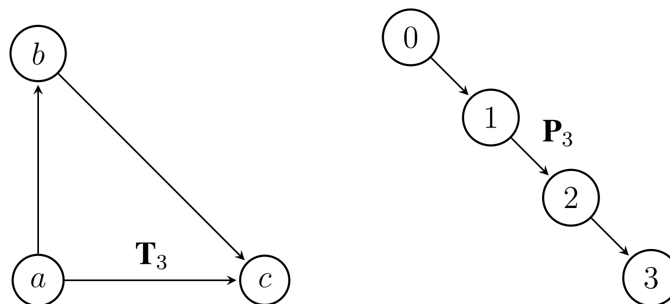
A set of structures S belongs to the descriptive complexity class FO if there exists a first order sentence ϕ such that $G \in S$ if and only if G satisfies ϕ; a first order sentence is one that can be written using atomic formulas, connectives and both the universal and existence quantifiers. It is known that FO is a proper subclass of L. [16]

With these definitions in place it is possible to give the following chain of inclusions to give the reader a better understanding of where FO definability is positioned in terms of relative hardness.

$$\mathsf{FO} \subset \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP}$$

Through our homomorphism point of view of the CSP we have that if we take some fixed finite structure **G** then the question presented by CSP(**G**) is "which structures have a homomorphism to **G**?". It is well known that in general the problem CSP(**G**) is in NP but it natural to wonder what kind of structures yield tractable problems. The easiest of these problems, that are not trivial, would be those which can be defined by first order logic. It is reasonable to ask if it is possible to characterize which constraint satisfaction problems are first order definable and we now provide an example of a digraph with such a property..

Example 5.4.1. Let $\mathbf{T}_3$ be the transitive tournament on 3 vertices and let $\mathbf{P}_3$ be a directed path of length 3. It is well known and easy to see that a digraph **G** has a homomorphism to $\mathbf{T}_3$ if and only if there does not exist a homomorphism from $\mathbf{P}_3$ to **G**. This restriction can be defined in a first order sentence as follows: $\neg(\exists a \exists b \exists c \exists d\ E(a, b) \wedge E(b, c) \wedge E(c, d))$ where $E(x, y)$ indicates that there is an arc from x to y. This sentence clearly states that there is no homomorphism from $\mathbf{P}_3$ to **G**. Thus CSP($\mathbf{T}_3$) is FO-definable. We will confirm this in the next chapter using a special algorithm that compares the coordinates in the square of $\mathbf{T}_3$.



This is of course a very specific example of a well known result. In general these first order sentences are not always easy to formulate. Luckily there are some combinatorial methods which

can help us determine if a particular CSP is FO-definable without needing to explicitly find a first order sentence to describe it.

There is a simple and easy algorithm for determining which relational structures with lists have CSPs which are FO-definable. It is known as the dismantling algorithm and it is the main subject of our next chapter.

Chapter 6

Characterizations of Relational Structures That Define CSPs Which Are FO-definable With Lists

In the previous chapter we saw an example of a constraint satisfaction problem that was definable in first order logic. In this chapter we present a general technique for determining if a certain relational structure with lists defines a CSP that is FO-definable or not. This is the dismantling algorithm and it was first defined by Larose, Loten and Tardif in [5]. We will also present a characterization described by Lemaître in [6] for which digraphs with lists satisfy this algorithm. Motivated by these results we begin our investigation of CSPs defined by larger relational structures, with many binary relations and lists, by focusing on the special case of transitive tournaments. We present a new algorithm for classifying these types of structures and give a full classification. Lastly, we generalize the classification theorem for a single digraph for those structures with lists that have multiple binary relations. Lastly, using the various results of this chapter we present examples of obstructions in particular structures as well as some ideas on how to build structures that will define constraint satisfaction problems that are FO-definable.

6.1 The dismantling algorithm

There is a simple algorithm based on comparing relations in the square of a relational structure that can determine whether a particular structure has a CSP which is FO-definable. In order to describe this algorithm we will first to introduce the idea of domination in a relational structure.

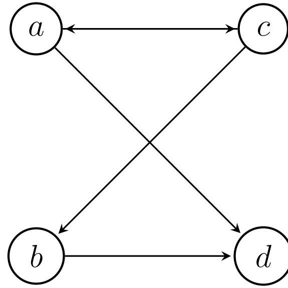
Definition 6.1.1. Let $\mathbf{A} = \langle A; R_0, R_1, \ldots, R_n \rangle$ be a relational structure. Let $x, y \in V(\mathbf{A})$ where $x \neq y$. We say that x ***dominates*** y in $\mathbf{A}$ if for all relations R_i of $\mathbf{A}$ we have $(y_1, \ldots, y_n) \in R_i$, with $y_j = y$, implies that $(x_1, \ldots, x_n) \in R_i$ with $x_j = x$ and $y_k = x_k$ for all $k \neq j$. We say that y is ***dominated*** in $\mathbf{A}$ if there exists a vertex $x \in V(A)$ such that x dominates y.

The definition of domination becomes simpler when we focus specifically on the study of digraphs. We can restate this definition using the concepts of in and out neighbourhoods of a particular vertex.

Definition 6.1.2. Let $\mathbf{G} = (V, E)$ be a digraph and let $x, y \in V(\mathbf{G})$. We say that x ***dominates*** y in $\mathbf{G}$ if for all edges of the form (y, a) and (a, y) where a is some vertex in V, we also have that $(x, a), (a, x) \in E$. This is equivalent to saying that x dominates y if $N_{in}(y) \subseteq N_{in}(x)$ and $N_{out}(y) \subseteq N_{out}(x)$.

Notice that in this definition the in and out neighbourhoods do not need to be proper subsets in order for there to be domination. This means that if two vertices have the exact same relations then x dominates y and y dominates x. Also notice that any isolated vertex v is dominated by any other vertex, isolated or not, because for any isolated vertex we have that $N_{out}(v) = \emptyset$ and $N_{in}(v) = \emptyset$.

Example 6.1.1. Consider the digraph $\mathbf{G} = \{(a, b, c, d), \{(a, c), (a, d), (b, d), (c, a), (c, b)\}\}$

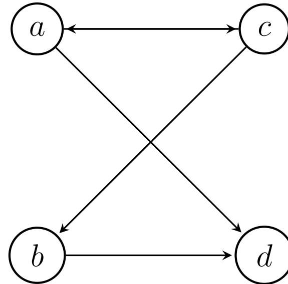


It is easy to see that the vertex a dominates the vertex b either by quickly reviewing the relations on $\mathbf{G}$ or by general visual inspection. We can also write it out more formally using the in and out edge sets of both a and b. For the vertex a we have $N_{out}(a) = \{c, d\}$ and for the vertex b we have $N_{in}(a) = \{c\}$ where $N_{out}(b) = \{d\}$ and $N_{in}(b) = \{c\}$. Thus we see that $N_{out}(a) \supseteq N_{out}(b)$ and $N_{in}(a) \supseteq N_{in}(b)$ and these are the conditions for vertex domination.

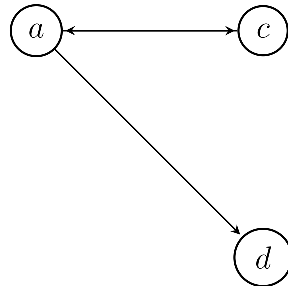
Definition 6.1.3. Let $\mathbf{A}$ be a relational structure and let there be a sequence $\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_n$ where $\mathbf{A}_0 = \mathbf{A}$ and $\mathbf{A}_n = \mathbf{A}'$ such that for $0 \leq i \leq n$ we have that $\mathbf{A}_i$ is the induced substructure of $\mathbf{A}_{i-1}$ with $V(\mathbf{A}_i) = V(\mathbf{A}_{i-1}) \setminus \{x\}$ where x is dominated in $\mathbf{A}_{i-1}$. We say that $\mathbf{A}$ ***dismantles*** to $\mathbf{A}'$.

Continuing with the digraph in Example 6.1.1 we can proceed further with the dismantling process. In the next part of the example we remove the dominated vertex and show the induced subdigraph that remains.

Example 6.1.2. In Example 6.1.1 we had $\mathbf{G} = ((a, b, c, d), \{(a, c), (a, d), (b, d), (c, a), (c, b)\})$ and observed that the vertex a dominated the vertex b thus in our dismantling we are to remove the edges involving b. This means we would remove the edges (b, d) and (c, b). Leaving us with the graph below.



The digraph $\mathbf{G}$ where the vertex b is dominated by he vertex a.



The induced subdigraph $\mathbf{G} \setminus \{b\}$ with the edges (b, d) and (c, b) removed.

At this point the induced subdigraph $\mathbf{G} \setminus \{b\}$ does not contain any more dominated vertices and the dismantling can not be performed further. It turns out that the structure $\mathbf{A}'$, the remaining induced substructure of $\mathbf{A}$, can be instrumental in determining the complexity of CSP($\mathbf{A}$).

Theorem 5. *[5] Let* $\mathbf{A}$ *be a relational structure then the problem CSP(*$\mathbf{A}$*) is* FO*-definable if and only if there is a core* $\mathbf{C}$ *of* $\mathbf{A}$ *such that* $\mathbf{C}^2$ *dismantles to the diagonal.*

Since relational structures with lists are also always cores we are able to rewrite this theorem as the following useful corollary.

Corollary 6. *[5] Let* $\mathbf{A}$ *be a relational structure then the problem CSP(*$\mathbf{A}_L$*) is* FO*-definable if and only if* $\mathbf{A}_L^2$ *can be dismantled to the diagonal.*

Since we will be focusing our attention on the dismantling of digraphs with lists the following lemma will be extremely useful. In fact this lemma will be one of our primary tools used in many of the proofs that follow later.

Lemma 7. *Let $\mathbf{G}$ be a digraph and let $x, y \in V(\mathbf{G})$. If $\mathbf{G}_L^2$ dismantles to the induced substructure $\mathbf{H}$ that contains the diagonal then if (x, y) is dominated in $\mathbf{H}$ it is dominated by either (x, x) or (y, y).*

Proof. Let $\mathbf{G}_L$ be a digraph with lists and let $x, y \in V(\mathbf{G}_L)$. Then by the definition of the square of a digraph with lists we have that $(x, x), (x, y), (y, x), (y, y) \in V({\mathbf{G}_L}^2)$ and that $R = \{(x, x), (x, y), (y, x), (y, y)\}$ is a relation of $\mathbf{G}_L^2$. This means that the vertex (x, y) must be dominated by (x, x), (y, x) or (y, y) because in order to dominate (x, y) in $\mathbf{H}$ a vertex must also satisfy R. If it is (x, x) or (y, y) we are done. So we suppose (y, x) dominates (x, y) and show that if this is the case then (x, x) and (y, y) must dominate as well.

By the definition of domination if (y, x) dominates (x, y) then for any $(u, v) \in V(\mathbf{H})$ if $((x, y), (u, v)) \in E(\mathbf{H})$ then we have $((y, x), (u, v)) \in E(\mathbf{H})$. This implies that we also have $((x, x), (u, v)), ((y, y), (u, v)) \in E(\mathbf{H})$. Similarly if we have $((u, v), (x, y)) \in E(\mathbf{H})$ then we have $((u, v), (y, x)) \in E(\mathbf{H})$ then we also have $((u, v), (x, x)), ((u, v), (y, y)) \in E(\mathbf{H})$. Which means that if (y, x) dominates (x, y) then so do both (x, x) and (y, y). □

This is a very useful property because it simplifies the dismantling process significantly for digraphs with lists. To check the domination of any particular vertex (x, y) we do not need to compare its in and out neighbourhoods with all other vertices in the square of the digraph but just (x, x) and (y, y). For larger structures this greatly reduces the time needed to confirm if a structure dismantles to the diagonal or not.

6.2 A classification of digraphs with lists whose CSPs are FO-definable

In a previous study [6] Lemaître introduced a classification for the FO-definability of a CSP defined by a single digraph. It was found that there are two digraph characteristics that impede the dismantling process and therefore act as markers for digraphs that do not have FO-definable CSPs. The first property is that all digraphs which dismantle must be telescopic. To understand this property we need to define add another definition to our digraph vocabulary.

Definition 6.2.1. Let $G = (V, E)$ be a digraph where $a, b, c, d \in V$ and $(a, b), (c, d) \in E$ but $(a, d), (c, b) \notin E$ then we call the edges (a, b) and (c, d) ***separated edges***. (Figure 6.1)

Definition 6.2.2. Let $G = (V, E)$ be a digraph if G does not contain any ***separated edges*** then we call G a ***telescopic digraph***.

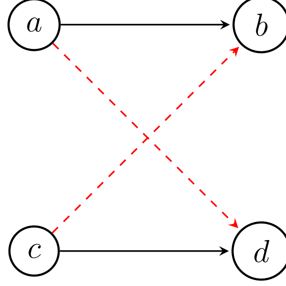


Figure 6.1: Digraph with separated edges (a, b) and (c, d). (The red dotted edge indicates that this edge is not present)

It can be shown that a digraph that has separated edges, and is therefore not telescopic, can not be dismantled:

Proposition 8. *Let $\mathbf{G}_L$ be a digraph with lists. If $\mathbf{G}_L^2$ dismantles to the diagonal then $\mathbf{G}_L$ is telescopic.*

Proof. Let $(a, b), (c, d) \in E(\mathbf{G}_L)$ be separated edges. Thus we have $((a, c), (b, d)) \in E(\mathbf{G}_L^2)$ but $((a, a), (b, d)), ((c, c), (b, d)) \notin E(\mathbf{G}_L^2)$. This means that the vertex (a, c) can not be dominated until we remove the vertex (b, d). Notice we have $((a, c), (b, d)) \in E(\mathbf{G}_L^2)$ but $((a, c), (b, b))$, $((a, c), (d, d)) \notin E(\mathbf{G}_L^2)$. This means thats (c, d) can not be dominated until we remove (a, c) therefore neither will be dominated in the dismantling and we will not arrive at the diagonal. □

It turns out that the telescopic property is not enough to characterize which digraphs have squares that dismantle to the diagonal or not. Even in telescopic digraphs another obstruction was found in [6] and we call this obstruction an ***impeding bicycle***. This obstruction involves two directed cycles, an upper and a lower, (hence the bicycle name) with the forced exclusion of a particular edge for each corresponding pair of edges.

Definition 6.2.3. Let $G = (V, E)$ be a digraph we say that G has an ***impeding bicycle*** if there exists vertices $x_0, x_1, \dots, x_k, y_0, y_1, \dots, y_k \in V$ such that for all t, $1 \le t \le k$, we have $(x_t, x_{t+1}), (y_t, y_{t+1}) \in E$ and $(x_t, y_{t+1}) \notin E$. The indices are defined modulo $k + 1$ (eg. $x_{k+1} = x_0$). (Figure 6.2)

Notice that if the digraph G is telescopic, and thus contains no separated edges, then we must have $(x_i, y_{i+1}) \in E(\mathbf{G}_L)$ because the edge $(y_i, x_{i+1}) \notin E(\mathbf{G}_L)$. Now we show that the square of a digraph which contains one of these impeding bicycles can not dismantle to the diagonal.

