Semigroup Congruences: Computational Techniques and Theoretical Applications

Michael Torpey

University of St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the University of St Andrews

February 2019

Declarations

Candidate's declaration

I, Michael Colin Torpey, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 59,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2014.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date: .... Signature of candidate: ....

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date: .... Signature of supervisor: ....

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Michael Colin Torpey, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis: no embargo on print copy; no embargo on electronic copy.

Date: .... Signature of candidate: ....

Date: .... Signature of supervisor: ....

Underpinning Research Data or Digital Outputs

I, Michael Colin Torpey, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date: .... Signature of candidate: ....

Abstract

Computational semigroup theory is an area of research that is subject to growing interest. The development of semigroup algorithms allows for new theoretical results to be discovered, which in turn informs the creation of yet more algorithms. Groups have benefitted from this cycle since before the invention of electronic computers, and the popularity of computational group theory has resulted in a rich and detailed literature. Computational semigroup theory is a less developed field, but recent work has resulted in a variety of algorithms, and some important pieces of software such as the Semigroups package for GAP.

Congruences are an important part of semigroup theory. A semigroup's congruences determine its homomorphic images in a manner analogous to a group's normal subgroups. Prior to the work described here, there existed few practical algorithms for computing with semigroup congruences. However, a number of results about alternative representations for congruences, as well as existing algorithms that can be borrowed from group theory, make congruences a fertile area for improvement. In this thesis, we first consider computational techniques that can be applied to the study of congruences, and then present some results that have been produced or precipitated by applying these techniques to interesting examples.

After some preliminary theory, we present a new parallel approach to computing with congruences specified by generating pairs. We then consider alternative ways of representing a congruence, using intermediate objects such as linked triples. We also present an algorithm for computing the entire congruence lattice of a finite semigroup. In the second part of the thesis, we classify the congruences of several monoids of bipartitions, as well as the principal factors of several monoids of partial transformations. Finally, we consider how many congruences a finite semigroup can have, and examine those on semigroups with up to seven elements.

Contents

List of figures, tables and algorithms

Preface

Semigroup theory has its roots in group theory, and no thesis on semigroups would be complete without a discussion of groups. The history of group theory goes back centuries: groups were studied in some form as early as the late 1700s by Lagrange, in order to solve numerical equations; this work was continued in the early 1800s by Galois, who first used the word *group* to describe them. Groups were then studied in various different contexts – in geometry, in number theory, and as permutation groups – for some time before the various branches of theory were united into one, with von Dyck inventing the modern abstract definition of a group in 1882 [Dyc82]. The study of group theory has flourished since then, becoming a major area of research in pure mathematics. By contrast, semigroup theory is a relatively young area of study, having been defined only in the early 1900s and having been studied very little before the 1950s. A few early papers by authors such as Suschkewitch [Sus28] started things off, but it was not until the second half of the twentieth century that semigroup theory really gained traction. It now accounts for a significant body of work, with dedicated journals such as Semigroup Forum, and seminal books such as [How95] and [Pet84] – but it has never become as popular at its older counterpart, resulting in an interesting relationship between semigroup theory and group theory.

Since any group is a semigroup, it might be imagined that groups would be studied simply as a special case within semigroup theory. In practice, however, we see the reverse: group theory tends to inform semigroup theory, since it turns out that groups are a very important topic within the study of semigroups in general. Important features of a semigroup's structure depend on its maximal subgroups – for example its Green's relations, or in the case of a completely simple semigroup, its linked triples. As a result, groups are a central part of semigroup theory, allowing us to borrow from the richly developed field of group theory in order to solve problems for semigroups that are not groups.

Computational algebra has existed for nearly as long as the theory of computation itself. For instance, perhaps the earliest published group theory algorithm is that by Dehn [Deh11] for solving the word problem in certain groups. The Todd–Coxeter algorithm [TC36], which enumerates the cosets of a subgroup in a group, was certainly designed to be carried out by hand, having been described in the 1930s before the invention of electronic computers. When computers did arrive, there was early interest in using them for group theory problems, with the Todd–Coxeter algorithm being implemented on the EDSAC II in Cambridge as early as 1953 [Lee63]. Since then, computational group theory has flourished, with a variety of software packages such as Magma [BCP97], ACE [RH09], and particularly GAP [GAP18], which has a variety of packages containing algorithms to solve a wide range of problems. A wealth of material is available on computational group theory, including several dedicated books [Sim94, HEO05]. By comparison, computational semigroup theory is much younger and less developed, as is semigroup theory as a whole. Computers were used as early as 1953 to classify semigroups of low order up to isomorphism and anti-isomorphism: all 4 semigroups of order 2 and all 18 semigroups of order 3 were classified by Tamura in 1953 [Tam53]; Forsythe followed in 1955 with the 126 semigroups of order 4 [For55]; and the following year Motzkin and Selfridge found the 1160 semigroups of order 5 [MS56, Jür77]. But despite these early successes, the theory of computing with semigroups developed more slowly than with groups, and no package has yet emerged for semigroups on the scale of the group algorithms in GAP. However, there has been considerable interest in computational semigroup theory in recent years, and increasingly there do exist algorithms for semigroups, as well as software packages implementing them, such as Semigroup for Windows [McA99], Semigroupe [Pin09], libsemigroups [MT$^+$18], and the GAP packages Semigroups [M$^+$19], smallsemi [DM17] and kbmag [Hol19].

In the same way that semigroup theory borrows results from group theory, computational semigroup theory often borrows algorithms from computational group theory. The Todd–Coxeter algorithm, for instance, was originally designed to calculate a subgroup's cosets inside a group – but with a few changes, it can be used to find the elements of a finitely presented semigroup, or the classes of a congruence on such a semigroup, as we will see in this thesis. We also borrow from computational group theory by using properties of a semigroup's subgroups. For example, when computing the linked triples on a completely simple semigroup, we require algorithms from computational group theory to find all the normal subgroups of a given maximal subgroup. In this way, computational group theory is not just a subset, but a key part, of computational semigroup theory.

This thesis deals primarily with the congruences on a semigroup. A semigroup's congruences describe its homomorphic kernels and images – that is, the ways in which the semigroup can be mapped onto another semigroup while preserving the operation. In this way, a semigroup's congruences serve the same function as a group's normal subgroups, or a ring's two-sided ideals, and are of as much interest in semigroup theory as those structures are in their respective fields. Classifying the congruences of important semigroups has long been a major activity in semigroup theory. Some important early examples are the full transformation monoid $\mathcal{T}_n$ by Mal′cev [Mal52], the symmetric inverse monoid $\mathcal{I}_n$ by Liber [Lib53], and the partial transformation monoid $\mathcal{PT}_n$ by Shutov [Shu88] – and more recently, the direct product of any pair of these by Araújo, Bentz and Gomes [ABG18]. A host of other semigroups have had their congruences classified over the years – for example, from [Fer00, FGJ05, FGJ09] we know the congruences on various monoids of partial transformations restricted to elements that preserve or reverse the order or orientation of the set being acted on. Moving away from partial transformations, Mal′cev also classified the congruences on the semigroup F_n of $n \times n$ matrices with entries from a field F [Mal53] – and the principal congruences on a direct product $F_m \times F_n$ of these are also classified in [ABG18]. While these examples are by no means an exhaustive list, there remain many important semigroups whose congruences have not yet been classified. Finding the congruences on various other semigroups will form the majority of Part II of this thesis.

The computational theory of semigroup congruences is also a young field. Algorithms for computing information about a given congruence have existed in the GAP library for many years, but many of them lack sophistication and may take an unreasonably long time to return results about semigroups with more than about 25000 elements. We can improve on these algorithms in two different ways. Firstly we can consider alternative structures that correspond to a semigroup's congruences. For instance, in group theory we consider a group's normal subgroups rather than studying its congruences directly; in the same way, we can study a completely simple semigroup's linked triples, or an inverse semigroup's kernel–trace pairs, in place of their congruences, and thus we can often produce the answers to computational questions more quickly than by using direct methods. This approach forms the basis of the present author's previous works [Tor14a, Tor14b], which are expanded upon in this thesis. Secondly, in the more general cases where no such alternative structure exists, we may still make improvements on existing algorithms by applying new algorithms that are not currently used for congruences, such as the Todd–Coxeter coset enumeration procedure, or the Knuth–Bendix completion process. We will take both approaches in Part I of this thesis, presenting new congruence algorithms, as well as showing ways in which existing algorithms can be successfully adapted for congruence-related purposes.

After introducing some preliminary theory, this thesis is divided into two broad parts. Part I discusses the computational theory of congruences, and presents algorithms that can be used to answer congruence-related questions. Part II shows some results that have been obtained by applying these algorithms, as well as results that have been proven by hand using computational output as a starting point.

Chapter 1 acts as an introduction to this document, providing the preliminary knowledge which is required to understand the material in the rest of the thesis. It is mostly concerned with ideas from semigroup theory such as Green's relations, generators, congruences and presentations. It also introduces several important types of elements that form semigroups, such as partial transformations and bipartitions. Some computational issues such as algorithms and decidability are also covered, as well as some algorithms that are used later on.

Chapter 2 presents a new way of computing with congruences defined by generating pairs. The method presented uses parallel computation to run a variety of algorithms – including the Todd–Coxeter algorithm, the Knuth–Bendix algorithm, and an unsophisticated algorithm known as *pair orbit enumeration* – that test whether a given pair lies in a congruence, given its generating pairs. This approach takes advantage of the different algorithms' abilities to return an answer quickly in various cases, in each case exhibiting run-times close to the minimum of all the different algorithms. Modifications for left and right congruences are explained, and different versions are described for finitely presented semigroups and for *concrete* semigroups (semigroups known in advance to be finite). This approach was implemented in libsemigroups [MT$^+$18], and the results of benchmarking tests on that implementation are shown near the end of the chapter.