Lemma 9. *[6] Let $\mathbf{G} = (V, E)$ be a digraph. If $\mathbf{G}$ contains an impeding bicycle then $\mathbf{G}_L^2$ does not dismantle to the diagonal.*

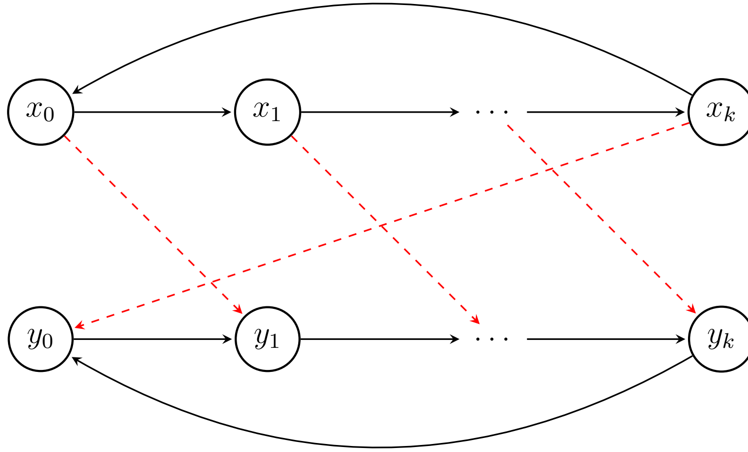


Figure 6.2: An impeding bicycle

Proof. Suppose $\mathbf{G}$ contains an impeding bicycle with the sequence $(x_0, x_1, \ldots, x_k)$ as its top and $(y_0, y_1, \ldots, y_k)$ as its bottom. Suppose that $\mathbf{G}_L^2$ does in fact dismantle to the diagonal. Let the vertex $(x_i, y_i) \in V(\mathbf{G}_L^2)$ be the first vertex of the impeding bicycle to be dismantled. Then (x_i, y_i) is dominated by either (x_i, x_i) or (y_i, y_i) in a substructure of $\mathbf{G}_L^2$ that contains the entire bicycle. Notice that $((x_{i-1}, y_{i-1}), (y_i, y_i)), ((x_i, x_i), (x_{i+1}, y_{i+1})) \notin E(\mathbf{G}_L^2)$ but $((x_{i-1}, y_{i-1}), (x_i, y_i)), ((x_i, y_i), (x_{i+1}, y_{i+1})) \in E(\mathbf{G}_L^2)$. This means that (x_i, y_i) is not dominated by either (x_i, x_i) or (y_i, y_i) and we arrive at a contradiction.

□

The reason that this impeding bicycle obstructs dismantling can be found in its repeating structure. For any $a_i, b_i \in V(\mathbf{G})$ that appear in our bicycle we are not able to remove the vertex $(x_i, y_i) \in V(\mathbf{G}_L^2)$ during the dismantling of $\mathbf{G}_L^2$ without first removing a different vertex (x_j, y_j). Thus we can not remove any.

We now describe a particular configuration that can not be present in the induced substructure $\mathbf{H}$ of $\mathbf{G}_L^2$, obtained by dismantling, when $\mathbf{H}$ is not the diagonal. This property will aid us in the proof of Proposition 11.

Proposition 10. *[6] Let $\mathbf{G}$ be a telescopic digraph such that $\mathbf{G}_L^2$ does not dismantle to the diagonal. Let $\mathbf{H}$ be the induced subgraph of $\mathbf{G}_L^2$ that is not the diagonal and such that there is no vertex in $\mathbf{H}$ that is dominated. Then there can not exist vertices $(a, b), (c, d), (e, f) \in E(\mathbf{G})$ such that:*

$$((c, d), (a, b)), ((a, b), (e, f)) \in E(\mathbf{H}) \text{ and}$$
$$((c, d), (a, a)), ((a, a), (e, f)) \notin E(\mathbf{G}_L^2)$$

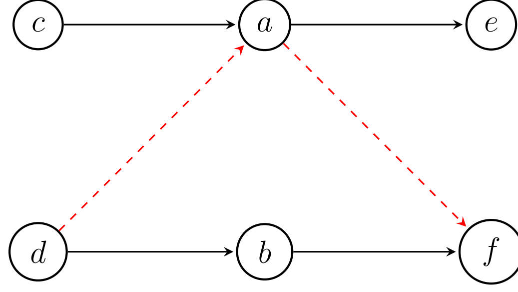


Figure 6.3: Configuration from Proposition 10.

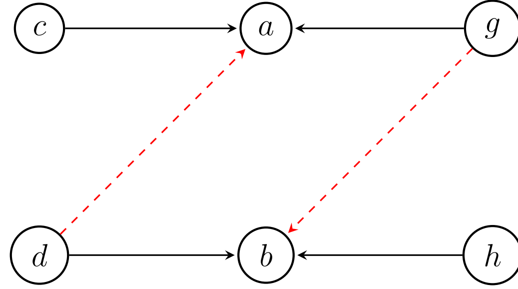


Figure 6.4: Case (1) from Proposition 10.

Proof. We will suppose that we have 3 such vertices and obtain a contradiction. Since **G** is telescopic we have that $(c, b), (b, e) \in E(\mathbf{G})$ (Figure 6.3). Which means we have $((c, d), (b, b)), ((b, b), (c, f)) \in E(\mathbf{H})$. Then there exists a vertex $(g, h) \in V(\mathbf{H})$ such that:

(1) $((g, h), (a, b)) \in E(\mathbf{H})$ and $((g, h), (b, b)) \notin E(\mathbf{H})$; or
(2) $((a, b), (g, h)) \in E(\mathbf{H})$ and $((b, b), (g, h)) \notin E(\mathbf{H})$.

One of these cases must occur or else (b, b) dominates (a, b) in **H** which would contradict our earlier assumption that no vertex in **H** is dominated. Suppose we have case (1) then we have $((g, h), (a, b)), ((c, d), (a, b)) \in E(\mathbf{H})$ and $((g, h), (b, b)), ((c, d), (a, a)) \notin \mathbf{H}$. This would produce the separated edges (g, a) and (d, b) but **G** is telescopic so we reach a contradiction. (Figure 6.4) Now suppose that we have case (2) then we have $((a, b), (g, h)), ((a, b), (e, f)) \in E(\mathbf{H})$ and $((a, a), (e, f)), ((b, b), (g, h)) \notin \mathbf{H}$. This would produce the separated edges (a, g) and (b, f) but **G** is telescopic and we reach another contradiction (Figure 6.5). Thus we cannot have the given structure in **H**.

□

Proposition 11. *[6] Let* **G** *be a telescopic digraph such that* $\mathbf{G}_L^2$ *does not dismantle to the diagonal*

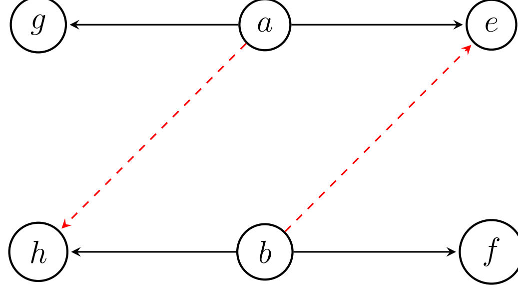


Figure 6.5: Case (2) from Proposition 10.

then $\mathbf{G}_L$ *contains an impeding bicycle.*

Proof. Since $\mathbf{G}_L^2$ does not dismantle to the diagonal we know that it dismantles to an induced substructure $\mathbf{H}$, that contains the diagonal properly, such that no vertex in $\mathbf{H}$ is dominated. Let $(x, y) \in V(\mathbf{H})$, $x \neq y$. Because (x, y) is not dominated by either (x, x) or (y, y) in $\mathbf{H}$ we have that there exists vertices $(a, b), (c, d), (e, f), (g, h) \in V(\mathbf{H})$, not necessarily unique, such that:

(i) $(x, a), (y, b), (y, a) \in E(\mathbf{G})$ and $(x, b) \notin E(\mathbf{G})$, or
(ii) $(e, x), (f, y), (e, y) \in E(\mathbf{G})$ and $(f, x) \notin E(\mathbf{G})$
and
(iii) $(x, c), (y, d), (x, d) \in E(\mathbf{G})$ and $(y, c) \notin E(\mathbf{G})$, or
(iv) $(g, x), (h, y), (h, x) \in E(\mathbf{G})$ and $(g, y) \notin E(\mathbf{G})$

Cases (i) and (ii) say that (x, x) does not dominate (x, y) and cases (iii) and (iv) say that (y, y) does not dominate (x, y) so one of each of these must be true. Cases (i) and (iii) are mutually exclusive, as are cases (ii) and (iv), because $\mathbf{G}_L$ is telescopic. If we have both (i) and (iii) we would have the separated edges (x, c) and (y, b). If we have both (ii) and (iv) we would then have the separated edges (g, x) and (f, y). Thus we must have that both (i) and (iv) are true or both (ii) and (iii) are true. Assume (i) and (iv) are true. Then using induction we can show that there exists a sequence $(x_0, y_0), \ldots, (x_n, y_n)$ with $x_i, y_i \in V(\mathbf{H})$ that satisfies the following properties:

$(x, y) = (x_0, y_0)$,
$(x_i, x_{i+1}), (y_i, y_{i+1}), (x_i, y_{i+1}) \in E(\mathbf{G})$,
$(x_i, y_{i+1}) \notin E(\mathbf{G})$

The case where $n = 1$ is clear. We assumed (i) and we let $a = x_1$ and $b = y_1$. Then we assume this sequence exists for $n = k$, ie. we have a sequence $(x_0, y_0), \ldots, (x_k, y_k)$ that satisfies

the conditions above. Now we show that the sequence exists for $n = k+1$. We know that (x_k, y_k) is not dominated by either (x_k, x_k) or (y_k, y_k) because $(x_k, y_k) \in V(\mathbf{H})$, so again we have the cases (i) and (iv) or (ii) and (iii), with $x = x_k$ and $y = y_k$. We can not have case (iii) because that would introduce the separated edges (x_{k-1}, x_k) and (f, y_k). Thus we must have (i) and (iv). If we let $a = x_{k+1}$ and $b = y_{k+1}$ in (i) then we have satisfied our conditions for $n = k+1$. We have shown by induction that the sequence $(x_0, y_0), \ldots, (x_{k+1}, y_{k+1})$ exists in $\mathbf{H}$.

Now because $\mathbf{G}$ is finite we have that both $\mathbf{G}_L^2$ and $\mathbf{H}$ are also finite. This means that if we choose a suitably large n then a vertex in the sequence $(x_0, y_0), \ldots, (x_n, y_n)$ must appear twice. Let this vertex be $(x_i, y_i) = (x_j, y_j)$ with $i < j$ then if we move back to $\mathbf{G}$ from the square we have the cycles $x_i, \ldots, x_j$ and $y_i, \ldots, y_j$ and these form the top and bottom respectively of the impeding bicycle and the condition that $(x_{t-1}, y_t) \notin E(\mathbf{G})$ for $i \leq t \leq j$ gives us the missing middle edges. Thus we have shown that if $\mathbf{G}$ is telescopic and does not dismantle to the diagonal then it contains an impeding bicycle. The same argument can be made to show the existence of an impeding bicycle if we had assumed cases (ii) and (iii) were true. □

Theorem 12. *[6] Let $\mathbf{G}$ be a digraph, then $\mathbf{G}_L^2$ dismantles to the diagonal if and only if $\mathbf{G}$ is telescopic and does not contain an impeding bicycle.*

Proof. The proof is clear from the results found above. The left side comes from Proposition 8 and Lemma 9 and the right side from Proposition 11. □

This theorem characterizes, using graph properties, all digraphs with lists that have constraint satisfaction problems that are FO-definable. Digraphs are simple relational structures that are quite easy to work with and we are naturally motivated by this result to want to investigate more complicated structures such as those with many binary relations. If we build these larger structures using only binary relations that define digraphs that dismantle to he diagonal on their own can we expect that this larger structure will dismantle as well? If not, can we characterize these larger structures with some similar obstruction in order to know their descriptive complexity? In the next section we investigate a special case where each binary relation defines a transitive tournament on the vertex set.

6.3 CSPs that are FO-definable when the relations of $\mathbf{G}_L$ are transitive tournaments

We will now investigate the interesting example of transitive tournaments. These highly structured digraphs that when you add lists have squares that always dismantle to the diagonal. These properties make them a natural candidate to begin our study of larger relational structures with many binary relations. In this section we will often be viewing these tournaments in their equivalent description as a total strict ordering on the vertex set.

We begin by showing that the square of any single transitive tournament $\mathbf{T}_n$ with lists dismantles to the diagonal.

Proposition 13. *Let $\boldsymbol{T} = (V, E)$ be a digraph where E defines a transitive tournament on V. Then $\boldsymbol{T}_L^2$ dismantles to the diagonal.*

Proof. From Theorem 12 we know that if $\mathbf{T}$ does not dismantle to the diagonal then it must have a separated edge or an impeding bicycle. We will show that $\mathbf{T}$ has neither. First we show that transitive tournaments do not contain any separated edges. If we think of our transitive tournament as a strict total ordering then for any $a, b, c, d \in V(\mathbf{T})$ if $(a, b), (c, d) \in E(\mathbf{T})$ then we have that $a < b$ and $c < d$. There are three possibilities either $a < c$, $a = c$ or $a > c$. If $a < c$ then $a < d$ and we have $(a, d) \in E(\mathbf{T})$. If $a = c$ we have both $a < d$ and $c < b$. Thus $(a, d), (c, b) \in E(\mathbf{T})$. Lastly if $a > c$ then we have $c < b$ and $(c, b) \in E(\mathbf{T})$. Thus we have no separated edges and all transitive tournaments are telescopic. Secondly, by definition transitive tournaments are acyclic and therefore cannot contain an impeding bicycle. Thus ${\mathbf{T}_L}^2$ dismantles to the diagonal and CSP($\mathbf{T}$) is FO-definable.

□

We can also show that a particular tournament is FO-definable explicitly by using our dismantling algorithm. Next we present the examples for transitive tournaments with 2 and 3 vertices.

Example 6.3.1. Let $\mathbf{T}$ be the transitive tournament on 2 vertices then $\mathbf{T} = (\{0, 1\}, \{(0, 1)\})$ and $\mathbf{T}^2 = (\{(0, 0), (0, 1), (1, 0), (1, 1)\}, \{((0, 0), (1, 1))\})$. Since both $(0, 1)$ and $(1, 0)$ are isolated vertices in $\mathbf{T}^2$ both can be removed leaving us with the diagonal. Therefore CSP($\mathbf{T}_L$) is FO-definable.

Example 6.3.2. Let $\mathbf{T}$ be the transitive tournament on 3 vertices then $\mathbf{T} = (\{0, 1, 2\}, \{(0, 1), (0, 2), (1, 2)\})$.

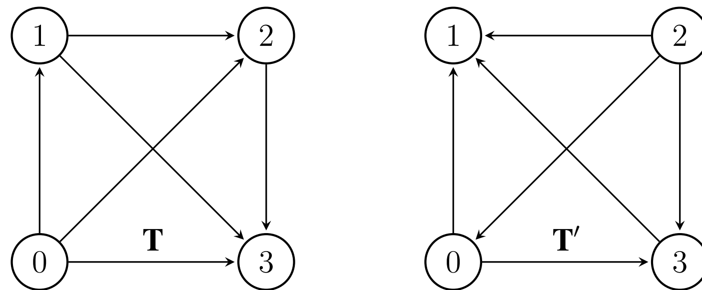
The square $\mathbf{T}^2 = (\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$, $\{((0,0),(1,1)),((0,0),(1,2)),((0,1),(1,2)),((0,0),(2,1)),((0,0),(2,2)),((0,1),(2,2)),$ $((1,0),(2,1)),((1,0),(2,2)),((1,1),(2,2))\})$.

We will show that all vertices of $\mathbf{T}^2$ outside of the diagonal are dominated in the first step. Both $(1,0)$ and $(0,1)$ are dominated by $(0,0)$ and both $(2,1)$ and $(1,2)$ are dominated by $(2,2)$. Lastly we have that $(0,2)$ and $(2,0)$ are isolated so they are dominated by both $(0,0)$ and $(2,2)$. Thus we can remove these 6 vertices in one step and be left with the diagonal. Thus CSP($\mathbf{T}_L$) is FO-definable.

One can define a unique transitive tournament, up to isomorphism, on any set of n vertices. Thus, when determining whether a set of relations, which all define transitive tournaments on a set of vertices, dismantles to the diagonal or not we only need to consider the difference in the ordering of the vertices for each tournament. For example, we consider the standard ordering of a transitive tournament $\mathbf{T}$ with 4 vertices to be $[0,1,2,3]$, but we could define a different ordering for a digraph $\mathbf{T}'$ as $[2,0,3,1]$. The difference is simple, for $\mathbf{T}$ the vertex labeled 0 is the source whereas for $\mathbf{T}'$ the vertex labeled 2 is the source. The vertex labeled 3 in $\mathbf{T}$ is the sink whereas for $\mathbf{T}'$ the vertex labeled 1 is the sink. In the following the standard ordering on n vertices will always be considered to be $[0,1,\ldots,n-2,n-1]$, where 0 is the source and $n-1$ is the sink with the other vertices ordered sequentially between them.

Example 6.3.3. The transitive tournament with the standard labeling on 4 vertices is defined as $\mathbf{T} = \langle\{0,1,2,3\},\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}\rangle$.

Another ordering we could have on this 4 vertex set is $[2,0,3,1]$ which would result in the digraph: $\mathbf{T}' = \langle\{0,1,2,3\},\{(2,0),(2,3),(2,1),(0,3),(0,1),(3,1)\}\rangle$



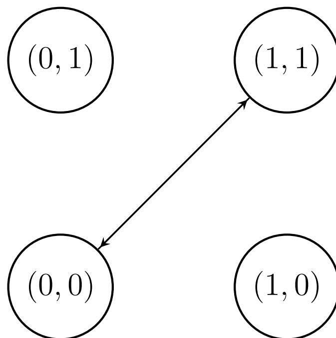
Two different orderings of the 4 vertex transitive tournament.

The problem we will be focusing on is this: for any $n,k>0$ if $\mathbf{T} = \langle\{0,\ldots,n-1\};E_1,\ldots,E_k\rangle$, where each E_i is a binary relation that defines a transitive tournament, then can we characterize

which sets of orderings allow for CSP($\mathbf{T}_L$) to be FO-definable and which sets of orderings will not?

To begin our search for such a characterization of this set of orderings we decided to write a digraph drawing and dismantling software in Java and run exhaustive dismantlings on all pairs of transitive tournaments with $|V(\mathbf{T})| \in \{2,3,4,5,6,7,8\}$. The source code for this program can be found in Appendix A. Our first experiment was to dismantle pairs of orderings. We paired the standard ordering on n vertices with all other possible orderings. The first observation from this experiment was that all pairs of relations where $n = 2, 3$ dismantled with each other. We can illustrate this using our dismantling algorithm explicitly.

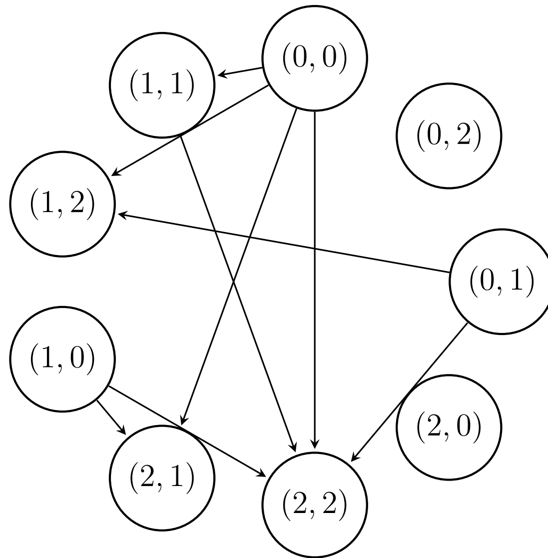
Example 6.3.4. Let $\mathbf{T} = \langle \{0,1\}; E_1, E_2 \rangle$ With $E_1 = \{(0,1)\}$ and $E_2 = \{(1,0)\}$. These are the only two orderings of $\{0,1\}$.

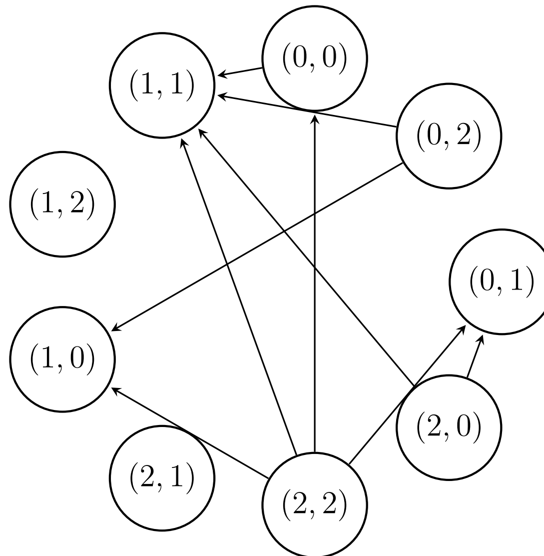


We can see that both E_1^2 and E_2^2 have $(0,1)$ and $(1,0)$ as isolated vertex and therefore both are dominated.Thus $\mathbf{T}^2$ dismantles using the dismantling sequence $S = ((0,1),(1,0))$.

Next we look at the slightly more interesting example of two orderings on T_3.

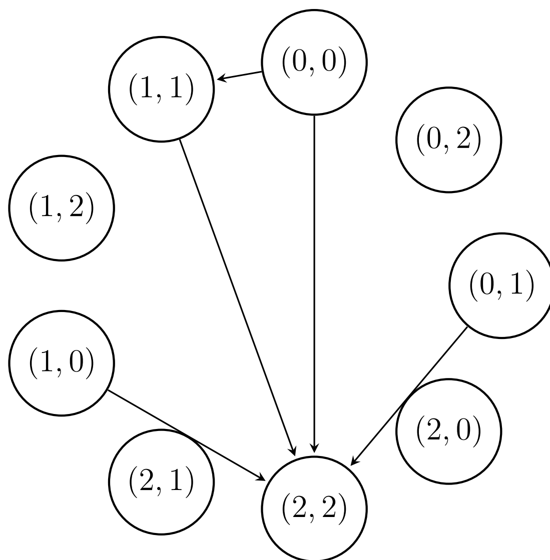
Example 6.3.5. Let $\mathbf{T} = \langle \{0,1,2\}; E_1, E_2 \rangle$ With $E_1 = \{(0,1),(0,2),(1,2)\}$ and $E_2 = \{(2,0),(2,1),(0,1)\}$. The relation E_1 corresponds to the standard ordering on 3 vertices and E_2 corresponds to the ordering $[2,0,1]$.



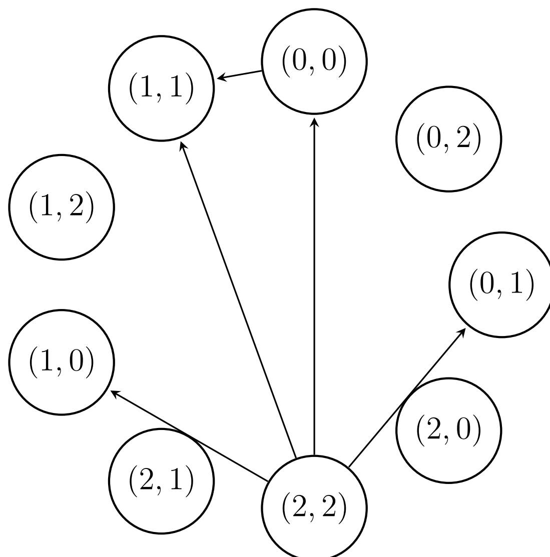


The structures E_1^2 and E_2^2.

For **T** to dismantle we need to find a mutual dismantling sequence for both E_1^2 and E_2^2. This can be performed in just two steps. Notice that $(0, 1)$ and $(1, 0)$ are dominated by $(0, 0)$ in E_1^2 and are dominated by $(1, 1)$ in E_2^2 thus we can not remove them yet. If we turn our attention to $(0, 2)$ and $(2, 0)$ we see that both are isolated in E_1^2 and both are dominated by $(2, 2)$ in E_2^2. Next, $(1, 2)$ and $(2, 1)$ are dominated by $(2, 2)$ in E_1^2 and they are isolated in E_2^2. Removing these vertices and their incident edges we are now left with:



The induced substructure with $E_1^2 \setminus \{(0,2),(2,0),(1,2),(2,1)\}$



The induced substructure with $E_2^2 \setminus \{(0,2),(2,0),(1,2),(2,1)\}$

Now we see that $(0,1)$ and $(1,0)$ are dominated by both $(0,0)$ and $(1,1)$ in both digraphs. We can now remove these last two vertices. We have now removed all 6 of the vertices in two steps of dismantling and we are left with just the diagonal elements, therefore CSP($\mathbf{T}_L$) is FO-definable.

From our exhaustive search we learned that all pairs of relations that define transitive tournaments on 3 vertices ($n = 3$) dismantle with each other. The more interesting cases appear when $n \geq 4$. In the case where $n = 4$ we find that there are 4 orderings that do not dismantle when paired with the relation that has the standard ordering $[0,1,2,3]$ (Figure 6.6). These "bad" pairs correspond to the orderings $[1,0,3,2]$, $[1,3,0,2]$, $[2,0,3,1]$, and $[2,3,0,1]$ (Figure 6.7). These pairings demonstrate an important point for our continued study. Specifically, if each relation in a set of

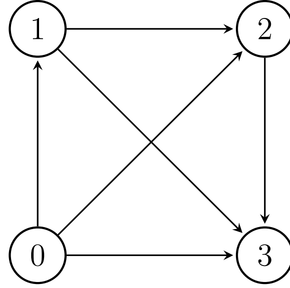


Figure 6.6: The standard ordering of the transitive tournament on 4 vertices.

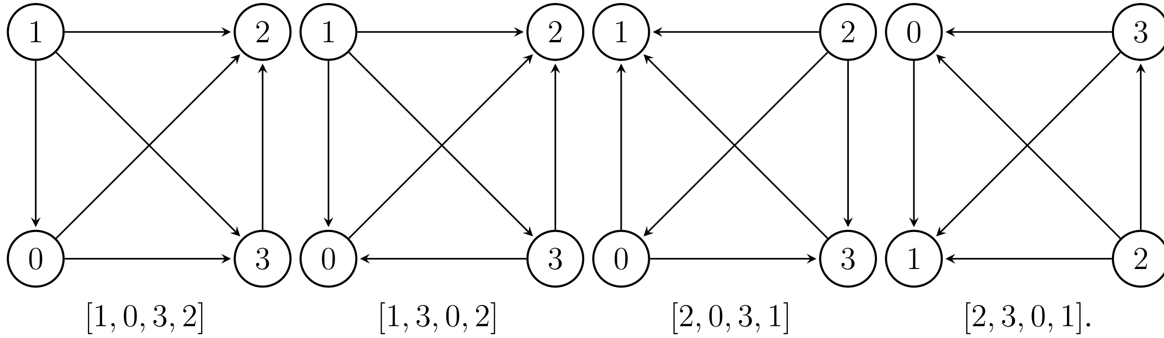


Figure 6.7: The 4 tournament orderings that do not dismantle when paired with the standard ordering

binary relations has a square that dismantles to the diagonal on its own we do not have that the larger structure formed by adding these relations to the relation set will also dismantle. This means that there must be more general obstructions than in the single digraph case when attempting to dismantle these larger structures. Reflecting on these "bad" orderings of $\mathbf{T}_4$ we noticed that none of these orderings share a source or a sink with the standard ordering. This observation turns out to be very important when trying to characterize which sets of transitive tournaments dismantle to the diagonal or not.

Definition 6.3.1. Let $[v_1, v_2, \ldots, v_{n-1}, v_n]$ be a total strict ordering such that $v_1 < v_2 < \cdots < v_{n-1} < v_n$. Then we call v_1 and v_n the ***extrema*** of the ordering.

Transitive tournaments are total strict orderings and we can speak of them interchangeably so it is natural for us to define of the extrema of these digraphs.

Definition 6.3.2. Let $\mathbf{T}_n = (\{0, \ldots, n-1\}, E)$ be a transitive tournament. Then the edge relations of $\mathbf{T}$ define a total strict order $[v_1, \ldots, v_n]$ with each distinct $v_i \in V(\mathbf{T}_n)$. We define the vertices $\underline{S}$ and $\bar{S}$ as the vertices such that for any $v_i \in V(\mathbf{T})$ we have $\underline{S} < v_i$ and $v_i < \bar{S}$. We call the set $\{\underline{S}, \bar{S}\}$ the ***extrema*** of the order defined by the transitive tournament $\mathbf{T}_n$.

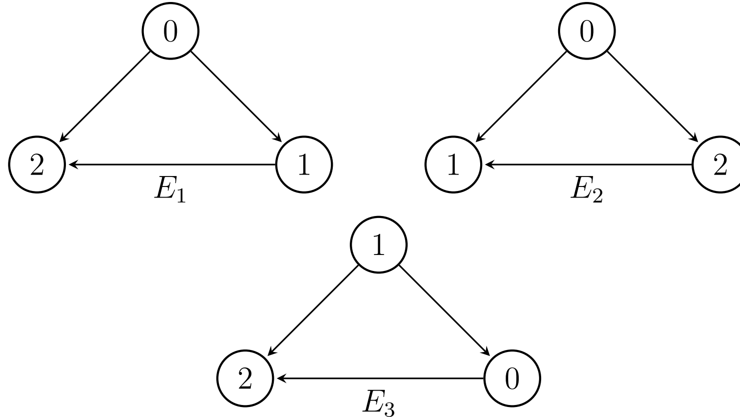


Figure 6.8: The square of the structure $\mathbf{T} = \langle\{0,1,2\}; E_1, E_2, E_3\rangle$ does not dismantle to the diagonal

Notice that for any transitive tournament the extrema are the sink and the source of the tournament.

Another interesting observation obtained from our exhaustive dismantlings was that no set of 3 transitive tournaments on 3 vertices with pairwise distinct extrema had squares that dismantled to the diagonal. Actually there were no dominated vertices in these structures at all.

Example 6.3.6. The square of the structure $\mathbf{T} = \langle\{0,1,2\}; E_1, E_2, E_3\rangle$, where E_1 is the standard ordering $[0,1,2]$, E_2 corresponds to the ordering $[0,2,1]$ and E_3 corresponds to the ordering $[1,0,2]$, does not dismantle to the diagonal. (Figure 6.8)

It turns out that there are no dominated vertices in this structure. This is because in order to be dominated in the total structure a pair must be dominated by the same pair in all relations. The simplest way to see that there are no dominated pairs is to look at the isolated vertices in each square. In E_1^2 the vertices $(0,2)$ and $(2,0)$ are isolated and therefore dominated by both $(0,0)$ and $(2,2)$, but in E_2^2 they are dominated by $(0,0)$ but not $(2,2)$ and in E_3^2 by $(2,2)$ but not $(0,0)$, thus they are not dominated by a common vertex in the total structure. Now $(0,1)$ and $(1,0)$ are isolated in E_2^2 and therefore dominated by both $(0,0)$ and $(1,1)$ but they are not dominated by $(1,1)$ in E_1^2 and not dominated by $(0,0)$ in E_3^2. Lastly, the vertices $(1,2)$ and $(2,1)$ are isolated in E_3^2 but are not dominated by $(1,1)$ in E_1^2 and not dominated by $(2,2)$ in E_2^2. Thus there is no vertex in $\mathbf{T}^2$ that can be removed and it does not dismantle to the diagonal.

Example 6.3.6 shows how restrictive our dismantling process becomes as we add more relations. The square of the single transitive tournament on 3 vertices dismantled to the diagonal in one step, when paired with a different ordering we saw it took at most two steps. Now we have found

a set of three orderings whose square does not dismantle at all. Again, looking at these orderings we see that they all share extrema pairwise but as a set there is no common extrema. From this observation we decided to look back at the dismantling of the square of individual transitive tournaments to see if we could account for this behaviour by studying their dismantling sequences. From this investigation we discovered that there is a simple way to construct dismantling sequences for transitive tournaments. Using this we can produce a simple algorithm for dismantling relational structures with lists which have many binary relations that define transitive tournaments.

The following propositions will help give us a better understanding of the strict order in which vertices can be dominated in the square of a transitive tournament.

Proposition 14. *Let $\boldsymbol{T} = (\{0, \dots, n-1\}, E)$ be a transitive tournament. A vertex $(x, y) \in V(\boldsymbol{T}_L^2)$ is the first pair to be dismantled if and only if $x \in \{\underline{S}, \bar{S}\}$ or $y \in \{\underline{S}, \bar{S}\}$ and $x \neq y$.*

Proof. First we show that if $x \notin \{\underline{S}, \bar{S}\}$ and $y \notin \{\underline{S}, \bar{S}\}$ then (x, y) is not dominated and can not be the first pair in the dismantling sequence. Without loss of generality let $x < y$. Since neither x nor y are extrema we have $\underline{S} < x < y < \bar{S}$. This means that (x, x) does not dominate (x, y) in $\mathbf{T}_L^2$ because $((\underline{S}, x), (x, y)) \in E^2$ but $((\underline{S}, x), (x, x)) \notin E^2$. Similarly, (y, y) does not dominate (x, y) in $\mathbf{T}_L^2$ because $((x, y), (y, \bar{S})) \in E^2$ but $((x, y), (y, y)) \notin E^2$.