Chapter 3 concerns the various ways of representing a congruence, other than as a set of pairs. Five possible representations are described – generating pairs, normal subgroups, linked triples, kernel–trace pairs, and ideals – along with explanations of the precise semigroups and congruences to which they apply. Algorithms are given for converting from one representation to another, so that a computational algebra system may quickly convert a congruence specified in a certain way to the most efficient representation, and thus answer questions about the congruence in as short a time as possible. All twenty of the possible conversions between the five representations are considered: many of these are currently implemented in Semigroups

[M$^+$19], some having never been described before; others are trivial or unnecessary; and two remain open problems (converting from kernel–trace to generating pairs, and from generating pairs to an ideal). See Table 3.1 for a summary of all the conversions described in the chapter.

Chapter 4 presents an algorithm for computing the entire congruence lattice of a finite semigroup. This algorithm is rather rudimentary, but various shortcuts and improvements are described to try to reduce the computational work required, where possible. A version of this algorithm is implemented in Semigroups, and thus takes advantage of the methods described in Chapters 2 and 3. This makes it possible to compute the lattices of any sufficiently small semigroup in a reasonable time; a brief analysis is included of the sizes of semigroup and lattice which are feasible, along with some visual examples of lattices that have been computed using this method.

Chapter 5 considers the Motzkin monoid $\mathcal{M}_n$ – that is, the monoid of all planar bipartitions of degree n with blocks of size no greater than 2. An investigation into the congruence lattice of this monoid was initiated by computational experiments, using the method described in Chapter 4, and resulted in a complete classification of the congruence lattice of $\mathcal{M}_n$ for arbitrary n, along with generating pairs for each congruence. This classification, originally published in [EMRT18], is shown here with the kind permission of my co-authors. Other important monoids of bipartitions are also considered, and their congruence lattices classified.

Chapter 6 completes this thesis by presenting some other results obtained or precipitated by computational experiments in the Semigroups package. Firstly, we consider the congruences on the principal factors of the full transformation monoid $\mathcal{T}_n$ and some other related monoids, classifying them for arbitrary n. Secondly, we consider the number of congruences that exist on an arbitrary finite semigroup. We give some upper and lower bounds for this number based on a semigroup's size, and we present two conjectures about the second-largest number of congruences a finite semigroup can have. To support these conjectures, we also show some computational evidence produced with the aid of smallsemi [DM17]. Finally, we present some findings from an exhaustive classification of the congruences on all 1658439 semigroups of size no larger than 7, up to isomorphism.

At the end of this document we provide an index of the various terms that are used. We also provide a list of notation, with a brief description of what each mathematical symbol means. In both of these, each entry has a reference to the page on which the term or symbol is first defined. In the original digital version of this document, all citations and numbered references act as hyperlinks, allowing the reader to click them to be redirected to the appropriate location.

Acknowledgements

Attempting to complete a PhD has been a great undertaking, and in completing this thesis I am nearing the end of an important chapter in my life. The years I have spent as a postgraduate researcher have probably been the happiest of my life, but at times the work involved has been tough, and without the support of people around me I certainly couldn't have made it this far. Almost everyone I have met and got to know during this period has touched my life in a positive way, but there are a few people in particular that I wish to thank.

Firstly, I would like to thank my supervisor James D. Mitchell, for his honesty and friendliness, and for the many hours he has spent correcting my work and making me a better mathematician. Secondly, I would like to thank my friend and office-mate Wilf Wilson, whose wonderful company has kept me from falling asleep through many weary afternoons of writing and coding. I am also indebted to the Engineering and Physical Sciences Research Council (EPSRC), whose generous grant has allowed me to pursue computational semigroup theory freely for the last four years.

Finally, I wish to thank Claire Young. She has been the most important part of my life throughout my postgraduate career, and her love and support during the tougher moments of this PhD have given me the motivation to overcome what sometimes felt like insurmountable obstacles.

Michael Torpey

St Andrews
July 2018

Having completed the final version of this thesis, I also wish to thank my examiners Martyn Quick and Wolfram Bentz, for the time and effort they spent reading my work, conducting my viva, and providing the detailed feedback that helped me improve this thesis to its current state.

Michael Torpey

St Andrews
February 2019

Chapter 1

Preliminaries

In this chapter, we will introduce various objects and results that are required to understand this thesis. For reference works relating to semigroup theory and to algebra in general, see [War90], [Pet84], and particularly [How95].

1.1 Basic notation

We should first mention some conventions adopted in this thesis where notation might differ from other authors. On the whole, care has been taken to deviate as little as possible from standard notation, but where ambiguity might arise, the following are the conventions that were chosen.

Maps are written on the right. Hence, if we have a function f that takes an input x, its output is written $(x)f$, with the parentheses sometimes omitted. If two maps are composed, they are written left-to-right. Hence, if we have another map g to be applied to the output of f, then their composition is written $f \circ g$ or simply fg, and we have $\big((x)f\big)g = (x)fg$.

Where we denote subsets, the symbol $\subseteq$ is used to denote containment with possible equality, while $\subset$ is used to denote strict containment. Hence $X \subset Y$ implies that X is not equal to Y, while $X \subseteq Y$ implies that X may be equal to Y or may contain only some of the elements of Y. In this way, the symbols are consistent with the symbols $<$ and $\leq$ for comparing numbers.

If E is an equivalence relation on a set X, and $x \in X$, then $[x]_E$ will denote the equivalence class of E containing the element x. That is,

$$[x]_E = \{y \in X : (x, y) \in E\}.$$

Where there is no risk of ambiguity, we will omit the subscript $_E$ and simply write $[x]$. We will denote by Δ_X the *diagonal relation* or *trivial equivalence* $\Delta_X = \{(x, x) : x \in X\}$, and we will denote by ∇_X the *universal relation* $\nabla_X = X \times X = \{(x, y) : x, y \in X\}$. Note that throughout this thesis, a *relation* on X is assumed to mean a binary relation – that is, a subset of $X \times X$.

The set of natural numbers will be denoted by $\mathbb{N}$, and will be equal to the set $\{1, 2, 3, \ldots\}$, excluding 0. For a given $n \in \mathbb{N}$ we will sometimes refer to the set $\{1, \ldots, n\}$ using the notation $\mathbf{n}$.

1.2 Semigroups

We start with an introduction to semigroups from a completely abstract point of view. Here we will define a semigroup and a few closely related concepts, and in later sections we will see examples of how semigroups arise (see Sections 1.7 and 1.11).

Definition 1.1. A **semigroup** is a non-empty set S together with a binary operation $\cdot : S \times S \to S$ such that

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

for all $x, y, z \in S$.

Note that some authors leave out the requirement that S must be non-empty, resulting in the empty semigroup $\varnothing$. This extra semigroup is of little theoretical interest, but requires awkward caveats to be added to various statements that only apply to a non-empty semigroup, adding unnecessary complication. Hence, we exclude the empty semigroup from our considerations, and we will require that all semigroups contain at least one element. Note also that the operation symbol $\cdot$ is often omitted where there is no risk of ambiguity, in the manner of multiplication.

A semigroup S may contain a few special elements: an **identity** is an element $e \in S$ such that $ex = xe = x$ for any $x \in S$; a **zero** is an element $0 \in S$ such that $0x = x0 = 0$ for any $x \in S$; an **idempotent** is an element $e \in S$ such that $ee = e$; and an element x has an **inverse** y if $xyx = x$ and $yxy = y$. All of these will be used later to define certain semigroups, or properties of semigroups. We will use the notation S^1 to denote the semigroup S with an identity 1 appended if S does not already have one. We will use the notation $S^{(1)}$ to denote the semigroup S with an extra identity 1 appended, whether S already has an identity or not.

We now define three important categories of semigroup, based on properties possessed by their elements.

Definition 1.2. A **monoid** M is a semigroup containing a distinguished element e such that

$$ex = xe = x$$

for all $x \in M$. The element e is called the *identity* of M.

Definition 1.3. An **inverse semigroup** is a semigroup S in which every element $x \in S$ has a unique inverse, i.e. a unique element $x^{-1} \in S$ such that $xx^{-1}x = x$ and $x^{-1}xx^{-1} = x^{-1}$.

Definition 1.4. A **group** G is a monoid in which every element $x \in G$ has a group inverse, i.e. an element $x^{-1} \in G$ such that $xx^{-1} = x^{-1}x = e$.

These three definitions represent important subcategories of semigroups, with group theory in particular being an important field in its own right. Inverse semigroups are not a core part of this thesis, but we will state a few elementary facts which will be required in Chapter 3:

Proposition 1.5. *Let S be an inverse semigroup, and let x be an element of S. The following hold:*

(i) *The elements $x^{-1}x$ and xx^{-1} are idempotent;*

(ii) *Every $\mathscr{L}$-class and every $\mathscr{R}$-class of S contains exactly one idempotent;*

where $\mathscr{L}$ and $\mathscr{R}$ are as defined in Section 1.9.

Proof. For (i), simply observe that $(x^{-1}x)(x^{-1}x) = (x^{-1}xx^{-1})x = x^{-1}x$ and $(xx^{-1})(xx^{-1}) = (xx^{-1}x)x^{-1} = xx^{-1}$. For (ii), see [How95, Theorem 5.1.1]. $\square$

As mentioned in the preface, groups are also an important part of any semigroup's structure – we will encounter a semigroup's *subgroups* as part of the study of its linked triples, for example. We define subgroups along with *submonoids* and *subsemigroups*, as follows.

Definition 1.6. Let $(S, *)$ be a semigroup, consisting of a set S together with an associative binary operation $* : S \times S \to S$. Let T be a subset of S, and let $*|_{T\times T}$ be the restriction of $*$ to $T \times T$. If $(T, *|_{T\times T})$ is a semigroup, then it is a **subsemigroup** of $(S, *)$.

Where the operations $*$ and $*|_{T\times T}$ are well-understood, we will simply write that T is a subsemigroup of S.

Similarly, if M is a monoid, then a subsemigroup of M which has the same identity is called a **submonoid** of M.

If a subsemigroup of S happens to be a group, we call it a **subgroup** of S. The subgroups of a semigroup are an important part of its structure, and will be discussed later in the context of Green's relations (see Proposition 1.55). We should also mention an important type of subgroup possessed by a group.

Definition 1.7. Let G be a group, and let H be a subgroup of G. We call H a **normal subgroup** of G if one, and hence all, of the following equivalent statements hold:

- $Hg = gH$ for all $g \in G$;
- $g^{-1}Hg = H$ for all $g \in G$;
- $g^{-1}hg \in H$ for all $g \in G$ and $h \in H$.

Trivially, we can see that any group G has both the trivial group $\{\mathrm{id}_G\}$ and the whole group G as normal subgroups. As will be seen later (Section 3.1.2) a group's normal subgroups correspond to its congruences, so these will come up at various times as important objects.

Definition 1.8. We can express a finite semigroup S using a **Cayley table** or **multiplication table**: a square $|S| \times |S|$ table with its rows and its columns labelled by all the elements of S. The cell in the row corresponding to an element x and the column corresponding to an element y contains the product xy. See Figure 1.9 for an example.

	a	b	c	d
a	a	b	c	d
b	b	a	c	d
c	c	d	c	d
d	d	c	c	d

Table 1.9: Cayley table of a semigroup with four elements.

We should define a few special semigroups which will be used later. Other important semigroups are described in Section 1.11.

Definition 1.10. We define three new types of semigroup as follows:

- a **zero semigroup** is a semigroup with a distinguished element 0 such that $ab = 0$ for all elements a and b in the semigroup;
- a **left zero semigroup** is a semigroup in which $ab = a$ for all elements a and b in the semigroup;
- a **right zero semigroup** is a semigroup in which $ab = b$ for all elements a and b in the semigroup.

We will sometimes refer to *the* zero semigroup of size n, which we will denote by $\mathcal{Z}_n$. This refers to the zero semigroup with elements $\{z, a_1, \dots a_{n-1}\}$, where z is the 0 element. Similarly we will sometimes refer to *the* left zero semigroup $\mathcal{LZ}_n$ and *the* right zero semigroup $\mathcal{RZ}_n$: these are the left and right zero semigroups respectively, of size n, with set of elements $\{a_1, \dots, a_n\}$.

1.3 Generators

It can be unwieldy or computationally costly to keep a list of all the elements of a semigroup. Instead, it is possible to specify a semigroup by storing only a small subset of its elements, known as *generators*, as follows.

Definition 1.11. Let S be a semigroup, and let X be a non-empty subset of S. The least subsemigroup of S containing all the elements of X is known as the semigroup **generated by** X, and is denoted by $\langle X \rangle$. We say that X is a set of **generators** for $\langle X \rangle$.

The above definition only makes sense when our semigroup $\langle X \rangle$ is defined as a subsemigroup of another semigroup S. A set of elements X that is not understood to lie inside a semigroup does not have a well-defined operation, so the concept of generating more elements does not make sense. However, most of the semigroups we will encounter will be comprised of elements that have a natural associative operation, and hence belong to an implicitly defined semigroup. For example, a set of transformations of degree n can generate a semigroup, because it is understood to be a subset of the full transformation monoid $\mathcal{T}_n$ (see Definition 1.62). Hence, we can talk about a set of transformations generating a semigroup, without explicitly stating that it is a subsemigroup of $\mathcal{T}_n$.

Aside from generators for a semigroup, we also have the concept of generators for a monoid, a group, and an inverse semigroup. These are defined analogously to the last definition, replacing the word "semigroup" with the appropriate structure. The notation $\langle X \rangle$ may be used for any of these. Note, however, that a generating set X for a monoid or a group can be empty, since there is a unique least submonoid or subgroup containing it – the trivial group containing just the identity.

A semigroup with a set of generators can be described using a *Cayley graph*, a directed graph with labelled edges that is defined as follows.

Definition 1.12. Let S be a semigroup, with a generating set X. The **right Cayley graph** of S with respect to X is the digraph-with-edge-labels Γ, which is described as follows:

- The vertices of Γ are the elements of S;
- For each pair $(s, x) \in S \times X$ there exists an edge from s to $s \cdot x$ labelled by x:

$$s \xrightarrow{x} s \cdot x$$

The **left Cayley graph** of S with respect to X is defined analogously, replacing $s \cdot x$ with $x \cdot s$.

Finally, it is worth noting that we can also use a set of elements to generate a normal subgroup of a group, using the following definition.

Definition 1.13. Let G be a group, and let X be a subset of G. The least normal subgroup of G containing all the elements of X is known as the **normal closure** of X, and is denoted by $\langle\langle X \rangle\rangle$.

We will see that this definition is particularly interesting in the study of congruences (see Section 1.5). A congruence on a group has classes equal to the cosets of a normal subgroup, and every congruence arises in this way. Hence, since normal subgroups are the group theory analogue of semigroup congruences, it follows that the taking of normal closures in a group is analogous to the use of generating pairs in a semigroup (see Section 1.6). This correspondence is explained in detail in Section 3.1.2.

1.4 Homomorphisms

Two semigroups can be related by a special kind of map from one to another: a *homomorphism* is a map from one semigroup to another that respects the semigroup operation, as follows.

Definition 1.14. Let S and T be semigroups. A semigroup **homomorphism** is a function $\phi : S \to T$ such that

$$(x)\phi \cdot (y)\phi = (xy)\phi,$$

for all $x, y \in S$.

Injective and surjective homomorphisms are special cases that will be important when we compare semigroups to each other. We give these homomorphisms names, as follows.

Definition 1.15. A semigroup **monomorphism** is a semigroup homomorphism which is injective (one-to-one). It is indicated on diagrams by a hooked arrow:

$$S \hookrightarrow T$$

Definition 1.16. A semigroup **epimorphism** is a semigroup homomorphism which is surjective (onto). It is indicated on diagrams by a double-headed arrow:

$$S \twoheadrightarrow T$$

Definition 1.17. A semigroup **isomorphism** is a semigroup homomorphism which is both injective (one-to-one) and surjective (onto). It is indicated on diagrams by a hooked double-headed arrow:

$$S \hookrightarrow\!\!\!\!\rightarrow T$$

Next we define some important attributes of a homomorphism.

Definition 1.18. The **kernel** $\ker \phi$ of a homomorphism $\phi : S \to T$ is the equivalence relation on S defined by the rule that $(a, b) \in \ker \phi$ if and only if

$$(a)\phi = (b)\phi,$$

for $a, b \in S$.

Definition 1.19. The **image** $\operatorname{im} \phi$ of a homomorphism $\phi : S \to T$ is $(S)\phi$, the set of elements $t \in T$ such that

$$(s)\phi = t$$

for some $s \in S$.

Monoid homomorphisms are defined analogously to semigroup homomorphisms. The definition is the same as Definition 1.14, replacing the word "semigroup" with "monoid", and with the additional requirement that ϕ must map the identity of S to the identity of T. If not specified, it is assumed that "homomorphism" refers to a semigroup homomorphism.

The first interesting result about homomorphisms will be the First Isomorphism Theorem (Theorem 1.26), which links homomorphisms to congruences.

1.5 Congruences

Congruences are the central topic of this thesis. We will describe a number of algorithms for congruences in Part I, and classify the congruences of several different semigroups in Part II. As seen in Chapter 3, there are many different ways to view congruences, but we will start with the original definition: as a special type of equivalence relation on a semigroup.

Definition 1.20. Let S be a semigroup, and let $\mathbf{R}$ be a relation on S. The relation $\mathbf{R}$ is:

- **left-compatible** if $(x, y) \in \mathbf{R}$ implies that $(ax, ay) \in \mathbf{R}$ for all $a \in S$;
- **right-compatible** if $(x, y) \in \mathbf{R}$ implies that $(xa, ya) \in \mathbf{R}$ for all $a \in S$;
- **compatible** if it is both left-compatible and right-compatible.

Definition 1.21. Let S be a semigroup, and let ρ be an equivalence relation on S. The relation ρ is:

- a **left congruence** if it is left-compatible;
- a **right congruence** if it is right-compatible;
- a **two-sided congruence** if it is compatible.

When we talk about a *congruence* without specifying that it is left or right, it is understood to be a two-sided congruence. An alternative definition of two-sided congruences is embodied in the following proposition.

Proposition 1.22. *Let $\mathbf{E}$ be an equivalence on a semigroup S. The equivalence $\mathbf{E}$ is a congruence on S if and only if*

$$(xs, yt) \in \mathbf{E}$$

for all pairs $(x, y), (s, t) \in \mathbf{E}$.

Proof. We prove the "only if" direction first, and then consider the "if" direction.

First, assume $\mathbf{E}$ is a congruence, containing pairs (x, y) and (s, t). Since $\mathbf{E}$ is a left congruence, we have $(xs, xt) \in \mathbf{E}$, and since it is a right congruence, we have $(xt, yt) \in \mathbf{E}$. Hence, by transitivity, $(xs, yt) \in \mathbf{E}$, as required.

For the converse, assume that $(xs, yt) \in \mathbf{E}$ for all pairs $(x, y), (s, t) \in \mathbf{E}$, and let $a \in S$ be arbitrary. Since $(a, a) \in \mathbf{E}$ by reflexivity, we must have $(xa, ya) \in \mathbf{E}$ by the assumption. Similarly, we must have $(ax, ay) \in \mathbf{E}$, by using (x, y) in place of (s, t). Hence $\mathbf{E}$ is a congruence, as required. $\square$

Congruences have an important property that allows new semigroups, known as *quotient semigroups*, to be made from old ones. We will state the definition of a quotient semigroup, and then show that the definition is well defined.

Definition 1.23. Let S be a semigroup, and let ρ be a congruence on S. The **quotient semigroup** S/ρ is the semigroup whose elements are the congruence classes of ρ, and whose operation $*$ is defined by

$$[a]_\rho * [b]_\rho = [ab]_\rho,$$

for $a, b \in S$.