Next we show that if $x \in \{\underline{S}, \bar{S}\}$ or $y \in \{\underline{S}, \bar{S}\}$ and $x \neq y$ then the vertex (x, y) is dominated. We will only prove the case where either x or y is a source because due to symmetry it is the same for when either x or y is a sink. First notice that $(v, \underline{S}) \notin E$ for all $v \in V(\mathbf{T})$ because $\underline{S}$ is a source. This means (x, y) is a source in T^2. Without loss of generality assume that $\underline{S} = x$ and $x < y$ then it is clear that for any $(w, z) \in V(\mathbf{T}^2)$ if $((x, y), (w, z)) \in E^2$ then $((\underline{S}, \underline{S}), (w, z)) \in E^2$. Thus (x, y) is dominated by $(x, x) = (\underline{S}, \underline{S})$ in $\mathbf{T}^2$.

□

Proposition 15. *Let $\boldsymbol{T}$ be a transitive tournament with n vertices and the ordering $[v_1, \dots, v_t, \dots, v_n]$. Let $\boldsymbol{H}$ be the induced substructure of $\boldsymbol{T}^2$ obtained by removing all non-diagonal pairs with a coordinate in $\{v_{t+1}, \dots, v_n\}$. Then any vertex (x, y) with v_t in one coordinate is dominated by (v_t, v_t) in $\boldsymbol{H}$. Similarly let $\boldsymbol{K}$ be the induced substructure of $\boldsymbol{T}^2$ obtained by removing all non-diagonal pairs with a coordinate in $\{v_{t+1}, \dots, v_n\}$. Then any vertex (x, y) with v_t in one coordinate is dominated by (v_t, v_t) in $\boldsymbol{K}$.*

Proof. Let $(x, y) \in V(\mathbf{H})$. Without loss of generality suppose $x = v_t$, $x \neq y$, and that $v_t < y$ (or else (x, y) would have already been removed). Since all pairs with a $v_1, \dots, v_{t-1}$ in only one coordinate have been removed the only in-edges, if any, of both (x, y) and (v_t, v_t) are the diagonal elements $(v_1, v_1), \dots, (v_{t-1}, v_{t-1})$ and since $x = v_t$ and $v_t < y$ we have that if $((x, y), (w, z))$ for

any $w, z \in V(\mathbf{T})$ then $((v_t, v_t), (w, z))$. Thus we have that (v_t, v_t) dominates (x, y) in $\mathbf{H}$. A similar argument can be made to show that if we had removed all pairs with $v_{t+1}, \ldots, v_n$ that (x, y) would be also be dominated by $(v_t, , v_t)$ in $\mathbf{K}$. □

We will now describe a special dismantling algorithm for relational structures comprised of many transitive tournaments.

Definition 6.3.3. Let $\mathbf{T} = \langle \{0, \ldots, n-1\}; E_1, \ldots, E_k \rangle$ be a relational structure with n vertices and where each E_i describes a transitive tournament with ordering $[v_1^i, \ldots, v_n^i]$. To begin we assign $\underline{S}_i = v_1^1$ and $\bar{S}_i = v_n^i$ for each $1 \leq i \leq k$ and call these vertices the ***extrema*** of E_i. We now describe our ***simplified dismantling for transitive tournaments***. We say that we can ***remove*** a vertex $x \in V(\mathbf{T})$ if $x \in \{\underline{S}_i, \bar{S}_i\}$ for all i, where $1 \leq i \leq k$. We remove the vertex x from each order E_i. We now reassign the extrema $\underline{S}_i$ and $\bar{S}_i$ for each order to reflect the removal of x. We now check to see if any vertices can be removed. We say a structure is ***dismantled simply*** if we can remove each of its vertices.

The dismantling process we described in Definition 6.1.3 and Theorem 6 for general digraphs involved moving to the square of its edge relations, but when we are working with transitive tournaments this simplified dismantling only relates the orderings of the tournaments and not their squares. We simply look for a common vertex x in the extrema of each ordering and if one is present we remove it from each ordering. We continue this process with the new orderings obtained by removing the common extrema x until we meet an obstruction or remove all $v \in V(\mathbf{T})$. We can now relate our simplified dismantling of these tournaments to the FO-definability of their CSPs.

Theorem 16. *Let $\boldsymbol{T} = \langle \{0, \ldots, n-1\}; E_1, \ldots, E_k \rangle$ be a relational structure where each E_i describes a transitive tournament with ordering $[v_1^i, \ldots, v_n^i]$ on $V(\boldsymbol{T})$. Then the following are equivalent:*

(1) *CSP($\boldsymbol{T}_L$) is* FO*-definable.*
(2) ***T*** *can be dismantled simply.*

Proof. (2) ⇒ (1)

Since $\mathbf{T}$ dismantles simply we know that there exists a sequence of vertices $\{s_1, \ldots, s_n\}$, $s_t \in V(\mathbf{T})$, which can be removed in the context of our simplified dismantling. We can easily construct a common dismantling sequence for each relation E_i. We begin this sequence with all vertices with s_1 in one coordinate we know these are removable by Proposition 15 since $s_1 \in \{v_1^i, v_n^i\}$

for $1 \leq i \leq k$. Next we add to our dismantling sequence each vertex with an s_2 in a coordinate (but not a s_1 in a coordinate because these have already been added) and so on. Since each s_t, $1 \leq t \leq n$, is always an extrema of each E_i Proposition 15 says that each of these vertices is commonly dominated by (s_t, s_t) at this point. We continue until we reach s_n and we have created a common dismantling sequence for each E_i and using this we can dismantle $\mathbf{T}^2$ to the diagonal.

(1) ⇒ (2)

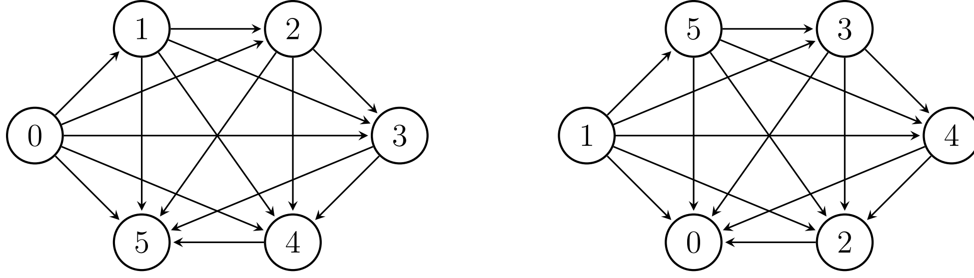
We will prove this by induction. Our hypothesis will be the statement $P(n)$ that if CSP($\mathbf{T}_L$) is FO-definable and $\mathbf{T}$ has n vertices then it admits a simplified dismantling. $P(1)$ is trivially true. We now assume $P(n-1)$, $n \geq 2$ and show $P(n)$. Since $\mathbf{T}_L^2$ dismantles to the diagonal, Proposition 14 says that each relation E_i must share a common extrema, which we will call S, or else there would be no vertex in which to start the dismantling. Consider the structure $\mathbf{T}' = \langle \{0, \ldots, n-1\} \setminus \{S\}; E_1', \ldots, E_k' \rangle$ where each E_i' is the restriction of E_i to $\{0, \ldots, n-1\} \setminus \{S\}$. Notice that each E_i' is still a transitive tournament. Now since CSP($\mathbf{T}_L$) is FO-definable we know that CSP($\mathbf{T}_L'$) is also FO-definable. Then by the induction hypothesis $\mathbf{T}'$ has a simplified dismantling sequence. Thus we can construct a simplified dismantling sequence for $\mathbf{T}$ by simply appending S to the beginning of the sequence for $\mathbf{T}'$.

□

Now that we have proven that our simplified dismantling algorithm can determine the FO-definability for relational structures with lists and many transitive tournaments we present some illustrative examples of its use.

Example 6.3.7. Let $\mathbf{T} = \langle \{0, 1, 2, 3, 4, 5\}; E_1, E_2 \rangle$ where E_1 is the transitive tournament with the standard ordering and E_2 is the transitive tournament ordered $[1, 5, 3, 4, 2, 0]$. Then $\mathbf{T}_L^2$ dismantles to the diagonal.

We start by comparing the sinks and sources. The extrema for E_1 are 0 and 5 and for E_2 we have 1 and 0. Thus the algorithm says that we we can begin by removing the common 0 element from both. This leaves us with $[1, 2, 3, 4, 5]$ for E_1 and $[1, 5, 3, 4, 2]$ for E_2. Now we compare extrema again. For E_1 we have 1 and 5 and for E_2 we have 1 and 2. So we remove common vertex 1 and we are left with $[2, 3, 4, 5]$ and $[5, 3, 4, 2]$. Next we remove the common 5 and that leaves us with $[2, 3, 4]$ and $[3, 4, 2]$. We can now remove 2 leaving us with $[3, 4]$ for both orderings and clearly we can remove each of these. Since we have successfully removed all the vertices in both relations of $\mathbf{T}$ then by Theorem 16 we have that CSP($\mathbf{T}_L$) is FO with lists.

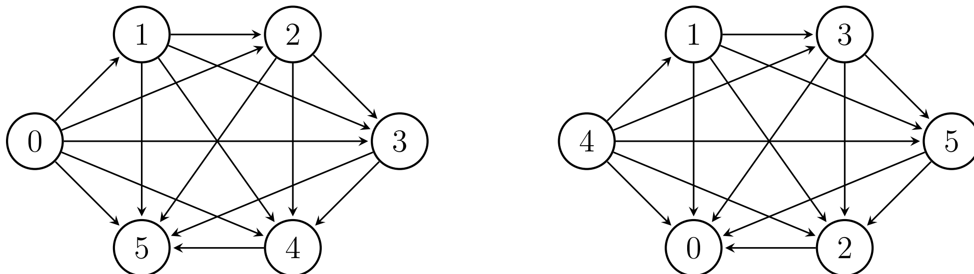


These two orderings on $\mathbf{T}_6$ share a dismantling sequence.

Example 6.3.7 provided us with a simplified dismantling sequence for each of our transitive tournaments E_1 and E_2 which corresponds to $S = \{0, 1, 5, 2, 3, 4\}$. The sequence S, consisting of vertices from $\mathbf{T}$, can be easily translated into a dismantling sequence for $\mathbf{T}_L^2$. Let S_i correspond to the ith element in the simplified dismantling sequence then to construct a dismantling sequence of $\mathbf{T}_L^2$ we begin by adding all vertices of the form (S_1, v) and (v, S_1) for all $v \in V(\mathbf{T})$ in any order. We then add all vertices of the form (S_2, v) and (v, S_2) for all $v \in V(\mathbf{T})$ that have not already been added to the sequence and continue this up to $S_6 = 4$. Note that the order of each (S_i, v) and (v, S_i) in our sequence for a particular vertex S_i does not matter because as long as each (S_{i-1}, v) and (v, S_{i-1}) has already been removed then each (S_i, v) and (v, S_i) are equally dominated. Using this sequence of pairs we can dismantle $\mathbf{T}_L^2$ to the diagonal. Now let us turn our attention to an example that does not dismantle.

Example 6.3.8. Let $\mathbf{T} = \langle\{0, 1, 2, 3, 4, 5\}; E_1, E_2\rangle$ where E_1 is the transitive tournament with the standard ordering and E_2 is the transitive tournament ordered $[4, 1, 3, 5, 2, 0]$. Then $\mathbf{T}_L^2$ does not dismantle to the diagonal.

In the first step we see that the extrema are 0 and 5 for E_1 and 4 and 0 for E_2. Again we begin by removing 0. We are now left with $[1, 2, 3, 4, 5]$ and $[4, 1, 3, 5, 2]$. At this point the two orderings do not share any extrema and therefore no simplified dismantling sequence exists for $\mathbf{T}$ and by Theorem 16 CSP($\mathbf{T}_L$) is not FO-definable.



We can easily construct relational structures comprised of sets of transitive tournaments which either dismantle or not by using the reverse of the algorithm found in Definition 16. If we are considering only sets of orderings which can be dismantled simply then we see that there are limits to the the largest sets of unique orderings based on the amount of vertices n. If we fix any particular simple dismantling sequence, a sequence constructed from Definition 16, then we see that there are only a finite number of orderings that also can be dismantled using that same sequence.

Consider the standard ordering on $\mathbf{T}_n$, and the simple dismantling sequence $S = \{0, 1, \ldots, n-2, n-1\}$. Using the strict rules of the simplified dismantling we can use this sequence to construct each of the unique orderings that dismantle simply using the same sequence. This is because in the simplified dismantling we are always removing a common extrema from each order, which is either a bottom or top element of a particular ordering. Using our sequence S we start with the vertex 0. It must be placed at an extrema of the ordering, the first element or the last, since it is removed first. We will fix 0 as the source for all these orderings and note that the reverse of any order we construct will also dismantle using the same sequence S. Thus with 0 fixed as the source we only have 2 choices for where to place the vertex 1. We can either put it in the second value of the ordering or as the sink in order (the n-1th value) to ensure it is removable at the second step and satisfies our dismantling sequence. This is actually the case for each vertex $v \in \{1, \ldots . n-2\}$. We always have two choices of where we can place these vertices and we only have 1 choice for the vertices 0 and $n-1$. Thus combinatorially we see that we have 2^{n-2} unique orderings that satisfy any particular dismantling sequence of transitive tournaments on n vertices with 0 fixed as the source. Since the reverse of these orders will also dismantle simple the total becomes 2^{n-1}.

Example 6.3.9. If we fix a dismantling sequence for each structure $\mathbf{T}$ with n vertices and whose relations define transitive tournaments then we can determine the largest sets of unique orderings that dismantle to the diagonal. For $n = 2$ we have that this set can have at most $2^{2-1} = 2^1 = 2$ orderings. This corresponds to the only two orderings of $\mathbf{T}_2$ $[1, 2]$ and $[2, 1]$. For $n = 3$ we have $2^2 = 4$ and only pairs and their reverse orderings dismantle. For example the orderings $[0, 1, 2]$, $[0, 2, 1]$, $[2, 1, 0]$ and $[1, 2, 0]$ are the only orderings that dismantle using $S = \{0, 1, 2\}$. This actually verifies our results from Example 6.3.6. Any set of unique orderings on 3 vertices of size 3 or more must contain a reverse ordering of one of the orderings if it is to dismantle. For $n = 4$ we have that there are sets of $2^3 = 8$ unique orderings and if we include the standard ordering these correspond to $[0, 1, 2, 3]$, $[0, 2, 3, 1]$, $[0, 3, 2, 1]$, $[0, 1, 3, 2]$ and their revere orderings.

6.4 A classification of relational structures with lists and n binary relations whose CSPs are FO-definable

In Section 6.2 we presented a characterization for which digraphs with lists have CSPs which are FO-definable. These results motivated us to ask if we could extend this characterization to a general relational structure which has k relations that are all binary. In this section we present a general theorem and classification for determining exactly this.

In terms of the work we have already done our new task is to dismantle the squares of many digraphs at once. We will be using tools developed for the single digraph case and extending them to help us understand these larger structures. We do not have to make too large of a logical leap since domination and dismantling are defined for all relational structures. The primary consideration with these larger structures is that each of the squares must share a mutual dismantling sequence. This means that each square of a digraph needs to be dismantled in the same order and each dominated pair needs to be dominated by the same pair in each square. We could easily construct a larger structure by adding to its relation set multiple copies of the same digraph and we could still use Theorem 12 to show if this structure dismantles or not. This of course would add nothing new to our understanding. We are interested in those relational structures with multiple unique digraphs as relations and we would like to characterize which of these structures with lists have CSPs that are FO-definable.

As we move to these larger structures with multiple binary relations a simple and interesting starting point is to construct a relation set with two relations. One relation, E, defining a digraph and the other being what we will call its flip, E_F.