Proposition 1.24. *A quotient semigroup is well-defined by Definition 1.23. That is to say, for two ρ-classes A and B, the value of $A * B$ is the same regardless of which representatives are chosen from the two classes.*

Proof. Let S and ρ be as in Definition 1.23, and let A and B be classes of ρ, with elements $a_1, a_2 \in A$ and $b_1, b_2 \in B$. The product $A * B$ is defined to be the class which contains the element $a_1 b_1$, but it is also defined as the class which contains the element $a_2 b_2$. For this definition to be consistent, we need to prove that $a_1 b_1$ and $a_2 b_2$ are in the same ρ-class.

To prove this, we observe that $(a_1, a_2) \in \rho$ and $(b_1, b_2) \in \rho$. Hence, by Proposition 1.22, $(a_1 b_1, a_2 b_2) \in \rho$, so the two representatives are in the same class, as required. Hence $A * B$ is well-defined. $\square$

Note that Definition 1.23 does not apply to left and right congruences, which do not generally satisfy the condition stated in Proposition 1.22. A quotient semigroup can only be taken using a two-sided congruence.

Definition 1.25. Let S be a semigroup, and let ρ be a congruence on S. The **natural homomorphism** $\pi_\rho : S \to S/\rho$ is the map which takes an element of S to its ρ-class:

$$\pi_\rho : x \mapsto [x]_\rho.$$

It is denoted simply by π where there is no risk of ambiguity.

Congruences have long been an important area of study in semigroup theory. Perhaps the most important feature of two-sided congruences is that they determine the homomorphic images of a semigroup, and therefore describe an important part of a semigroup's structure. Consider the following theorem.

Theorem 1.26 (First isomorphism theorem)**.** *Let S and T be semigroups, and let ϕ be a homomorphism from S to T. Then the kernel of ϕ is a congruence on S, and the image of ϕ is isomorphic to the quotient semigroup $S/\ker\phi$.*

$$\begin{tikzcd}
S \arrow[r, "\phi"] \arrow[d, two heads, "\pi"'] & T \\
S/\ker\phi \arrow[ur, hook, dashed, "\bar{\phi}"'] &
\end{tikzcd}$$

Figure 1.27: Illustration of Theorem 1.26, where $\pi : S \to S/\ker\phi$ is the natural homomorphism, and $\bar{\phi} : S/\ker\phi \to T$ acts as an isomorphism between $S/\ker\phi$ and $\operatorname{im}\phi$.

The congruences of a semigroup are related to each other in some interesting ways. Let ρ and σ be congruences on a semigroup S. The congruence ρ may be contained in σ as a subrelation, as we can see when we consider the congruences as sets of pairs: intuitively, we have $\rho \subseteq \sigma$ if ρ is a refinement of σ, with some of its congruence classes broken down into smaller pieces. The intersection $\rho \cap \sigma$ is also a congruence on S, as can be seen from the definition of a congruence. Less obvious is that their *join* $\rho \vee \sigma$ (the least equivalence containing both ρ and σ) is also a congruence on S, as shown in [How95, §1.5]. Hence we have a relation $\subseteq$ and two operations $\cap$ and $\vee$ which apply to the set of congruences on S. We will soon find that the set of congruences forms a special structure together with $(\subseteq, \cap, \vee)$. We will first define this structure, and then go on to explain how it can be viewed as a semigroup.

Recall that a *poset* is a set X together with a partial order $\leq$ – that is, a relation $\leq$ on X which is reflexive, anti-symmetric and transitive. If X is a poset, and $Y \subseteq X$, then $x \in X$ is called an *upper bound* for Y if $y \leq x$ for all $y \in Y$; similarly, x is a *lower bound* for Y if $x \leq y$ for all $y \in Y$. An upper bound for Y is called the *least upper bound* or *join* if it lies below all other upper bounds with respect to $\leq$; similarly, a lower bound is called the *greatest lower bound* or *meet* if it lies above all other lower bounds.

Definition 1.28. A **lattice** is a poset $(X, \leq)$ such that any two elements of X have a greatest lower bound and a least upper bound.

If we have two elements x_1 and x_2 in X, we write their greatest lower bound (meet) using the notation $x_1 \wedge x_2$ and their least upper bound (join) as $x_1 \vee x_2$. We can now view a lattice as a semigroup in two different ways: the set X together with the binary operation of meet $(\wedge)$ or the binary operation of join $(\vee)$. In fact, both of these semigroups are commutative, and all their elements are idempotents [How95, Proposition 1.3.2], and so any finite subset $Y = \{y_1, \dots y_n\}$ has a meet $\bigwedge Y = y_1 \wedge \dots \wedge y_n$ and a join $\bigvee Y = y_1 \vee \dots \vee y_n$ that do not depend on any ordering of the set.

We can now describe how this definition applies to the congruences of a semigroup.

Proposition 1.29 ([How95, §1.5]). *The congruences on a semigroup S form a lattice under the partial order of containment. The meet operation is intersection ($\cap$), while the join operation is the usual join operation on equivalence relations ($\vee$).*

The fact that congruences lie in a lattice, and therefore form a semigroup, allows us to view congruences as semigroup elements in their own right. In Chapter 4 in particular we will generate a congruence lattice using just its principal congruences as generators and join ($\vee$) as a semigroup operation (see Proposition 1.42), and at various points in this thesis we will show the Hasse diagrams of congruence lattices, viewing them as partial orders (for some examples, see Figures 4.3 and 5.24).

The idea of a congruence fits closely with the concept of a semigroup *presentation*, which we will introduce in Section 1.7, after the prerequisite concept of *generating pairs*.

1.6 Generating pairs

We now describe a concept key to Chapter 2 as well as to semigroup presentations, that of *generating pairs*. Much of the description in this section is adapted from a previous thesis, [Tor14b], which itself closely follows [How95, §1.4–1.5].

Definition 1.30. Let S be a semigroup and let $\mathbf{R}$ be a subset of $S \times S$.

- The **equivalence generated by R** is the least equivalence relation (with respect to containment) which contains $\mathbf{R}$ as a subset.
- The **left congruence generated by R** is the least left congruence (with respect to containment) which contains $\mathbf{R}$ as a subset.
- The **right congruence generated by R** is the least right congruence (with respect to containment) which contains $\mathbf{R}$ as a subset.
- The **congruence generated by R** is the least congruence (with respect to containment) which contains $\mathbf{R}$ as a subset.

Chapter 2 deals in detail with congruences specified by generating pairs. We now present some theory relating to generating pairs, in order to inform discussions later.

We first need to establish a few definitions. Let S be a semigroup, and let $\mathbf{R}$ be a relation on S. We define

$$\mathbf{R}^{-1} = \{(x, y) \in S \times S \mid (y, x) \in \mathbf{R}\},$$

so that $\mathbf{R}^{-1}$ is a copy of $\mathbf{R}$ but with the entries in each pair swapped. Next, let $\circ$ be the operation of concatenation, so that for two relations $\mathbf{R}_1$ and $\mathbf{R}_2$ on S,

$$\mathbf{R}_1 \circ \mathbf{R}_2 = \{(x, y) \in S \times S \mid \exists z \in S : (x, z) \in \mathbf{R}_1, (z, y) \in \mathbf{R}_2\},$$

and for $n \in \mathbb{N}$ let

$$\mathbf{R}^n = \underbrace{\mathbf{R} \circ \cdots \circ \mathbf{R}}_{n \text{ times}}.$$

Definition 1.31. The **transitive closure** $\mathbf{R}^\infty$ of a relation $\mathbf{R}$ is the relation given by

$$\mathbf{R}^\infty = \bigcup_{n \in \mathbb{N}} \mathbf{R}^n$$

The transitive closure $\mathbf{R}^\infty$ of $\mathbf{R}$ is the least transitive relation on S containing $\mathbf{R}$ [Tor14b, Lemma 2.3]. This allows us to give a useful description of the equivalence relation generated by $\mathbf{R}$.

Definition 1.32. For a relation $\mathbf{R}$ on a semigroup S, we define $\mathbf{R}^e$ as the relation $\big(\mathbf{R} \cup \mathbf{R}^{-1} \cup \Delta_S\big)^\infty$.

Lemma 1.33. *The relation $\mathbf{R}^e$ is the smallest equivalence on S that contains $\mathbf{R}$ as a subset.*

Proof. Clearly $\mathbf{R} \subseteq \mathbf{R}^e$.

We will prove that $\mathbf{R}^e$ is an equivalence relation, and then go on to prove that there is no smaller equivalence relation containing $\mathbf{R}$.

Let $\mathbf{Q} = \mathbf{R} \cup \mathbf{R}^{-1} \cup \Delta_S$, so that $\mathbf{R}^e = \mathbf{Q}^\infty$. Since Δ_S contains all the pairs necessary for reflexivity, we know that $\mathbf{Q}$ is reflexive, and therefore $\mathbf{Q}^\infty$ is reflexive and transitive.

To show symmetry, observe that $(x, y) \in \mathbf{R}$ if and only if $(y, x) \in \mathbf{R}^{-1}$, and that $(x, y) \in \Delta_S$ if and only if $x = y$. $\mathbf{Q}$ is therefore certainly symmetric, and

$$\mathbf{Q}^n = (\mathbf{Q}^{-1})^n = (\mathbf{Q}^n)^{-1}$$

for any $n \in \mathbf{N}$, and so $\mathbf{Q}^n$ is symmetric.

Now let $(x, y) \in \mathbf{R}^e$. For some $n \in \mathbb{N}$, we have $(x, y) \in \mathbf{Q}^n$. By the symmetry of $\mathbf{Q}^n$,

$$(y, x) \in \mathbf{Q}^n \subseteq \mathbf{Q}^\infty = \mathbf{R}^e,$$

and so $\mathbf{R}^e$ is symmetric. Hence $\mathbf{R}^e$ is an equivalence.

Now to show that $\mathbf{R}^e$ is the *least* such equivalence, consider any equivalence $\mathbf{E}$ on S such that $\mathbf{R} \subseteq \mathbf{E}$. Since $\mathbf{E}$ is reflexive, we know that $\Delta_S \subseteq \mathbf{E}$, and since $\mathbf{E}$ is symmetric and contains $\mathbf{R}$, we know that $\mathbf{R}^{-1} \subseteq \mathbf{E}$. Hence

$$\mathbf{Q} = \mathbf{R} \cup \mathbf{R}^{-1} \cup \Delta_S \subseteq \mathbf{E}.$$

Finally, since $\mathbf{E}$ is transitive and contains $\mathbf{Q}$, we know that $\mathbf{Q}^\infty \subseteq \mathbf{E}$. Hence $\mathbf{R}^e$ is contained in $\mathbf{E}$, and so is no larger than any equivalence on S. ☐

Definition 1.34. For a relation $\mathbf{R}$ on a semigroup S, we define three relations:

(i) $\mathbf{R}^c = \big\{(xay, xby) \bigm| (a, b) \in \mathbf{R},\ x, y \in S^1\big\}$;

(ii) $\mathbf{R}^l = \big\{(xa, xb) \bigm| (a, b) \in \mathbf{R},\ x \in S^1\big\}$;

(iii) $\mathbf{R}^r = \big\{(ay, by) \bigm| (a, b) \in \mathbf{R},\ y \in S^1\big\}$.

Lemma 1.35. *Let $\mathbf{R}$ be a relation on a semigroup S. The following hold:*

(i) *$\mathbf{R}^c$ is the smallest compatible relation on S containing $\mathbf{R}$;*

(ii) *$\mathbf{R}^l$ is the smallest left-compatible relation on S containing $\mathbf{R}$;*

(iii) *$\mathbf{R}^r$ is the smallest right-compatible relation on S containing $\mathbf{R}$.*

Proof. We prove the statement for $\mathbf{R}^c$, and note that the proofs for $\mathbf{R}^l$ and $\mathbf{R}^r$ are very similar. $\mathbf{R}^c$ certainly contains $\mathbf{R}$ – all the elements of $\mathbf{R}$ are encountered in the case that $x = y = 1$.

Let us show first that $\mathbf{R}^c$ is compatible. Let $(u, v) \in \mathbf{R}^c$ and let $w \in S$. Now there must exist $a, b \in S$ and $x, y \in S^1$ such that $u = xay$, $v = xby$, and $(a, b) \in \mathbf{R}$. Hence $wu = wx \cdot a \cdot y$ and $wv = wx \cdot b \cdot y$, and $wx \in S^1$, so $(wu, wv) \in \mathbf{R}^c$ and $\mathbf{R}^c$ is left-compatible. Similarly, $uw = x \cdot a \cdot yw$ and $vw = x \cdot b \cdot yw$, and $yw \in S^1$, so $(uw, vw) \in \mathbf{R}^c$ and $\mathbf{R}^c$ is right-compatible.

Finally we need to show that there is no compatible relation smaller than $\mathbf{R}^c$ which contains $\mathbf{R}$. For this purpose, let $\mathbf{C}$ be a compatible relation on S such that $\mathbf{R} \subseteq \mathbf{C}$. Now for any $(a, b) \in \mathbf{R}$ and $x, y \in S^1$, we must have $(xay, xby) \in \mathbf{C}$ by the definition of compatibility. Every element of $\mathbf{R}^c$ has this form, hence $\mathbf{R}^c \subseteq \mathbf{C}$. [How95, §1.5] $\square$

These three relations $\mathbf{R}^c$, $\mathbf{R}^l$ and $\mathbf{R}^r$ have some properties which will be useful later. These properties make up the following lemmas.

Lemma 1.36. *Let $\mathbf{R}$ be a relation on a semigroup S. The following hold:*

(i) $(\mathbf{R}^{-1})^c = (\mathbf{R}^c)^{-1}$;

(ii) $(\mathbf{R}^{-1})^l = (\mathbf{R}^l)^{-1}$;

(iii) $(\mathbf{R}^{-1})^r = (\mathbf{R}^r)^{-1}$.

Proof. Let $\mathbf{R}$ be a relation on a semigroup S. $\mathbf{R}^{-1} = \{(a, b) \mid (b, a) \in \mathbf{R}\}$, so

$$(\mathbf{R}^{-1})^c = \{(xay, xby) \mid x, y \in S^1, (b, a) \in \mathbf{R}\}.$$

The inverse of this last expression is

$$\{(xay, xby) \mid x, y \in S^1, (a, b) \in \mathbf{R}\},$$

which is equal to $\mathbf{R}^c$. Now $\left((\mathbf{R}^{-1})^c\right)^{-1} = \mathbf{R}^c$, which is equivalent to what we wanted.

A similar argument holds for both $\mathbf{R}^l$ and $\mathbf{R}^r$ $\square$

Lemma 1.37. *Let $\mathbf{A}$ and $\mathbf{B}$ be relations on a semigroup S. If $\mathbf{A} \subseteq \mathbf{B}$, then $\mathbf{A}^c \subseteq \mathbf{B}^c$, $\mathbf{A}^l \subseteq \mathbf{B}^l$, and $\mathbf{A}^r \subseteq \mathbf{B}^r$.*

Proof. Let $\mathbf{A} \subseteq \mathbf{B}$, and let (xay, xby) be an arbitrary element of $\mathbf{A}^c$ where $(a, b) \in \mathbf{A}$ and $x, y \in S^1$. Since $\mathbf{A} \subseteq \mathbf{B}$, we have that $(a, b) \in \mathbf{B}$, and hence also that $(xay, xby) \in \mathbf{B}^c$.

A similar statement holds for $\mathbf{A}^l \subseteq \mathbf{B}^l$ and $\mathbf{A}^r \subseteq \mathbf{B}^r$. $\square$

Lemma 1.38. *If $\mathbf{R}$ is a (left/right) compatible relation, then $\mathbf{R}^n$ is also (left/right) compatible, for all $n \in \mathbb{N}$.*

Proof. Let $\mathbf{R}$ be a compatible relation on a semigroup S, and let $n \in \mathbb{N}$. Now let $(a, b) \in \mathbf{R}^n$, and $x \in S$. Hence there exist $c_1, c_2, \ldots, c_n, c_{n+1} \in S$ such that $a = c_1$, $b = c_{n+1}$, and

$$(c_1, c_2), (c_2, c_3), \ldots, (c_n, c_{n+1}) \in \mathbf{R}.$$

Since $\mathbf{R}$ is left-compatible,

$$(xc_1, xc_2), (xc_2, xc_3), \ldots, (xc_n, xc_{n+1}) \in \mathbf{R},$$

and since $\mathbf{R}$ is right-compatible,

$$(c_1x, c_2x), (c_2x, c_3x), \dots, (c_nx, c_{n+1}x) \in \mathbf{R},$$

and therefore $(xa, xb) \in \mathbf{R}^n$ and $(ax, bx) \in \mathbf{R}^n$, so $\mathbf{R}^n$ is compatible. [How95, §1.5] A similar argument holds for left and right compatibility. $\square$

These lemmas now enable us to give a theorem characterising the congruence, left congruence, and right congruence generated by $\mathbf{R}$.

Theorem 1.39. *Let $\mathbf{R}$ be a relation on a semigroup S. The following hold:*

(i) $\mathbf{R}^\sharp$*, the least congruence on S which contains $\mathbf{R}$, is equal to $(\mathbf{R}^c)^e$;*

(ii) $\mathbf{R}^\triangleleft$*, the least left congruence on S which contains $\mathbf{R}$, is equal to $(\mathbf{R}^l)^e$;*

(iii) $\mathbf{R}^\triangleright$*, the least right congruence on S which contains $\mathbf{R}$, is equal to $(\mathbf{R}^r)^e$.*

Proof. Since $\mathbf{R}^c$ is a relation, it follows from Lemma 1.33 that $(\mathbf{R}^c)^e$ is an equivalence, and it certainly contains $\mathbf{R}$. To show that it is a congruence, we now only need to show that it is compatible.

By Definition 1.32, $(\mathbf{R}^c)^e = \mathbf{Q}^\infty$, where

$$\mathbf{Q} = \mathbf{R}^c \cup (\mathbf{R}^c)^{-1} \cup \Delta_S.$$

Lemma 1.36 gives us that $(\mathbf{R}^c)^{-1} = (\mathbf{R}^{-1})^c$, and we know $\Delta_S = {\Delta_S}^c$, so

$$\mathbf{Q} = \mathbf{R}^c \cup (\mathbf{R}^{-1})^c \cup {\Delta_S}^c.$$

Now, we can see directly from Definition 1.34 that $R_1^c \cup R_2^c \cup R_3^c = (R_1 \cup R_2 \cup R_3)^c$ for any relations R_1, R_2, R_3 on S. Hence we can conclude that

$$\mathbf{Q} = (\mathbf{R} \cup \mathbf{R}^{-1} \cup \Delta_S)^c.$$

Hence by Lemma 1.35, $\mathbf{Q}$ is a compatible relation.

Let $a \in S$ and let $(x, y) \in (\mathbf{R}^c)^e = \mathbf{Q}^\infty$. By Definition 1.31, $(x, y) \in \mathbf{Q}^n$ for some $n \in \mathbb{N}$, and by Lemma 1.38 we know that $\mathbf{Q}^n$ is compatible. Hence

$$(ax, ay), (xa, ya) \in \mathbf{Q}^n \subseteq \mathbf{Q}^\infty = (\mathbf{R}^c)^e,$$

and so $(\mathbf{R}^c)^e$ is a congruence.