Definition 6.4.1. Let $\mathbf{G} = (V, E)$ be a digraph. We define E_F as the edge relation such that $(y, x) \in E_F$ whenever $(x, y) \in E$. We call the relation E_F the ***flip*** of E and the digraph $\mathbf{G}_F = (V, E_F)$ the ***flip*** of $\mathbf{G}$.

The following observation will provide us with a powerful tool when working with these larger binary structures.

Lemma 17. *Let $\mathbf{G} = (V, E)$ be a digraph with lists and let $\mathbf{G}_F = \langle V; E, E_F\rangle$ be the relational structure where E is a binary relation and E_F is its flip. Then CSP($\mathbf{G}_L$) is FO-definable if and only if CSP($(\mathbf{G}_F)_L$) is FO-definable.*

Proof. We will prove this lemma using ideas from first order logic directly. One direction is obvious. If $(\mathbf{G}_F)_L$ satisfies a first order sentence ϕ then so does $\mathbf{G}_L$

The other direction is more involved. We want to show that if $\mathbf{G}_L$ satisfies a first order sentence ϕ then we can create a different first order sentence ϕ_F that $(\mathbf{G}_F)_L$ satisfies. Let $\mathbf{G} = (V, E)$ be a digraph such that CSP($\mathbf{G}_L$) is FO-definable. This means that there exists an existential positive sentence ϕ with atomic subformulas of the form "$(x, y) \in \alpha$" such that a structure $\mathbf{H} = \langle H; \alpha \rangle$ satisfies ϕ if and only if there is no homomorphism $h : \mathbf{H}_L \to \mathbf{G}_L$. We will create the first order sentence ϕ_F from the sentence ϕ by replacing each of these atomic formulas with "$(x, y) \in \alpha \vee (y, x) \in \beta$".

Claim: Let $\mathbf{H}_F = \langle H; \alpha, \beta \rangle$ be a relational structure with α, β binary relations. Then $\mathbf{H}_F$ satisfies ϕ_F if and only if there is no homomorphism $h_F : (\mathbf{H}_F)_L \to (\mathbf{G}_F)_L$.

Proof. We will show that the following holds and use it to prove our claim: If $\mathbf{K} = \langle H; \alpha \cup \beta_F \rangle$, then $(\mathbf{H}_F)_L \to (\mathbf{G}_F)_L$ if and only if $\mathbf{K}_L \to \mathbf{G}_L$. Suppose $h : (\mathbf{H}_F)_L \to (\mathbf{G}_F)_L$ then for any $a_1, a_2, b_1, b_2 \in H$ such that $(a_1, a_2) \in \alpha$ and $(b_1, b_2) \in \beta$ then we have $(b_2, b_1) \in \beta_F$. Since h is a homomorphism we have $(h(a_1), h(a_2)) \in E$ and $(h(b_1), h(b_2)) \in E_F$ which implies $(h(b_2), h(b_1)) \in E$. Thus, $h : \mathbf{K}_L \to \mathbf{G}_L$. Now suppose $f : \mathbf{K}_L \to \mathbf{G}_L$ and let $a_1, a_2, b_1, b_2 \in H$ such that $(a_1, a_2) \in \alpha$ and $(b_2, b_1) \in \beta_F$ then $(a_1, a_2), (b_2, b_1) \subseteq \alpha \cup \beta_F$ then $(f(b_2), f(b_1)) \in E$ which means $(f(b_1), f(b_2)) \in E_F$. So $f : (\mathbf{H}_F)_L \to (\mathbf{G}_F)_L$.

This implies that there is no homomorphism $h_F : (\mathbf{H}_F)_L \to (\mathbf{G}_F)_L$ if and only if $\mathbf{K}_L$ satisfies ϕ. $\mathbf{K}_L$ satisfies ϕ if and only if we have elements of $\mathbf{K}_L$ that satisfy all atomic formula "$(x, y) \in \alpha \cup \beta_F$", but this is equivalent to saying that we can find elements in $(\mathbf{H}_F)_L$ thats satisfy each "$(x, y) \in \alpha \vee (y, x) \in \beta$". This means that $(\mathbf{H}_F)_L$ satisfies ϕ_F. □

□

This turns out to be a very useful and interesting observation. We could have also proved Lemma 17 by showing that both structures actually have the same dismantling sequence of their squares. The idea of these flip relations gives rise to a new structure which we define now.

Definition 6.4.2. Let $\mathbf{G} = (V, E)$ be a relational structure such that $E = \{E_1, E_2, \ldots, E_n\}$ is a list of binary relations. If for every $E_i \in E$ we have $E_{i_F} \in E$, where E_{i_F} is the flip of E_i, then we call the structure $\mathbf{G}$ ***flip closed***.

Since adding the flip of a relation does not affect a structure's property to dismantle we can now build flip closed structures by adding the flip of any edge relation that is missing from a structure's set of relations. Using the idea of ***flip closure*** we can generalize Lemma 17.

Lemma 18. *Let* $\mathbf{G}$ *be a relational structure with binary relations and let* $\mathbf{G}_F$ *be the flip closure of* $\mathbf{G}$ *then CSP(*$\mathbf{G}_L$*) is* FO-*definable if and only if CSP(*$(\mathbf{G}_F)_L$*) is* FO-*definable.*

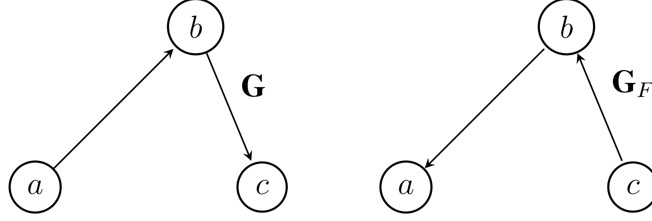


Figure 6.9: $\mathbf{G}_F$ is the flip of $\mathbf{G}$.

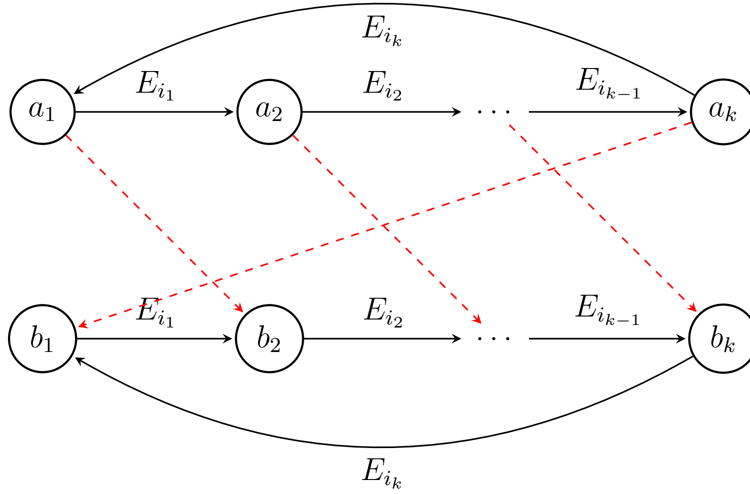


Figure 6.10: The generalized impeding bicycle

Proof. The proof of this is simple. By the comment following Lemma 17 we know that the dismantling sequence for $\mathbf{G}_L^2$ will also work for $(\mathbf{G}_F)_L^2$. The dismantling sequence for $(\mathbf{G}_F)_L^2$ also clearly works for $\mathbf{G}_L^2$ because we have $E(\mathbf{G}_L) \subseteq E((\mathbf{G}_F)_L)$. $\square$

We can close the relation set of a relational structure with lists with all of its flipped binary relations and not affect its dismantling. Thus for the remainder of this section we will consider all of our relational structures to be flip closed. Before we can give our full characterization of these structures we need to define the extension of the impeding bicycle from the previous section.

Definition 6.4.3. Let $\mathbf{G} = \langle V; E_1, E_2, \ldots, E_n \rangle$ be a relational structure where each E_i is a binary relation. Then $\mathbf{G}$ has a ***generalized impeding bicycle*** if there exists vertices $a_1, a_2 \ldots, a_k, b_1, b_2, \ldots, b_k \in V$ and indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that for all t, $1 \leq t \leq k$, we have $(a_t, a_{t+1}), (b_t, b_{t+1}) \in E_{i_t}$ and $(a_t, b_{t+1}) \notin E_{i_t}$. (where the indices are considered modulo $k+1$) (Figure 6.10)

Notice that this configuration is the same as the impeding bicycle from Definition 6.2 when we are working with a single digraph and each of the relations, E_i, are the same.

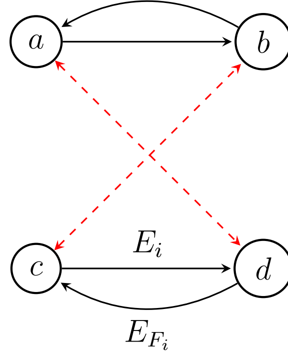


Figure 6.11: Generalized impeding bicycle formed by separated edges in a flip closed structure.

Proposition 19. *Let $\mathbf{G} = \langle V; E_1, \ldots, E_n \rangle$ be a flip closed relational structure such that each E_i is a binary relation. If any relation E_i defines a digraph on V that is not telescopic then $\mathbf{G}$ contains a generalized impeding bicycle.*

Proof. Let $\mathbf{G}_{E_i} = (V, E_i)$ be a digraph which is not telescopic. Thus $\mathbf{G}_{E_i}$ has separated edges. Then we have $(a, b), (c, d) \in E_i$ and $(a, c), (b, d) \notin E_i$. Since $\mathbf{G}_L$ is flip closed $E_{i_F} \in E$. Then we have the bicycle with the top formed by $(a, b) \in E_i$ and $(b, a) \in E_{i_F}$ and the bottom formed by $(c, d) \in E_i$ and $(d, c) \in E_{i_F}$. □

By Proposition 19 if we have one binary relation in our flip closed relational structure that is not telescopic then we can always find an impeding bicycle. This can be seen in Figure 6.11. This result allows us to remove the necessary condition that the relations be telescopic as we extend Theorem 12 to our generalized theorem.

Theorem 20. *Let $\mathbf{G} = \langle V; E_1, \ldots, E_n \rangle$ be a flip closed relational structure where each E_i is a binary relation. Then the following are equivalent:*

(1) *CSP($\mathbf{G}_L$) is* FO*-definable.*
(2) *$\mathbf{G}$ does not contain a generalized impeding bicycle.*

Proof. (1) ⇒ (2)

We will suppose that $\mathbf{G}$ has a generalized impeding bicycle and arrive at a contradiction.

Now because CSP($\mathbf{G}_L$) is FO-definable there exists a dismantling sequence from $\mathbf{G}_L^2$ to the diagonal. Without loss of generality there must be a vertex (a_t, b_t) from the generalized impeding

bicycle that appears first in the dismantling sequence. We dismantle $\mathbf{G}_L^2$ to an induced substructure $\mathbf{H}$ and let E'_i be the restriction of each E_i to $\mathbf{H}$. The vertex (a_t, b_t) must be dominated by either (a_t, a_t) or (b_t, b_t) in each E'_{i_t}.

Notice that $(a_{t-1}, a_t), (b_{t-1}, b_t) \in E'_{i_t}$ but $(a_{t-1}, b_t) \notin E'_{i_t}$. Therefore $((a_{t-1}, b_{t-1}), (a_t, b_t)) \in {E'_{i_t}}^2$ but $((a_{t-1}, b_{t-1}), (b_t, b_t)) \notin {E'_{i_t}}^2$. Meaning that (a_t, b_t) cannot be dominated by (b_t, b_t) in ${E'_{i_t}}^2$.

Now if we turn our attention to $E'_{i_{t+1}}$ we have that $(a_t, a_{t+1}), (b_t, b_{t+1}) \in E'_{i_{t+1}}$ but $(a_t, b_{t+1}) \notin E'_{i_{t+1}}$. Therefore $((a_t, b_t), (a_{t+1}, b_{t+1})) \in {E'_{i_{t+1}}}^2$ but $((a_t, a_t), (a_{t+1}, b_{t+1})) \notin {E'_{i_{t+1}}}^2$. Meaning that (a_t, b_t) cannot be dominated by (a_t, a_t) in ${E'_{i_{t+1}}}^2$. Thus, (a_t, b_t) is not dominated and we have a contradiction.

Next we will show that we can find an impeding bicycle in any flip closed $\mathbf{G}_L$ such that CSP($\mathbf{G}_L$) is not FO-definable.

$(2) \Rightarrow (1)$

Assume CSP($\mathbf{G}_L$) is not FO-definable. This implies that $\mathbf{G}_L^2$ does not dismantle to the diagonal. Therefore we have that $\mathbf{G}_L^2$ dismantles to some structure $\mathbf{H}$ where $\mathbf{H}$ contains the diagonal properly and no vertex of $\mathbf{H}$ is dominated. Let E'_i be the restriction of each E_i to $\mathbf{H}$. We will use the following proposition in order to construct our generalized impeding bicycle.

Proposition 21. *For every vertex $(a, b) \in V(\boldsymbol{H})$ there exists a relation E'_i and a vertex $(c, d) \in V(\boldsymbol{H})$ such that $((a, b), (c, d)) \in E_i'^2$ and $((a, a), (c, d)) \notin E_i'^2$.*

Proof. We will suppose that no such relation exists and reach a contradiction. If no such relation exists then for all E'_i, $1 \leq i \leq k$ and for all $(c, d) \in V(\mathbf{H})$, we have that if $((a, b), (c, d)) \in E_i'^2$ then $((a, a), (c, d)) \in E_i'^2$; since $\mathbf{G}$ is flip-closed we have that ${E'_{i_F}}^2 \in E(\mathbf{H})$ and that $((a, b), (c, d)) \in {E'_{i_F}}^2$ and $((a, a), (c, d)) \in {E'_{i_F}}^2$ but this implies that (a, a) dominates (a, b). This is a contradiction because we have assumed that there are no dominated vertices in $\mathbf{H}$. □

Now to show that $\mathbf{H}$ has a generalized impeding bicycle we construct a new digraph $\mathbf{D} = (V, R)$ with $V(\mathbf{D}) = V(\mathbf{H})$ and with the edge relation R such that $((a_t, b_t), (a_{t+1}, b_{t+1})) \in R$ if it satisfies the property that there exists a relation E_i such that $((a_t, b_t), (a_{t+1}, b_{t+1})) \in E_i^2$ and $((a_t, a_t), (a_{t+1}, b_{t+1})) \notin E_i^2$. Proposition 21 states that every vertex of $\mathbf{D}$ will have an out-degree of at least 1. Now since the out-degree of every vertex of our new digraph $\mathbf{D}$ is at least 1 we know that it contains a directed cycle of some length n. Let $C = \{(a_1, b_1), (a_2, b_2), \ldots, (a_{n-1}, b_{n-1}),$

$(a_n, b_n)\}$ be this directed cycle. If we consider the meaning of this cycle C in $\mathbf{D}$ in terms of $\mathbf{G}_L^2$ we see that this implies there are two sequences in $\mathbf{G}_L$, $C_1 = \{a_1, a_2, \ldots . a_n\}$ and $C_2 = \{b_1, b_2, \ldots . b_n\}$, such that each $(a_t, a_{t+1}), (b_t, b_{t+1}) \in E_i$ and $(a_t, b_{t+1}) \notin E_i$ for some relation E_i. This is exactly the definition of our generalized impeding bicycle. Thus $\mathbf{G}_L$ contains a generalized impeding bicycle.

□

We have discovered a combinatorial characteristic which we can use to classify the CSPs of flip closed relational structures with lists and k binary relations. This is the digraph-like structure we have named a generalized impeding bicycle. We now show that the problem of detection for this structure is tractable and that it is in NL.

Definition 6.4.4. We can describe the decision problem for detecting the impeding bicycle as follows:

Problem : *GENERALIZED BICYCLE*

Input: A relational structure $\mathbf{G} = \langle V, E \rangle$

Output: True if $\mathbf{G}$ contains a generalized impeding bicycle and false if $\mathbf{G}$ does not.

Lemma 22. *The decision problem GENERALIZED BICYCLE is in* NL.

Proof. We describe the algorithm for which a nondeterministic Turing machine could perform in order to determine if a generalized impeding bicycle is present in a given relational structure in terms of the space needed.