All that remains is to show that there is no congruence containing $\mathbf{R}$ which is smaller than $(\mathbf{R}^c)^e$. Let ρ be a congruence containing $\mathbf{R}$. Since ρ is compatible, $\rho^c = \rho$ by Lemma 1.35; and since $\mathbf{R} \subseteq \rho$, by Lemma 1.37, $\mathbf{R}^c \subseteq \rho^c$. So we have

$$\mathbf{R}^c \subseteq \rho.$$

Finally, since ρ is an equivalence containing $\mathbf{R}^c$, we know from Lemma 1.33 that $(\mathbf{R}^c)^e \subseteq \rho$, so $(\mathbf{R}^c)^e$ is the smallest congruence on S containing $\mathbf{R}$. [How95, §1.5]

A similar argument holds for $\mathbf{R}^\triangleleft$ and $\mathbf{R}^\triangleright$. $\square$

We will make another definition, which can be seen as the opposite of the $\sharp$ (sharp) operator, hence the musical notation $\flat$ (flat).

Definition 1.40. Let $\mathbf{E}$ be an equivalence on a semigroup S. We denote by $\mathbf{E}^\flat$ the greatest congruence contained in $\mathbf{E}$.

Note that the above definition is not well-defined for a generic relation $\mathbf{R}$, but only for an equivalence $\mathbf{E}$. This is because $\mathbf{R}$ is not guaranteed to contain any congruence at all, whereas $\mathbf{E}$ must always contain the trivial congruence Δ_S. To see that $\mathbf{E}^\flat$ is well-defined, see [How95, Proposition 1.5.10], which uses the characterisation

$$\mathbf{E}^\flat = \left\{(a, b) \in S \times S : (\forall x, y \in S^1)\ (xay, xby) \in \mathbf{E}\right\},$$

and shows that $\mathbf{E}^\flat$ is a congruence contained in $\mathbf{E}$, and that any congruence contained in $\mathbf{E}$ is also contained in $\mathbf{E}^\flat$.

Now that we understand the concept of generating pairs, we can define a *principal congruence*, a concept related to that of a principal ideal.

Definition 1.41. A congruence is **principal** if it is generated by a single pair. If the pair is (x, y) then we may write the principal congruence as $(x, y)^\sharp$.

The principal congruences of a finite semigroup generate all its congruences, as shown in the following proposition. Note that the join of two congruences $X^\sharp$ and $Y^\sharp$ is equal to $(X \cup Y)^\sharp$.

Proposition 1.42. *Let S be a finite semigroup. The semigroup $(\mathbf{C}, \vee)$ consisting of the congruences on S under the join operation (see Proposition 1.29) is generated by the subset $\mathbf{P} \subseteq \mathbf{C}$ consisting of the principal congruences.*

Proof. Let ρ be a congruence on S, so $\rho \in \mathbf{C}$. We can generate ρ using a set of generating pairs $X \subseteq \rho$, choosing $X = \rho$ if necessary. Each pair $(x, y) \in X$ generates a principal congruence $(x, y)^\sharp$. The join of all such principal congruences is equal to $X^\sharp$, which is equal to ρ. Hence any congruence is the join of a set of principal congruences, and so the principal congruences generate all the congruences under the join operation. $\square$

1.7 Presentations

We have now encountered two important concepts for congruences – a congruence can be defined by generating pairs, and a quotient semigroup is defined by a congruence. We can combine these two concepts to give a very general way of describing semigroups: presentations. First we require the definition of a *free semigroup*.

Definition 1.43. Let X be a set. The **free monoid** over X is denoted by X^*, and consists of all finite sequences of elements in X, with the operation of concatenation. If X is non-empty, then the **free semigroup** X^+ is the subsemigroup of X^* consisting of sequences of length at least 1.

When we consider free semigroups and monoids, the set X is usually referred to as an **alphabet**, its elements as **letters**, and sequences of letters as **words**.

The justification for the use of the name "free" comes from category theory. We can see that the free semigroup X^+ has the following property, an alternative formulation of "free" given in [How95, §1.6]:

Proposition 1.44. *Let X be a non-empty set. The following hold:*

(i) *there is a map $\alpha : X \to X^+$;*

(ii) *for every semigroup S and every map $\phi : X \to S$ there exists a unique homomorphism $\psi : X^+ \to S$ such that $\phi = \alpha\psi$.*

$$\begin{tikzcd} X \arrow[r, "\alpha"] \arrow[d, "\phi"'] & X^+ \arrow[dl, dashed, "\exists!\psi"] \\ S & \end{tikzcd}$$

Figure 1.45: Commutative diagram illustrating Proposition 1.44.

Proof. We can choose α to be the obvious embedding which takes a letter x in X to the corresponding word x of length 1 in X^+. Then, for any S and $\phi : X \to S$ we can define $\psi : X^+ \to S$ by

$$(x_1 x_2 \ldots x_n)\psi = (x_1)\phi(x_2)\phi \ldots (x_n)\phi,$$

for $x_1, x_2, \ldots, x_n \in X$. This clearly satisfies $\phi = \alpha\psi$, and it is certainly a homomorphism, since

$$\begin{aligned} (x_1 \ldots x_n)\psi \cdot (y_1 \ldots y_m)\psi &= (x_1)\phi \ldots (x_n)\phi \cdot (y_1)\phi \ldots (y_m)\phi \\ &= (x_1 \ldots x_n y_1 \ldots y_m)\psi \end{aligned}$$

for $x_1, \ldots, x_n, y_1, \ldots, y_m \in X$. For uniqueness, let ψ' be any homomorphism $X^+ \to S$ such that $\phi = \alpha\psi'$. By this condition, we have

$$(x_1)\phi = (x_1)\alpha\psi' = (x_1)\psi',$$

for any $x_1 \in X$; and since ψ' is a homomorphism, this gives us

$$(x_1 x_2 \ldots x_n)\psi' = (x_1)\psi'(x_2)\psi' \ldots (x_n)\psi' = (x_1)\phi(x_2)\phi \ldots (x_n)\phi,$$

for $x_1, x_2, \ldots, x_n \in X$. This shows that $\psi' = \psi$, and so ψ is unique. ☐

We can now define semigroup presentations, a useful method of describing a semigroup which will be encountered many times in this thesis, particularly in Chapter 2. We will give the definition, and then discuss how the definition is used to describe a semigroup.

Definition 1.46. A **semigroup presentation** is a pair $\mathfrak{P} = (X, R)$ consisting of a set X and a set of pairs $R \subseteq X^+ \times X^+$. A semigroup is **defined by** the presentation $\mathfrak{P}$ if it is isomorphic to $X^+/R^\sharp$, i.e. the quotient of the free semigroup X^+ by the least congruence containing all the pairs in R.

We will normally write a presentation (X, R) using the notation $\langle X \mid R \rangle$. Furthermore, if we refer to explicit pairs with this notation, we will write a pair (a, b) as $a = b$, and we will omit the braces $\{\}$ from both sets. Example 1.49 demonstrates this notation.

Definition 1.47. A semigroup presentation $\langle\, X \mid R \,\rangle$ is **finite** if X and R are finite. A semigroup is **finitely presented** if there exists some finite presentation that defines it, i.e. if it is isomorphic to $X^+/R^\sharp$ for some finite presentation $\langle\, X \mid R \,\rangle$.

The exact wording used to talk about a semigroup S defined by a presentation $\langle\, X \mid R \,\rangle$ varies. Some sources view $\langle\, X \mid R \,\rangle$ as a semigroup in its own right: they might describe S as "isomorphic to" $\langle\, X \mid R \,\rangle$, or they might even say S "equals" $\langle\, X \mid R \,\rangle$. We will opt for the more careful language which separates a semigroup from its presentation: S is **defined by** or **presented by** $\langle\, X \mid R \,\rangle$, if and only if it is isomorphic to $X^+/R^\sharp$.

If S is presented by $\langle\, X \mid R \,\rangle$, there is an epimorphism from X^+ to S: if π is the natural homomorphism from X^+ to the quotient semigroup $X^+/R^\sharp$ (see Definition 1.25) and ι is an isomorphism from $X^+/R^\sharp$ to S, then $\pi\iota$ is an epimorphism from X^+ to S, which assigns each word in the generators to an element of the semigroup S. If $w \in X^+$ and $s \in S$ are such that $(w)\pi\iota = s$, then we say that the word w **represents** the element s, or that the element s can be **factorised** to the word w.

$$\begin{tikzcd} X^+ \arrow[r, two heads, "\pi"] & \frac{X^+}{R^\sharp} \arrow[r, hook, two heads, "\iota"] & S \end{tikzcd}$$

Figure 1.48: How a word from X^+ represents an element in S.

Semigroups are not uniquely defined by presentations: two different presentations may define the same semigroup, and those two presentations may not look similar at all. Consider the following example.

Example 1.49. The semigroup presentation $\left\langle\, a \,\middle|\, a = a^{10} \,\right\rangle$ defines the cyclic group C_9: there are 9 elements which can be represented by the words

$$\{a, a^2, a^3, a^4, a^5, a^6, a^7, a^8, a^9\}.$$

However, C_9 is also presented by $\left\langle\, b, c \,\middle|\, b = c^3,\ bc = cb,\ c = c^2b^2c^2 \,\right\rangle$, and an equivalence between the two presentations can be defined by identifying a with c. This second presentation, though it is more complicated to describe, allows us to use shorter words to describe many elements: the elements of C_9 are represented by the set of words $\{c, c^2, b, bc, bc^2, b^2, b^2c, b^2c^2\}$.

1.8 Ideals

Another semigroup-related object we should describe is an *ideal*, a particular type of subsemigroup linked to a semigroup's congruences and Green's relations.

Definition 1.50. An **ideal** of a semigroup S is a non-empty subset $I \subseteq S$ such that is and si are both in I, for all $i \in I$ and $s \in S$.