We have a input relational structure $\mathbf{G} = \langle V; E \rangle$ where E is a sequence of binary edge relations. We only need to remember the starting pair, a previous pair, that meets the conditions of our bicycle, and the guess pair. This amounts to writing only 6 vertex indices in memory. $|V| = n$, then total pairs $= (n)(n-1)/2$. We describe the algorithm with the following pseudo code:

```
start_pair = (a_1,b_1)
prev_pair = start_pair
count = (n)*(n-1)/2

loop while count > 0
```

```
{

  current_pair = (a_2,b_2) (correctly guess the next pair in the
     impeding bicycle)

  if there exists an E_i such that
     (prev_pair_1,current_pair_1), (prev_pair_1, current_pair_2)
     in E_i and (prev_pair_1,current_pair_2) not in E_i then

    if (current_pair = start_pair) then {
      accept
    }else
      prev_pair = current_pair
      count = count - 1
    }
  else{
    reject
  }
}
reject (We have gone through each vertex without an accept)
```

□

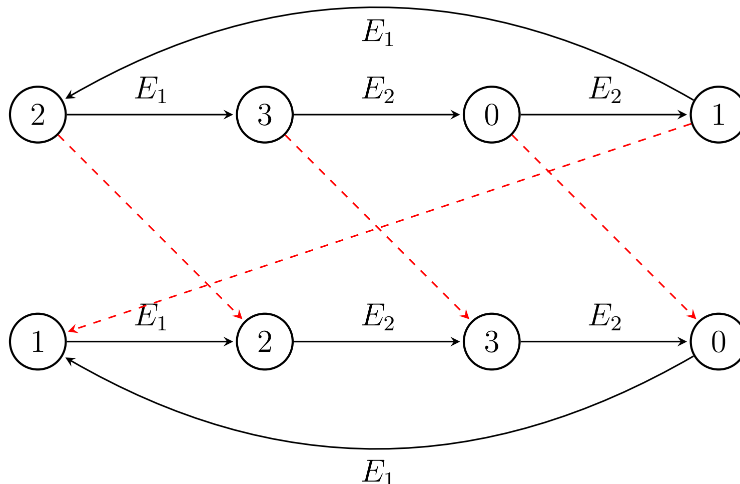
Thus we have found a universal obstruction to the dismantling of the square of these relational structures with many binary relations and lists. The generalized impeding bicycle is the universal combinatorial structure whose presence determines if the CSP of these structures are FO-definable or not. We have also shown that the problem of detecting these impeding bicycles is a tractable problem meaning that Theorem 20 is a reasonable tool for determining the descriptive complexity of these structures.

6.5 Observations and examples

We saw that the problem of detecting a generalized impeding bicycle is in NL but it can still be difficult to use in a practical sense and we think it best to illustrate these bicycles with some examples. We have already encountered some larger relational structures that do not dismantle to the diagonal and therefore are not FO-definable. By the theorem we should be able to find a generalized impeding bicycle lurking within its relations. We start with the following example of

transitive tournaments on 4 vertices.

Example 6.5.1. Let $\mathbf{T} = \langle \{0, 1, 2, 3\}; E_1, E_2 \rangle$, where E_1 defines the standard ordered tournament on 4 vertices and E_2 defines a tournament with the ordering $[2, 3, 0, 1]$. We know from Section 6.3 that $\mathbf{T}_L^2$ does not dismantle to the diagonal because the two orderings do not share a sink or a source. Theorem 20 says that there must exist an impeding bicycle. With some investigation the following obstruction can be found:



The impeding bicycle found in the structure $\mathbf{T}$.

In the above example we did not need to use the flip closure in order to find our generalized impeding bicycle. If we look back at the structure $\mathbf{T}$ in Example 6.3.6 we actually see that we can not find an impeding bicycle as defined in Definition 6.4.3 but we can find one in its flip closure $\mathbf{T}_F$.

Example 6.5.2. Let $\mathbf{T} = \langle \{0, 1, 2\}; E_1, E_2, E_3 \rangle$, where E_1 defines the standard ordering, E_2 defines a transitive tournament with the ordering $[0, 2, 1]$, and E_3 defines a transitive tournament with ordering $[1, 0, 2]$. We saw in Example 6.3.6 that $\mathbf{T}_L^2$ does not dismantle to the diagonal. Thus we can find an impeding bicycle in its flip closure $\mathbf{T}_F = \langle \{0, 1, 2\}; E_1, E_{1_F}, E_2, E_{2_F}, E_3, E_{3_F} \rangle$. This obstruction can be found in Figure 6.12.

Example 6.5.2 illustrates that we could have defined our obstruction for these larger structures differently. If we concern ourselves with just $\mathbf{T}$, instead of its flip closure $\mathbf{T}_F$, we can construct a slightly different obstruction which can be seen in Figure 6.13. This obstruction contains the same information as the bicycle in Figure 6.12. Mainly, that there exists a relationship between the vertices $(1, 0), (2, 1), (0, 2) \in V(\mathbf{T}^2)$ such that none of these vertices can be dismantled until one of the others is removed. The difference being that the bicycle in Figure 6.12 has a consistency in the direction of its edges for each segment where as the bicycle in Figure 6.13 does not. Of

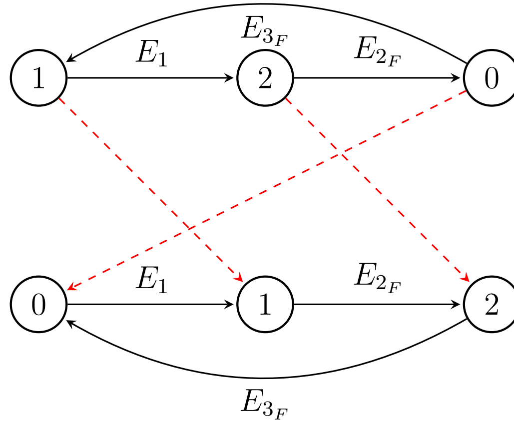


Figure 6.12: The impeding bicycle found in the flip closed structure $\mathbf{T}_F$ from Example 6.5.2.

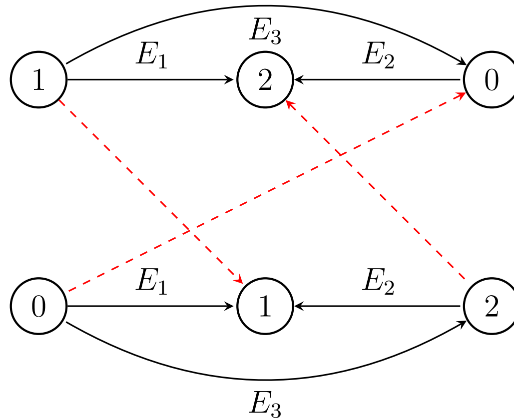


Figure 6.13: An obstruction found in the structure $\mathbf{T}$ from Example 6.5.2.

course finding an obstruction like the one in $\mathbf{T}$ implies there is a generalized impeding bicycle as in Definition 6.10 in $\mathbf{T}_F$. This is the equivalent of showing that the bicycle in $\mathbf{T}$ (Figure 6.13) implies the bicycle in $\mathbf{T}_F$ (Figure 6.12). This provides us with a good example of, and insight into, Proposition 18.

So far we have only been able to determine whether a particular structure has a CSP which is either FO-definable or not. A natural question, and a source for future study, is: "If these CSPs are not FO-definable then what is their complexity?" We will turn our attention back to the structure in Example 6.5.1 and show that interestingly its CSP is NL-complete.

Theorem 23. *Let* $\mathbf{T} = \langle\{0,1,2\}; E_1, E_2, E_3\rangle$ *where* E_1 *defines the standard ordering,* E_2 *defines a transitive tournament with the ordering* $[0,2,1]$*, and* E_3 *defines a transitive tournament with ordering* $[1,0,2]$*. Then CSP(*$\mathbf{T}_L$*) is* NL*-complete.*

Proof. We first show that the problem is in NL; by [17] it is sufficient to show that $\mathbf{T}_L$ admits a ***majority polymorphism***, i.e. that there exists a map $m : \{0,1,2\}^3 \to \{0,1,2\}$ satisfying the majority identities

$$m(x,x,y) = m(x,y,x) = m(y,x,x) = x$$

for all x, y, and that preserves every relation of $\mathbf{T}_L$, i.e. $m(x,y,z) \in \{x,y,z\}$ for all x,y,z, and $(m(x,y,z), m(x',y',z')) \in E_i$ whenever $(x,x'), (y,y'), (z,z') \in E_i$, for $1 \leq i \leq 3$.

Claim: Let m be an operation on $\{0,1,2\}$ satisfying the majority identities. Then m preserves any transitive tournament.

Proof. Indeed, it suffices to prove the result for the tournament with the standard ordering $[0,1,2]$. Suppose that $(x,x'), (y,y'), (z,z') \in E$; then $\{x,y,z\} \subseteq \{0,1\}$ and $\{x',y',z'\} \subseteq \{1,2\}$. Hence $(m(x,y,z), m(x',y',z')) \in E$ unless $m(x,y,z) = m(x',y',z') = 1$. But since m is majority, this implies that at least 2 entries of both $\{x,y,z\}$ and $\{x',y',z'\}$ are equal to 1, and thus one of the three pairs $(x,x'), (y,y'), (z,z')$ must be $(1,1)$, a contradiction. □

By the claim, it follows that $\mathbf{T}_L$ admits a majority polymorphism (in fact, several).

Secondly, we must show that CSP($\mathbf{T}_L$) is NL-hard; following results from [18] it will suffice to prove the following claim:

Claim: There exists a pp-definition from $\mathbf{T}_L$ of the order $\{(0,0),(0,1),(1,1)\}$.

Proof. In fact,

$$\{(x,y) \in \{0,1\}^2 : \exists u \, \exists v \, (x,u) \in E_1, \; (v,u) \in E_2, \; (y,v) \in E_3\} = \{(0,0),(0,1),(1,1)\}$$

Notice that we may restrict to the set $\{0,1\}$ as all lists are unary relations of the structures. For $x = 0 = y$ take $u = 1$, $v = 2$ and for $x \in \{0,1\}$ and $y = 1$ take $u = 2$, $v = 0$. On the other hand if $x = 1$ then it forces $u = 2$, which forces $v = 0$, which forces $y = 1$, and thus $(1,0)$ is not in the relation. □

□

Chapter 7

Conclusion

7.1 Contribution

Throughout this work we have investigated the complexity of constraint satisfaction problems from a combinatorial point of view. Motivated by the foundational research of Larose, Tardif and Loten [5] and the use of relational domination we aimed to further the classification results for constraint satisfaction problems described by digraphs with lists that are first order definable. Theorem 20 extended the results found in [6] by defining an obstruction that characterized those structures consisting of several digraphs with lists whose CSP is not FO-definable.

We designed software routines to facilitate the dismantling of sets of many digraphs. With this software we conducted the exhaustive dismantlings of sets of well known graph structures. From the results and observations this process provided we were able to postulate and then prove a classification result for the special case of transitive tournaments. We also presented an algorithm for checking the FO-definability of these structures. The simplicity of this algorithm gives one a very easy way to construct large structures that are guaranteed to be FO-definable without needing to consider the properties of that structure's square. Using the restrictive nature of the simplified dismantling sequence of a transitive tournament (Definition 16) and by fixing a simple dismantling sequence we found that we could produce an upper bound on the size of the set of unique transitive tournaments that can dismantle with each other in terms of the number of vertex n. For these structures with lists and n vertices we used a counting method to show that the upper bound on the size of their relational set is 2^{n-1}.

With these tools in hand we found structures with many binary relations which did not dismantle

to the diagonal even though each of its relations dismantled individually. This implied that there existed inter-relational obstructions and motivated us to search for a generalization of Theorem 12. We were successful in this. Our classification, Theorem 20, relied on the non-existence of a cycle-like structure which we named a generalized impeding bicycle (Definition 6.10). We saw that this is the only obstruction needed to characterize the first order definability of flip closed relational structures with lists and binary relations. Finally, we concluded by presenting some interesting examples that integrated many of our results in a concise and demonstrative way.

7.2 Outlook

During the process of writing this thesis it was announced that two proofs of the general dichotomy conjecture of Feder and Vardi had been found [19] [20]. This is a great advancement in our field and although this conjecture was a great motivator, the classification of CSPs is far from complete. There are still several dichotomies within the subclasses of P which remain unresolved. Thus, there are still open questions for future research of constraint satisfaction problems such as those belonging to classes such L or NL [21]. Direct extensions to the research carried out in this thesis could include discovering dismantling obstructions and other properties for structures with only ternary (or larger) relations or the formulation of a general combinatorial characterization of all FO-definable structures with lists. The dismantling software, whose source code is provided in Appendix A, could be modified to add new functionality to facilitate the dismantling of relations that have arities greater than 2. With these new functionalities in hand we could investigate and hopefully discover the properties of these structures that could classify their first order definability. Lastly, as problems in our field are often approached from an algebraic context it may be enlightening and useful to translate our combinatorial classifications and observations into the language of algebra and investigate if these results lead to any new and non-trivial observations in this field.

Appendix A

Dismantling Software

In this appendix we provide the source code to our dismantling software. This program is capable of drawing and dismantling relational structures with 1, 2 or 3 binary relations. It can also run exhaustive searches on which transitive tournaments on n vertices dismantle with the transitive tournament T_n with the natural ordering. This result can either be listed as the positive or negative outcome (ie. all bad pairs or all good pairs).

Dismantle.java is the main class and the class where one defines the parameters of the current dismantlings. These parameters include defining the structures to be dismantled or to set up an exhaustive search for transitive tournaments. The other 3 classes Graph.java, RStruct.java, RStruct3.java define the objects used throughout the dismantling. They represent the structures with 1,2, or 3 binary relations respectively.