We can think of an ideal as a set from which it is impossible to escape by left- or right-multiplying. Ideals appear on *eggbox diagrams* as downward-closed unions of $\mathscr{D}$-classes (see Section 1.9). If a semigroup S contains a zero element, then $\{0\}$ is an ideal, as is the whole semigroup S. A group G has no ideals other than the whole of G, since it has the cancellative property that any element x can be transformed into any other element y by right-multiplying by $x^{-1}y$.

We can generate an ideal using an *ideal generating set*, defined in much the same way as Definition 1.11: if X is a subset of a semigroup S, then the least ideal containing all the elements of X is called the ideal **generated** by X, and is equal to the set S^1XS^1. An ideal is called **principal** if it is generated by a single element – a principal ideal can be written S^1xS^1 for some $x \in S$.

An ideal gives rise to a special congruence, known as a *Rees congruence*, defined in the following way.

Definition 1.51. Let S be a semigroup with an ideal I. The congruence

$$\rho_I = \nabla_I \cup \Delta_S = \{(x, y) \in S \times S : x = y \text{ or } x, y \in I\}$$

is known as the **Rees congruence** of I.

Rees congruences will be mentioned at various times later on (for example, see Section 3.1.5 or Table 6.28). The quotient semigroup S/ρ_I is typically denoted by S/I.

1.9 Green's relations

A critically important feature of a semigroup is its Green's relations. First described by Green in 1951 [Gre51], a semigroup's Green's relations reveal a great deal of information about its multiplication, its ideals, its maximal subgroups and its congruences. We will define five relations $\mathscr{L}$, $\mathscr{R}$, $\mathscr{H}$, $\mathscr{D}$ and $\mathscr{J}$, and explain how they are linked to each other and some other features of a semigroup.

Definition 1.52. Let S be a semigroup. We define five relations $\mathscr{L}$, $\mathscr{R}$, $\mathscr{H}$, $\mathscr{D}$ and $\mathscr{J}$ on S as follows:

- $x \mathrel{\mathscr{L}} y$ if and only if $S^1x = S^1y$, i.e. $ax = y$ and $by = x$ for some $a, b \in S^1$;
- $x \mathrel{\mathscr{R}} y$ if and only if $xS^1 = yS^1$, i.e. $xa = y$ and $yb = x$ for some $a, b \in S^1$;
- $x \mathrel{\mathscr{H}} y$ if and only if $x \mathrel{\mathscr{L}} y$ and $x \mathrel{\mathscr{R}} y$;
- $x \mathrel{\mathscr{D}} y$ if and only if there exists some $z \in S$ such that $x \mathrel{\mathscr{L}} z \mathrel{\mathscr{R}} y$;
- $x \mathrel{\mathscr{J}} y$ if and only if $S^1xS^1 = S^1yS^1$, i.e. if x and y generate the same ideal of S;

for all $x, y \in S$.

A few features are fairly obvious straight from these definitions. We can see that $\mathscr{L}$, $\mathscr{R}$ and $\mathscr{J}$ are equivalences; it is also fairly obvious that $\mathscr{H} = \mathscr{L} \cap \mathscr{R}$, and hence that $\mathscr{H}$ is an equivalence. Finally, we can establish that $\mathscr{D}$ is an equivalence by the fact that $\mathscr{D} = \mathscr{L} \vee \mathscr{R}$ [How95, §2.1]. It is also fairly obvious that $\mathscr{L} \subseteq \mathscr{D}$, $\mathscr{R} \subseteq \mathscr{D}$, and $\mathscr{D} \subseteq \mathscr{J}$. Less obvious is the highly useful fact that $\mathscr{D} = \mathscr{J}$ if S is finite [How95, §2.1]. These containments are shown in Figure 1.53.

Proposition 1.54. *Let S be a semigroup. The relation $\mathscr{L}$ is a right congruence on S, and the relation $\mathscr{R}$ is a left congruence on S.*

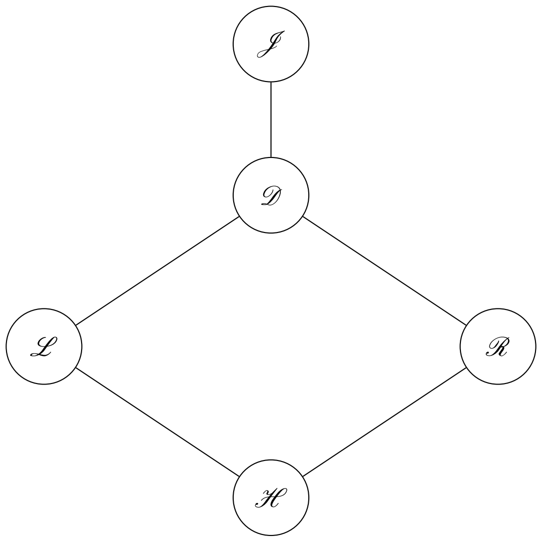


Figure 1.53: Hasse diagram of Green's relations under containment. Note that $\mathscr{D} = \mathscr{J}$ in the finite case.

Proof. Let $(x, y) \in \mathscr{L}$, and let $s \in S$. From Definition 1.52 we have $a, b \in S^1$ such that $ax = y$ and $by = x$. We have $xs = bys$ and $ys = axs$, so $(xs, ys) \in \mathscr{L}$, and $\mathscr{L}$ is a right congruence. By a similar argument, $\mathscr{R}$ is a left congruence. $\square$

The $\mathscr{J}$-classes of a semigroup are arranged in a natural partial order by their corresponding principal ideals, as follows. Let S be a semigroup, let a and b be elements of S^1, and let their $\mathscr{J}$-classes be denoted J_a and J_b. Just as $J_a = J_b$ if and only if $S^1aS^1 = S^1bS^1$, we say that $J_a \leq J_b$ if and only if $S^1aS^1 \subseteq S^1bS^1$.

The $\mathscr{H}$-classes of a semigroup have an interesting property which allow us to apply group theory to semigroups. Consider the following proposition.

Proposition 1.55 ([How95, Theorem 2.2.5])**.** *Let H be an $\mathscr{H}$-class of a semigroup S. Either H is a group, or $ab \notin H$ for all $a, b \in H$.*

This allows us to split the $\mathscr{H}$-classes of a semigroup into two categories: *group $\mathscr{H}$-classes* and *non-group $\mathscr{H}$-classes*.

We can display a finite semigroup's Green's relations pictorially using an **eggbox diagram**, which is constructed in the following way. First, note that $\mathscr{D} = \mathscr{J}$ for a finite semigroup; we break the semigroup into $\mathscr{D}$-classes and draw a box for each $\mathscr{D}$-class, arranged as a Hasse diagram according to the partial order of $\mathscr{J}$-classes described above. Now, since $\mathscr{L} \subseteq \mathscr{D}$ and $\mathscr{R} \subseteq \mathscr{D}$, we can break up a $\mathscr{D}$-class in two different ways. We split the box into rows representing its $\mathscr{L}$-classes and into columns representing its $\mathscr{R}$-classes. Now each cell in the box represents the intersection of an $\mathscr{L}$-class L with an $\mathscr{R}$-class R. Since L and R are in the same $\mathscr{D}$-class, $L \cap R$ must be non-empty (there must be some $z \in L \cap R$ linking each pair $(x, y) \in L \times R$, as in Definition 1.52). Since $\mathscr{H} = \mathscr{L} \cap \mathscr{R}$, $L \cap R$ is an $\mathscr{H}$-class. Hence, we have a diagram in which outer boxes represent $\mathscr{D}$-classes, rows represent $\mathscr{L}$-classes, columns represent $\mathscr{R}$-classes, and cells represent $\mathscr{H}$-classes. Finally, we highlight each group $\mathscr{H}$-class on the diagram: we shade it in, and mark it with a $*$ symbol or a symbol representing the isomorphism class of the group. An example of an eggbox diagram is shown in Figure 1.56.

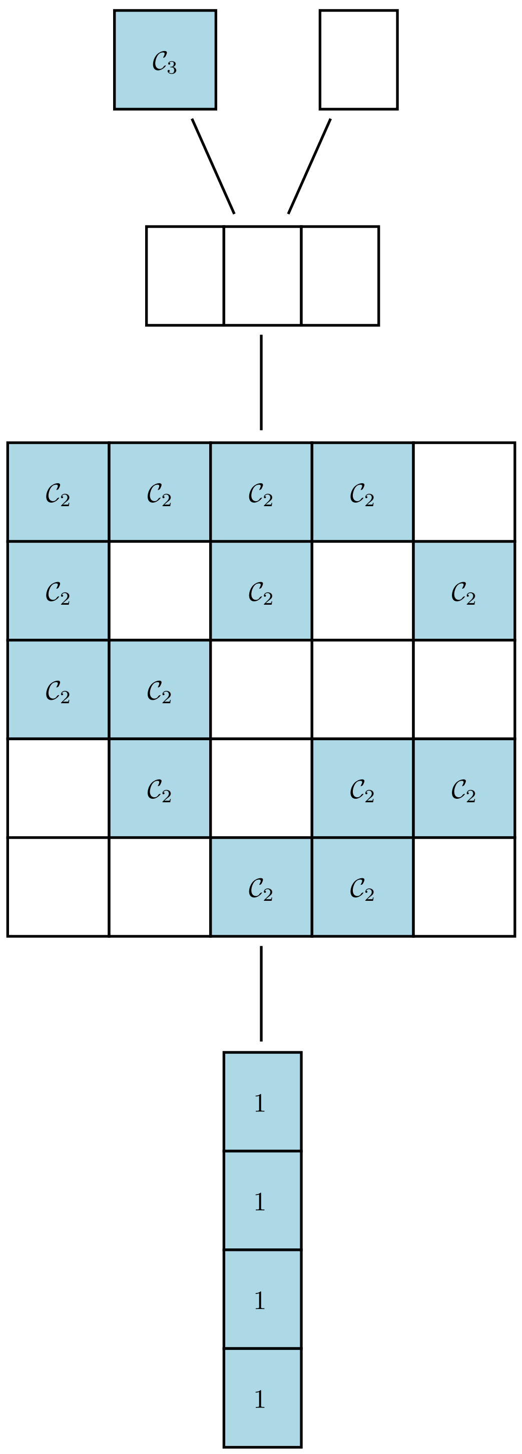


Figure 1.56: Eggbox diagram of the semigroup with 63 elements generated by the two transformations $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 3 \end{pmatrix}$ and $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 2 & 1 \end{pmatrix}$.