A.1 Dismantle.java

```
package dismantleFO;
import java.awt.*;

import java.util.Arrays;

import javax.swing.*;

public class Dismantle extends JFrame{
```

```
int[][] arra = new int[factorial(4)][];

public static void main(String arg[]){
  Dismantle frame = new Dismantle();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(1200,800);
  frame.setVisible(true);
}

public Dismantle(){
  super();
}

int hi = 0;

public void paint(Graphics g){

  Boolean oneDis = true; //true if we are dismantling one structure,
     false if we want to do exhaustive (tournaments)
  Boolean twoGraphs = true; //if structure has 2 binary relations
  Boolean threeGraphs = false; //if structure has 3 binary realtions

  int num = 5; //number of vertices
  int generator = 7; //used for randomizing vertex location in the
     digraph drawings.
  int mover = 7; //distance between arrow heads going in and out of
     vertex

  if(!oneDis){
    int[] intArray = new int[] {0,1,2,3,4};
    permute(intArray, 0);
  }

  String[] relations = {"0 1,0 2,0 3,0 4,1 2,1 3,1 4,2 3,2 4,3
     4",""}; //Definitions of the digraphs
  String[] relations2 = {"3 0,3 2,3 1,3 4,0 2,0 1,0 4,2 1,2 4,1
     4",""}; // and edge (a,b) is defined by "a b"
```

```
String[] relations3 = {"2 0,2 3,2 1,2 4,0 3,0 1,0 4,3 1,3 4,1
   4",""}; //these each define transitive tournaments
Graph orig = new Graph(num, relations); //stores a copy of the
   original graph for single dismantling

//This is the method that lets you dismantle a structure with 3
   binary relations
if (threeGraphs){
  int draw = 0;
  g.setColor(Color.WHITE);
  g.fillRect(0, 0, 1200, 800);

  if (draw < 1){
    if (oneDis){

      Graph Graph = new Graph(num,relations);
      Graph orig2 = new Graph(num,relations2);
      Graph Graph2 = new Graph(num, relations2);
      Graph orig3 = new Graph(num,relations3);
      Graph Graph3 = new Graph(num, relations3);

      RStruct3 RS = new RStruct3(Graph, Graph2, Graph3);

      RS.setOneDis(false);

      orig.GraphDraw(g,generator,mover,240,300,60,false);
      orig2.GraphDraw(g,generator,mover,440,160,60,false);
      orig3.GraphDraw(g,generator,mover,440,300,60,false);

      RS.dismantle();

      RS.getg1().GraphDraw(g,generator,mover,350,590,160,true);
      RS.getg2().GraphDraw(g,generator,mover,850,220,160,true);
      RS.getg3().GraphDraw(g,generator,mover,850,590,160,true);

    }else{ //TRANSITIVE TOURNAMENT LOOP
      int graphC = 0;
      int counter = 0;
```

```
        Graph Graph = new Graph(num,relations);
        Graph.squareGraph();
        for (int i = 0; i < factorial(num);i++){
          String[] g2Rel = {relabel(relations,arra[i]),""};
          Graph Graph2 = new Graph(num, g2Rel);
          Graph2.squareGraph();
          for(int j = 0; j < factorial(num); j++){
            counter++;
            String[] g3Rel = {relabel(relations,arra[j]),""};
            Graph Graph3 = new Graph(num, g3Rel);
            Graph3.squareGraph();
            RStruct3 RS = new RStruct3(Graph, Graph2, Graph3);
            RS.setOneDis(false);
            RS.dismantle();

            if(i == 0 || RS.getg1().getIsDis() == false ||
               RS.getg2().getIsDis() == false ||
               RS.getg3().getIsDis() == false){
              graphC++;
              System.out.println(counter + " " + i + " BAD " +
                 graphC + " --- G2: " + Arrays.toString(arra[i]) +
                 " G3: " + Arrays.toString(arra[j]));
            }
          }
        }
      }
      draw++;
    }
//this is the method for dismantling structures with 2 binary
   relations
}else if (twoGraphs){

    int draw = 0;
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, 1200, 800);

    if (draw < 1){
      if (oneDis){
```

```
Graph Graph = new Graph(num,relations);
Graph orig2 = new Graph(num,relations2);
Graph Graph2 = new Graph(num, relations2);
RStruct RS = new RStruct(Graph, Graph2);

RS.setOneDis(true);

orig.GraphDraw(g,generator,mover,500,450,60,false);
orig2.GraphDraw(g,generator,mover,700,450,60,false);
Graph.GraphDraw(g,generator,mover,300,250,160,true);
Graph2.GraphDraw(g,generator,mover,900,250,160,true);

RS.dismantle();

RS.getg1().GraphDraw(g,generator,mover,300,620,160,true);
RS.getg2().GraphDraw(g,generator,mover,900,620,160,true);

}else{
  int graphC = 0;
  for (int i = 0; i < factorial(num);i++){
    Graph Graph = new Graph(num,relations);
    String[] g2Rel = {relabel(relations,arra[i]),""};
    Graph Graph2 = new Graph(num, g2Rel);
    RStruct RS = new RStruct(Graph, Graph2);
    RS.setOneDis(false);
    RS.dismantle();

    if(Graph.getIsDis() == false || Graph2.getIsDis() ==
       false){
      graphC++;
      System.out.println(i + " BAD " + graphC + " " +
         Arrays.toString(arra[i]));
    }
  }
}
draw++;
```

```
      }

    //This is the method for dismantling a single digraph
    }else{
      Graph Graph = new Graph(num,relations);
      //Clear Screen
      g.setColor(Color.WHITE);
      g.fillRect(0, 0, 1200, 800);
      if(hi < 1){

        Graph.GraphDraw(g,1,mover,600,200,100,false);
        Graph.squareGraph();
        Graph.GraphDraw(g,generator,mover,300,500,250,true);

        //the dismantling algorithm
        int domCount = 0;
        int[] dominate = {0,0};

        while(dominate[0] != -1){
          dominate = Graph.findDomVert();
          if(dominate[0] != -1){
              Graph.RemoveVert(dominate[1]);
              domCount++;
          }
        }
        //draws the dominated graph.
        Graph.GraphDraw(g,generator,mover,900,500,250,true);
      }
      //double buffer of input graph image
      orig.GraphDraw(g,1,mover,599,200,100,false);
      hi++;
    }
  }

  //this function generates transitive tournaments with all possible
     orderings.
  public String relabel(String[] relations, int[] labels){
    String[] edges = relations[0].split(",");
```

```
  int numEdges = edges.length;
  String newLabel = "";

  for(int i = 0; i < numEdges; i++){
    String verts[] = edges[i].split(" ");
    newLabel =
       newLabel.concat(Integer.toString(labels[Integer.valueOf(verts[0])])
       + " " + labels[Integer.valueOf(verts[1])] + ",");
  }
  newLabel = newLabel.substring(0, newLabel.length()-1);

  String[] sorter = newLabel.split(",");
  Arrays.sort(sorter);
  newLabel = "";

  for(int i = 0; i < numEdges; i++){
    newLabel = newLabel.concat(sorter[i] + ",");
  }

  newLabel = newLabel.substring(0, newLabel.length()-1);

  return newLabel;
}

int count = 0;

public void permute(int[] input2, int startIndex) {
  int length = input2.length;

   if (length == startIndex + 1 && count < factorial(length)) {
    arra[count] = new int[5];
    arra[count] = input2;
    count++;
   } else {
    for (int i = startIndex; i < length; i++) {
      int[] input = input2.clone();
        int temp = input[i];
        input[i] = input[startIndex];
```

```
                input[startIndex] = temp;
                permute(input, startIndex + 1);
            }
        }
    }

    public int factorial(int a){
      int result = 1;
      for (int i = 1; i <= a; i++){
        result = result*i;
      }
      return result;
    }
}
```

A.2 Graph.java

```
package dismantleFO;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.Hashtable;

public class graph {

  private int numVert, numEdges, oldVert;
  private String[] relSet, newEdges,edges,newVert,vertSet;
  private String[][] inNeighbours, outNeighbours,outEdges,inEdges;

  private int[] numG2EdgeIn, numG2EdgeOut,numOutEdge,numInEdge;
  private String[][] domPair;
  private boolean isDis,oneDis;

  //square graph constructor
  public graph (int numVert, String[] relSet){

    this.numVert = numVert;
    this.relSet = relSet;
    oldVert = numVert;
    vertSet = new String[numVert*numVert];
    domPair = new String[numVert*numVert][2];
    isDis = false;
    for(int i = 0; i< numVert; i++){
      vertSet[i] = Integer.toString(i);
    }

    String[] sig = new String[relSet.length];

    //Consructs a dismantling signature for the graph
    for(int i = 0; i < relSet.length ;i++ ){
      String relSplit[] = relSet[i].split(",");
```

```
    int a = relSplit.length;
    sig[i] = Integer.toString(a);
}

numEdges = relSet[0].split(",").length;
edges = new String[numEdges];

edges = relSet[0].split(",");
outEdges = new String[numVert][numVert+1];
inEdges = new String[numVert][numVert + 1];

//Assigns the name to each outEdge array for the input graph
int r = 1;
for(int i = 0; i < numVert;i++){
    outEdges[i][0] = Integer.toString(i);
}
//Gathers the out edges of the input graph
for(int i = 0;i < numVert;i++){
    for(int t = 0; t< numEdges;t++){
        String[] g1node = edges[t].split(" ");
        if(g1node[0].equals(outEdges[i][0])){
            outEdges[i][r] = g1node[1];
            r++;
        }
    }
r = 1;
}

r = 1;
for(int i = 0; i < numVert;i++){
    inEdges[i][0] = Integer.toString(i);
}
//Gathers the out edges of the input graph
for(int i = 0;i < numVert;i++){
    for(int t = 0; t< numEdges;t++){
        String[] g1node = edges[t].split(" ");
        if(g1node[1].equals(outEdges[i][0])){
            inEdges[i][r] = g1node[0];
```

```
          r++;
        }
      }
    r = 1;
    }

    r = 0;

    //Count G1 in and out
    int q = 0;
    int out = 0;
    int in = 0;
    numOutEdge = new int[numVert];
    numInEdge = new int[numVert];

    for(int i = 0; i < numVert; i++){
      for(int t = 1; t < numVert + 1; t++){
        if(outEdges[i][t] != null){
          out++;
        }
        if(inEdges[i][t] != null){
          in++;
        }

      }
      numOutEdge[q] = out;
      numInEdge[q] = in;
    out = 0;
    in = 0;
    q++;
    }
}

//This methid squares the input relational structure
public void squareGraph(){
  newVert = new String[numVert*numVert] ;
  //Constructs vertices of the square
    int k = 0;
```

```
    for (int i = 0;i < numVert; i++){
      for (int t = 0;t < numVert; t++){
      newVert[k] = Integer.toString(i) +Integer.toString(t);
      k++;
      }
    }
    newEdges = new String[numEdges*numEdges];
    outNeighbours = new String[numVert*numVert][numVert*numVert+1];
    inNeighbours = new String[numVert*numVert][numVert*numVert+1];
  int r = 0;
    //constructs the new edges of the square graph
    for(int i = 0;i < numEdges;i++){
      for(int t = 0; t < numEdges; t++){
      String edgeSplit1[] = edges[i].split(" ");
      String edgeSplit2[] = edges[t].split(" ");
      newEdges[r] = edgeSplit1[0]+edgeSplit2[0]+"
         "+edgeSplit1[1]+edgeSplit2[1];
      r++;
      }
    }
    r = 0;
    for(int i = 0; i< numVert; i++){
      for(int t = 0; t < numVert; t++){
        outNeighbours[r][0] = Integer.toString(i)+Integer.toString(t);
        r++;
      }
    }

    r = 0;
    //sets name of each in edge array for square graph
    for(int i = 0; i< numVert; i++){
      for(int t = 0; t < numVert; t++){
        inNeighbours[r][0] = Integer.toString(i)+Integer.toString(t);
      r++;
      }
    }

    int numNewEdges = newEdges.length;
```

```
int e = 1;
//gathers the out neighbours for square
for(int i = 0; i<numVert*numVert;i++){
  for(int t = 0; t < numNewEdges; t++){
    String newNode = outNeighbours[i][0];
    String[] nodes = newEdges[t].split(" ");
    if (nodes[0].equals(newNode)){
      for(int p = 0; p < numVert*numVert;p++){
        if(nodes[1].equals(newVert[p])){
          outNeighbours[i][e] = Integer.toString(p);
        }
      }
      e++;
    }
  }
e = 1;
}

e = 1;
//gathers in neighbours of square graph
for(int i = 0; i<numVert*numVert;i++){
  for(int t = 0; t < numNewEdges; t++){

    String newNode = inNeighbours[i][0];
    String[] nodes = newEdges[t].split(" ");
    if (nodes[1].equals(newNode)){
      for(int p = 0; p < numVert*numVert;p++){
        if(nodes[0].equals(newVert[p])){
          inNeighbours[i][e] = Integer.toString(p);
        }
      }
      e++;
    }
  }
e = 1;
}

int q = 0;
```

```
    int out = 0;
    int in = 0;
    numG2EdgeIn = new int[numVert*numVert];
    numG2EdgeOut = new int[numVert*numVert];
    //counts number of out/in edges for each new vertex
    for(int i = 0; i < numVert*numVert; i++){
      for(int t = 1; t < numVert*numVert + 1; t++){
        if(outNeighbours[i][t] != null){
          out++;
        }
        if(inNeighbours[i][t] !=null){
          in++;
        }
      }
      numG2EdgeIn[q] = in;
      numG2EdgeOut[q] = out;
      out = 0;
      in = 0;
      q++;
    }
    outEdges = outNeighbours;
    inEdges = inNeighbours;
    vertSet = newVert;
    numVert = numVert*numVert;
    numInEdge = numG2EdgeIn;
    numOutEdge = numG2EdgeOut;

  }

//This is the graph drawing method

  public void GraphDraw(Graphics g, int coprime, int moverRate, int
     CentreX,int CentreY, int Rad, boolean Square){

    g.setColor(Color.BLACK);
    int x1,y1;
    double degrees;
    int[][] coords = new int[numVert][2];
```

```
  int counter = 0;
  int firstNum = 0;

  //draw input graph vertices and labels
  for (int i = 0; i< numVert; i++){

    degrees =
       ((360/(numVert))*((coprime*(i+1))%(numVert)))*(Math.PI/180);
    x1 = (int) Math.round(Rad*Math.cos(degrees));
    y1 = (int) Math.round(Rad*Math.sin(degrees));

    if (!Square){
      g.drawString(Integer.toString(i), CentreX+x1, CentreY-5 -y1);
    }else{
      g.drawString(Integer.toString(firstNum)+
         Integer.toString(i%oldVert), CentreX+x1, CentreY-5 -y1);
    }
    g.setColor(Color.GREEN);
    g.fillRect(CentreX+x1,CentreY-y1, 10, 10);
    g.setColor(Color.BLACK);

    coords[i][0] = x1+CentreX;
    coords[i][1] = -y1+CentreY;
    counter++;

    if(counter%oldVert == 0){
      firstNum++;
    }
  }


//sets up the g1mover array that spreads out edge ends
  int[] g1mover = new int[numVert];
  int[] g1moverOut = new int[numVert];
  for (int i =0; i < numVert;i++){
    g1mover[i] = moverRate;
    g1moverOut[i] = moverRate;
  }
```

```
  //draw Edges
    for (int i = 0; i < numVert; i++){
      for(int t = 0; t < numOutEdge[i]; t++){
        if(i == Integer.valueOf(outEdges[i][t+1])){
          g.drawOval(coords[i][0], coords[i][1], 30, 30);
        }else{
          g.drawLine(coords[i][0]-g1moverOut[i],
             coords[i][1],coords[Integer.valueOf(outEdges[i][t+1])][0]+g1mover[In
             , coords[Integer.valueOf(outEdges[i][t+1])][1]);
          g.setColor(Color.RED);
          g.fillRect(coords[Integer.valueOf(outEdges[i][t+1])][0] +
             g1mover[Integer.valueOf(outEdges[i][t+1])] ,
             coords[Integer.valueOf(outEdges[i][t+1])][1],5,5);
          g.setColor(Color.BLUE);
          g.fillRect(coords[i][0] - g1moverOut[i] , coords[i][1],5,5);
          g.setColor(Color.BLACK);
          g1mover[Integer.valueOf(outEdges[i][t+1])] =
             g1mover[Integer.valueOf(outEdges[i][t+1])] + moverRate;
          g1moverOut[i] = g1moverOut[i] + moverRate;
        }
      }
    }
  }


//FIND DOMINATED VERTEX METHOD
public int[] findDomVert(){
    int dom[] = {0,0};
    boolean foundDom = false;
    Hashtable<Integer, String> Hin = new Hashtable<Integer, String>();
    Hashtable<Integer, String> Hout = new Hashtable<Integer, String>();
    int countIn=0,countOut=0;
    int iter = 0;
    int isoCount = 0;

    while (!foundDom){
      Hout.clear();
```

```
Hin.clear();
for(int i = 0; i < numOutEdge[iter];i++){
  Hout.put(i,outEdges[iter][i+1]);
}
for(int i = 0; i < numInEdge[iter];i++){
  Hin.put(i,inEdges[iter][i+1]);
}

boolean iso = false;
if(numInEdge[iter] == 0 && numOutEdge[iter] == 0){
  iso = true;
  isoCount++;
}
for(int r = 0; r < numVert; r++){
  countIn = 0;
  countOut = 0;

  if(r != iter && iso == false){
  for (int k =0; k < numOutEdge[r];k++){
    if(Hout.containsValue(outEdges[r][k+1]) == true){
      countOut++;
    }
  }
  for (int k = 0; k< numInEdge[r];k++){
    if(Hin.containsValue(inEdges[r][k+1]) == true){
      countIn++;
    }
  }


if(countIn == numInEdge[iter] && countOut == numOutEdge[iter]
   && iter%(oldVert+1) != 0){
  String a = Character.toString(getVertSet()[iter].charAt(0)) +
     Character.toString(getVertSet()[iter].charAt(0));
  String b = Character.toString(getVertSet()[iter].charAt(1)) +
     Character.toString(getVertSet()[iter].charAt(1));;

  if(getVertSet()[r].equals(a) || getVertSet()[r].equals(b)) {
```

```
                dom[0] = r;
                dom[1] = iter;
                foundDom = true;
                break;
              }
            }
          }
          iter++;
          if(iter == numVert){
            foundDom = true;
            dom[0] = -1;
            dom[1] = -1;
          }
        }
        return dom;
      }
      //FIND DOMINATED LIST
      public void findDomList() {

          boolean foundDom = false;
          Hashtable<Integer, String> Hin = new Hashtable<Integer, String>();
          Hashtable<Integer, String> Hout = new Hashtable<Integer, String>();
          int countIn = 0, countOut = 0;
          int iter = 0;


          while (!foundDom) {
            Hout.clear();
            Hin.clear();
            for (int i = 0; i < numOutEdge[iter]; i++) {
              Hout.put(i, outEdges[iter][i + 1]);
            }
            for (int i = 0; i < numInEdge[iter]; i++) {
              Hin.put(i, inEdges[iter][i + 1]);
            }

            boolean iso = false;
            if (numInEdge[iter] == 0 && numOutEdge[iter] == 0) {
```

```
      iso = true;
      //set the dominated pair list for any isolated vertex
      domPair[iter][0] = newVert[(int) (Math.floor(iter / oldVert)
         * (oldVert + 1))];
      domPair[iter][1] = newVert[(iter % oldVert) * (oldVert + 1)];


    }
    for (int r = 0; r < numVert; r++) {
      countIn = 0;
      countOut = 0;

      if (r != iter && iso == false) {
        for (int k = 0; k < numOutEdge[r]; k++) {
          if (Hout.containsValue(outEdges[r][k + 1]) == true) {
            countOut++;
          }
        }
        for (int k = 0; k < numInEdge[r]; k++) {

          if (Hin.containsValue(inEdges[r][k + 1]) == true) {
            countIn++;
          }
        }



        if (countIn == numInEdge[iter]
            && countOut == numOutEdge[iter]
            && iter % (oldVert + 1) != 0) {

          if (r == Math.floor(iter / oldVert) * (oldVert + 1)) {
              domPair[iter][0] = newVert[r];
            }
          if (r == (iter % oldVert) * (oldVert + 1)) {
              domPair[iter][1] = newVert[r];
          }
```

```
        }
      }
    }
    iter++;
    if(iter == numVert){
      foundDom = true;
    }
  }
}

//REMOVE VERTEX METHOD
public void RemoveVert(int vertInt){

  String vert = vertSet[vertInt];
  for(int i = 0; i < numOutEdge[vertInt]; i++){
    outEdges[vertInt][i+1] = null;
  }
  for(int i = 0; i < numInEdge[vertInt]; i++){

    inEdges[vertInt][i+1] = null;
  }
  numInEdge[vertInt] = 0;
  numOutEdge[vertInt] = 0;

  //remove vertex from outedges
  int rCount = 1;
  for(int i = 0; i < numVert; i++){
    String[] temp = new String[numVert+1];
    temp[0] = outEdges[i][0];
    for(int t = 0; t < numOutEdge[i]; t++){
      if(outEdges[i][t+1].equals(Integer.toString(vertInt)) ==
         false){
        temp[rCount] = outEdges[i][t+1];
        rCount++;
      }
    }
    outEdges[i] = temp;
    numOutEdge[i] = numOutEdge[i] - (numOutEdge[i] - (rCount-1));
```

```
      rCount = 1;
    }

    //remove vertex from in edges
    rCount = 1;
    for(int i = 0; i < numVert; i++){
      String[] temp = new String[numVert+1];
      temp[0] = inEdges[i][0];
      for(int t = 0; t < numInEdge[i]; t++){
        if(inEdges[i][t+1].equals(Integer.toString(vertInt)) ==
           false){
          temp[rCount] = inEdges[i][t+1];
          rCount++;
        }
      }
      inEdges[i] = temp;
      numInEdge[i] = numInEdge[i] - (numInEdge[i] - (rCount-1));
      rCount = 1;
    }
  }
  //Checks if a particular graph is dismantled to the diagonal after
     Dismantle() is called.
  public void isDismantled(){
    for(int i = 0; i < numVert;i++){
      if(numInEdge[i] > 0 || numOutEdge[i] > 0){
        if (i % (oldVert + 1) != 0){
          isDis = false;
          break;
        }
      }else{
        isDis = true;
      }
    }
  }

  public boolean getIsDis(){
    isDismantled();
    return isDis;
```

```
}

public int getNumVert() {
  return numVert;
}

public void setNumVert(int numVert) {
  this.numVert = numVert;
}

public String[] getRelSet() {
  return relSet;
}

public void setRelSet(String[] relSet) {
  this.relSet = relSet;
}

public String[] getNewEdges() {
  return newEdges;
}

public void setNewEdges(String[] newEdges) {
  this.newEdges = newEdges;
}

public String[][] getInNeighbours() {
  return inNeighbours;
}

public void setInNeighbours(String[][] inNeighbours) {
  this.inNeighbours = inNeighbours;
}

public String[][] getOutNeighbours() {
  return outNeighbours;
}
```

```
public void setOutNeighbours(String[][] outNeighbours) {
  this.outNeighbours = outNeighbours;
}

public int[] getNumG2EdgeIn() {
  return numG2EdgeIn;
}

public void setNumG2EdgeIn(int[] numG2EdgeIn) {
  this.numG2EdgeIn = numG2EdgeIn;
}

public int[] getNumG2EdgeOut() {
  return numG2EdgeOut;
}

public void setNumG2EdgeOut(int[] numG2EdgeOut) {
  this.numG2EdgeOut = numG2EdgeOut;
}

public String[][] getInEdges() {
  return inEdges;
}

public void setInEdges(String[][] inEdges) {
  this.inEdges = inEdges;
}

public String[] getNewVert() {
  return newVert;
}

public void setNewVert(String[] newVert) {
  this.newVert = newVert;
}

public int getNumEdges() {
  return numEdges;
```

```
}

public void setNumEdges(int numEdges) {
  this.numEdges = numEdges;
}

public String[] getEdges() {
  return edges;
}

public void setEdges(String[] edges) {
  this.edges = edges;
}

public String[][] getOutEdges() {
  return outEdges;
}

public void setOutEdges(String[][] outEdges) {
  this.outEdges = outEdges;
}

public int[] getNumEdgeOut() {
  return numOutEdge;
}

public void setNumEdgeOut(int[] numEdgeOut) {
  this.numOutEdge = numEdgeOut;
}

public String[] getVertSet() {
  return vertSet;
}

public void setVertSet(String[] vertSet) {
  this.vertSet = vertSet;
}
```

```java
  public String[][] getDomPair() {
    return domPair;
  }

  public void setDomPair(String[][] domPair) {
    this.domPair = domPair;
  }

  public boolean isOneDis() {
    return oneDis;
  }

  public void setOneDis(boolean oneDis) {
    this.oneDis = oneDis;
  }

}
```

A.3 RStruct.java

```
package dismantleFO;

import java.lang.reflect.Array;
import java.util.Arrays;

public class RStruct {
  Graph g1,g2;
  int numVert;
  boolean oneDis;

  //This is the relational structure with 2 binary relations
  public RStruct(Graph g1, Graph g2){
    this.g1 = g1;
    this.g2 = g2;

    g1.squareGraph();
    g2.squareGraph();
    this.numVert = g1.getNumVert();

    for(int i = 0; i < numVert; i++){
      g1.getDomPair()[i][0] = "1";
      g1.getDomPair()[i][1] = "1";
      g2.getDomPair()[i][0] = "2";
      g2.getDomPair()[i][1] = "2";
    }

  }

  //this is the dismantling algorithm for a relataional structure with
     2 binary relations
  public void dismantle(){
    int removeCount = 1;
    boolean[] vRemoved = new boolean[numVert];

    for(int i = 0; i < numVert; i++){
      vRemoved[i] = false;
```

```
}

int steps = 0;
int prevRemoveCount = 1;
int totRemove = 0;
while(removeCount != 0){
  prevRemoveCount = removeCount;
  steps++;
  removeCount = 0;

  if(oneDis){
    System.out.println("--------------------------------");
  }

  g1.findDomList();
  g2.findDomList();

  for(int i = 0; i < numVert; i++){

    if(g1.getDomPair()[i][0].equals(g2.getDomPair()[i][0]) ||
       g1.getDomPair()[i][0].equals(g2.getDomPair()[i][1])){
      if((g1.getDomPair()[i][0] != "" || g1.getDomPair()[i][1] !=
         "") && vRemoved[i] == false ){
        g1.RemoveVert(i);
        g2.RemoveVert(i);
        if (oneDis){
          System.out.println(totRemove+1 + " " + "REMOVED: " +
             g1.getNewVert()[i] + " DOM'D BY " +
             g1.getDomPair()[i][0]);
        }
        removeCount++;
        totRemove++;
        vRemoved[i] = true;
      }
    }

    if(g1.getDomPair()[i][1].equals(g2.getDomPair()[i][0]) ||
       g1.getDomPair()[i][1].equals(g2.getDomPair()[i][1])){
```

```
          if((g1.getDomPair()[i][0] != "" || g1.getDomPair()[i][1] !=
             "") && vRemoved[i] == false){
            g1.RemoveVert(i);
            g2.RemoveVert(i);
            if(oneDis){
              System.out.println(totRemove+1 + " " + "REMOVED: " +
                 g1.getNewVert()[i] + " DOM'D BY " +
                 g1.getDomPair()[i][1]);
            }
              removeCount++;
            totRemove++;
            vRemoved[i] = true;
          }
        }

      }

    }
    if (oneDis){
      System.out.println("Removed: " + totRemove + " vertices in " +
         (steps-1) + " steps." );
    }
  }

  public boolean isOneDis() {
    return oneDis;
  }

  public void setOneDis(boolean oneDis) {
    this.oneDis = oneDis;
  }

  public Graph getg1() {
    return g1;
  }

  public void setg1(Graph G1) {
    g1 = G1;
```

```
  }

  public Graph getg2() {
    return g2;
  }

  public void setG2(Graph G2) {
    g2 = G2;
}

}
```

A.4 RStruct3.java

```
package dismantleFO;

import java.lang.reflect.Array;
import java.util.Arrays;

public class RStruct3 {
  Graph g1,g2, g3;
  int numVert;
  boolean oneDis;

  //Constructor methid for relational structure with 3 binary relations
  public RStruct3(Graph g1, Graph g2, Graph g3){
    this.g1 = g1;
    this.g2 = g2;
    this.g3 = g3;

    g1.squareGraph();
    g2.squareGraph();
    g3.squareGraph();

    this.numVert = g1.getNumVert();

    for(int i = 0; i < numVert; i++){
      g1.getDomPair()[i][0] = "1";
      g1.getDomPair()[i][1] = "1";
      g2.getDomPair()[i][0] = "2";
      g2.getDomPair()[i][1] = "2";
      g3.getDomPair()[i][0] = "3";
      g3.getDomPair()[i][1] = "3";
    }

  }

  //dismantling algorithm for structures with 3 binary relations
  public void dismantle(){
    System.out.println("--------------------------------");
```

```
int removeCount = 1;
boolean[] vRemoved = new boolean[numVert];

for(int i = 0; i < numVert; i++){
  vRemoved[i] = false;
}

int steps = 0;
int prevRemoveCount = 1;
int totRemove = 0;
while(removeCount != 0){
  prevRemoveCount = removeCount;
  steps++;
  removeCount = 0;

  System.out.println("--------------------------------");

  g1.findDomList();
  g2.findDomList();
  g3.findDomList();

  for(int i = 0; i < numVert; i++){

    if((g1.getDomPair()[i][0].equals(g2.getDomPair()[i][0]) ||
        g1.getDomPair()[i][0].equals(g2.getDomPair()[i][1])) &&
        (g1.getDomPair()[i][0].equals(g3.getDomPair()[i][0]) ||
        g1.getDomPair()[i][0].equals(g3.getDomPair()[i][1]))){
      if((g1.getDomPair()[i][0] != "" || g1.getDomPair()[i][1] !=
          "") && vRemoved[i] == false ){
        g1.RemoveVert(i);
        g2.RemoveVert(i);
        g3.RemoveVert(i);
        System.out.println(totRemove+1 + " " + "REMOVED: " +
            g1.getNewVert()[i] + " DOM'D BY " +
            g1.getDomPair()[i][0]);
        removeCount++;
        totRemove++;
        vRemoved[i] = true;
```

```
        }
      }

      if((g1.getDomPair()[i][1].equals(g2.getDomPair()[i][0]) ||
         g1.getDomPair()[i][1].equals(g2.getDomPair()[i][1])) &&
         (g1.getDomPair()[i][1].equals(g3.getDomPair()[i][0]) ||
         g1.getDomPair()[i][1].equals(g3.getDomPair()[i][1]))){
        if((g1.getDomPair()[i][0] != "" || g1.getDomPair()[i][1] !=
           "") && vRemoved[i] == false){
          g1.RemoveVert(i);
          g2.RemoveVert(i);
          g3.RemoveVert(i);
          System.out.println(totRemove+1 + " " + "REMOVED: " +
             g1.getNewVert()[i] + " DOM'D BY " +
             g1.getDomPair()[i][1]);
          removeCount++;
          totRemove++;
          vRemoved[i] = true;
        }
      }
    }
  }
  if (oneDis){
    System.out.println("Removed: " + totRemove + " vertices in " +
       (steps-1) + " steps." );
  }
}

public boolean isOneDis() {
  return oneDis;
}

public void setOneDis(boolean oneDis) {
  this.oneDis = oneDis;
}

public Graph getg1() {
  return g1;
```

```
  }

  public void setg1(Graph G1) {
    g1 = G1;
  }

  public Graph getg2() {
    return g2;
  }

  public void setG2(Graph G2) {
    g2 = G2;
  }
  public Graph getg3() {
    return g3;
  }
}
```

Bibliography

[1] Tomás Feder and Moshe Vardi. Monotone Monadic SNP and Constraint Satisfaction. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 612–622, New York, NY, USA, 1993. ACM.

[2] Toms Feder and Moshe Vardi. The Computational Structure of Monotone Monadic SNP and Constraint Satisfaction: A Study Through Datalog and Group Theory. 28:57–104, 01 1998.

[3] Pavol Hell and Jaroslav Nešetřil. On the Complexity of H-coloring. *J. Comb. Theory Ser. B*, 48(1):92–110, February 1990.

[4] Andrei A. Bulatov. Tractable Conservative Constraint Satisfaction Problems. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, LICS '03, pages 321–, Washington, DC, USA, 2003. IEEE Computer Society.

[5] B Larose, C Loten, and Claude Tardif. A Characterisation of First-Order Constraint Satisfaction Problems. *Logical Methods in Computer Science*, 3, 4 2007.

[6] Adrien Lemaître. *Complexité des Homomorphismes de Graphes avec Listes*. PhD thesis, Université de Montréal, 2012.

[7] P. Hell and J. Nešetřil. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications. OUP Oxford, 2004.

[8] Ralph McKenzie, G.F. McNulty, and W. Taylor. *Algebras, Lattices, Varieties*. Number v. 1 in Wadsworth & Brooks/Cole mathematics series. Wadsworth & Brooks/Cole Advanced Books & Software, 1987.

[9] A. Bondy and U.S.R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer London, 2011.

[10] Juris Hartmanis and Richard E Stearns. On the Computational Complexity of Algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[11] L. A. Levin. Universal Sequential Search Problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[12] Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[13] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[14] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[15] Avi Wigderson. Mathematics and Computation. https://www.math.ias.edu/files/mathandcomp.pdf, 2018.

[16] N. Immerman. *Descriptive Complexity*. Texts in Computer Science. Springer New York, 1998.

[17] Víctor Dalmau, Marcin Kozik, Andrei Krokhin, Konstantin Makarychev, Yury Makarychev, and Jakub Opršal. Robust Algorithms with Polynomial Loss for Near-unanimity CSPs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 340–357, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[18] L. Egri, A. Krokhin, B. Larose, and P. Tesson. The Complexity of the List Homomorphism Problem forGraphs. *Theor. Comp. Sys.*, 51(2):143–178, 2012.

[19] A. A. Bulatov. A Dichotomy Theorem for Nonuniform CSPs. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330, Oct 2017.

[20] D. Zhuk. A Proof of CSP Dichotomy Conjecture. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 331–342, Oct 2017.

[21] Benoît Larose and Pascal Tesson. Universal Algebra and Hardness Results for Constraint Satisfaction Problems. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, pages 267–278, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.