1.10 Regularity

In Section 1.2, we encountered the concept of a semigroup element's *inverse*. This definition is not the same as an inverse in a group (see Definition 1.4), but it is compatible with it, in the sense that a group element's inverse is also a semigroup inverse. In an inverse semigroup, every element has a unique inverse, but in general an element might not have an inverse, or might have more than one. We now present a slightly weaker condition than a semigroup inverse, the concept of a *regular* element.

Definition 1.57. Let S be a semigroup. An element $x \in S$ is called **regular** if there exists an element $y \in S$ such that

$$xyx = x.$$

Note that this definition is not enough to conclude that y is an inverse of x, without the additional requirement that $yxy = y$. However, an element with an inverse is certainly regular.

A semigroup may contain regular and non-regular elements. However, it turns out that a $\mathscr{D}$-class contains either no regular elements, or only regular elements [How95, Proposition 2.3.1]. This establishes the following definition.

Definition 1.58. A $\mathscr{D}$-class of a semigroup is called **regular** if some, and hence all, of its elements are regular. A semigroup is called **regular** if all its $\mathscr{D}$-classes are regular.

Since any element with an inverse is regular, we can see that all inverse semigroups are regular, as are all groups. We should mention one more interesting type of regular semigroup: a *rectangular band*.

Definition 1.59. A **rectangular band** is a semigroup S such that $xyx = x$ (i.e. y is an inverse of x) for all $x, y \in S$.

Rectangular bands will appear in Chapters 5 and 6, and the latter will use an important isomorphism theorem for rectangular bands, as follows.

Theorem 1.60 ([How95, Theorem 1.1.3])**.** *A rectangular band is isomorphic to the set $A \times B$ for two sets A and B, under the operation defined by*

$$(a_1, b_1)(a_2, b_2) = (a_1, b_2).$$

Since regular semigroups are not a central part of this thesis, we will not explain the wealth of theory attached to them, but where theory is needed we will cite appropriate literature. For a fuller explanation of regular semigroups, see [How95, §2.4].

1.11 Element types

In this thesis we will encounter several types of object that have a natural associative operation defined on them, and which are therefore well-suited to forming semigroups. We will define a few such objects in this section, drawing particular attention to Cayley's theorem and its analogues, which justify the heavy use of these objects as examples.

1.11.1 Transformations

Transformations are perhaps the most important type of element we will talk about in semigroup theory, as justified shortly by Theorem 1.63. They are defined very simply, as follows.

Definition 1.61. A **transformation** on a set X is a function $\tau : X \to X$.

In almost all cases in this thesis, a transformation will be on the set $\mathbf{n} = \{1, \dots, n\}$ for some $n \in \mathbb{N}$. This number n is called the **degree** of the transformation. A transformation τ on $\mathbf{n}$ can be written in *two-row notation*, that is with the numbers 1 to n written on one row, and their images under τ written directly beneath, all surrounded by parentheses. For example, the transformation of degree 5 sending even numbers to 1 and odd numbers to 3 would be written $\left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 3 & 1 & 3 \end{smallmatrix}\right)$.

Two transformations of a given degree can be composed to produce a new transformation of the same degree. For example, if $\tau = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 3 & 4 & 1 \end{smallmatrix}\right)$ and $\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 2 & 5 & 2 & 4 \end{smallmatrix}\right)$ then we can compose the two functions to give $\tau\sigma = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 5 & 2 & 2 \end{smallmatrix}\right)$. This operation is the basis of the following important semigroup.

Definition 1.62. The **full transformation monoid** on a set X is the set of all transformations on X, under the operation of composition, and it is denoted $\mathcal{T}_X$. If $X = \mathbf{n}$, then we call it the full transformation monoid of degree n, and denote it $\mathcal{T}_n$.

Any semigroup of finite-degree transformations is a subsemigroup of the full transformation monoid for some n. Note that this semigroup is a monoid because it contains the *identity map* $\mathrm{id}_n = \left(\begin{smallmatrix} 1 & 2 & \cdots & n \\ 1 & 2 & \cdots & n \end{smallmatrix}\right)$. We will refer to any subsemigroup of $\mathcal{T}_n$ as a **transformation semigroup**.

The true importance of transformations in semigroup theory is shown by the following theorem, which effectively states that any semigroup can be viewed as a transformation semigroup.

Theorem 1.63 (Cayley for semigroups [How95, Theorem 1.1.2])**.** *Every semigroup is isomorphic to a subsemigroup of a full transformation monoid.*

Proof. We give an outline of the proof. Let S be a semigroup, and consider the full transformation monoid $\mathcal{T}_{S^1}$ on the set of S^1, the semigroup S with an identity appended if it does not already contain one. Define a map $\phi : S \to \mathcal{T}_{S^1}$ by $(x)\phi : s \mapsto sx$ for $x \in S$ and $s \in S^1$; that is, $(x)\phi$ is the transformation in $\mathcal{T}_{S^1}$ which maps any point in S^1 to its right multiple by x. We can observe that this is a monomorphism, and hence that the image of ϕ is isomorphic to S. Hence S is isomorphic to a subsemigroup of $\mathcal{T}_{S^1}$. In particular, if S is finite, then it is isomorphic to a subsemigroup of $\mathcal{T}_n$, where $n = |S| + 1$. Note that S may also be isomorphic to a subsemigroup of $\mathcal{T}_n$ for some much smaller n. $\square$

One result of this theorem is that an algorithm for a semigroup of finite transformations can be applied to any finite semigroup. Furthermore, if we want to prove a result that only relies on the isomorphism class of a semigroup, we can prove it simply for semigroups of transformations.

The image of the map ϕ in the proof of the last theorem is called the **right regular representation** of S. In other words, to construct the right regular representation of S, we replace each element x by a transformation on S^1 which maps each element s to its right multiple sx. If we wish to express a generic semigroup as a transformation semigroup, this is a way in which we can always do so.

Next we consider a particular type of transformation of interest in group theory: permutations.

Definition 1.64. A **permutation** is a transformation that is a bijection, i.e. a transformation $\sigma : X \to X$ such that the following hold:

- $(i)\sigma = (j)\sigma$ if and only if $i = j$;
- every $j \in X$ has some $i \in X$ such that $(i)\sigma = j$.

Since a permutation σ is a bijection, we can define its inverse σ^{-1}, which acts an an inverse in the group theory sense (see Definition 1.4). This allows us to define a group comparable to the full transformation monoid, consisting of all the permutations in $\mathcal{T}_X$. Consider the following definition.

Definition 1.65. The **symmetric group** on a set X is the set of all permutations on X, under the operation of composition, and it is denoted $\mathcal{S}_X$. If $X = \mathbf{n}$, then we call it the symmetric group of degree n, and denote it $\mathcal{S}_n$.

The symmetric group plays the same role in group theory as the full transformation monoid does in semigroup theory, as shown in the following important theorem.

Theorem 1.66 (Cayley for groups [Rot65, Theorem 3.16])**.** *Every group is isomorphic to a subgroup of a symmetric group.*

This theorem is proven in much the same way as Cayley's theorem for semigroups (Theorem 1.63). It also entails a similar important fact, which is that any group can be viewed as a group of permutations, for the sake of determining information that is isomorphism-invariant.

1.11.2 Partial transformations

We will also deal with a generalisation of transformations known as *partial transformations.* A partial transformation can be seen as a transformation which simply fails to map certain points. The formal definition is as follows:

Definition 1.67. A **partial transformation** on a set X is a function $Y \to X$ for some subset Y of X.

Again, the set X in this thesis will typically be $\mathbf{n} = \{1, \dots, n\}$, and we will talk about a partial transformation of degree n in this case. In two-row notation we can show that a point is not mapped by a partial transformation by writing a '$-$' symbol under it, as in the partial transformation $\left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & - & - & 3 & 3 \end{smallmatrix}\right)$ which maps 1 to 4, 4 to 3 and 5 to 3, and does not map 2 or 3 at all. We compose two partial transformations σ and τ by the rule

$$(i)\sigma\tau = \begin{cases} \big((i)\sigma\big)\tau, & \text{if } (i)\sigma \text{ and } \big((i)\sigma\big)\tau \text{ are both defined;} \\ \text{undefined,} & \text{otherwise.} \end{cases}$$

Using this composition rule, we can define the **partial transformation monoid** $\mathcal{PT}_n$ of all partial transformations of degree n, in the same manner as the full transformation monoid $\mathcal{T}_n$.

We also have a partial counterpart to permutations, as follows.

Definition 1.68. A **partial permutation** is a partial transformation that is injective, i.e. a partial transformation such that $(i)\sigma = (j)\sigma$ if and only if $i = j$.

The product of two partial permutations is also a partial permutation, giving rise to the following important inverse semigroup, analogous to $\mathcal{T}_n$, $\mathcal{S}_n$ and $\mathcal{PT}_n$.

Definition 1.69. The **symmetric inverse monoid** on a set X is the set of all partial permutations on X, under the operation of composition, and it is denoted $\mathcal{I}_X$. If $X = \mathbf{n}$, then we call it the symmetric inverse monoid of degree n, and denote it $\mathcal{I}_n$.

This monoid plays the same important role for inverse semigroups that $\mathcal{T}_n$ and $\mathcal{S}_n$ play for semigroups and groups, as shown by the following analogue to the two Cayley theorems.

Theorem 1.70 (Wagner–Preston [How95, Theorem 5.1.7])**.** *Every inverse semigroup is isomorphic to a subsemigroup of a symmetric inverse monoid.*

We have now described several different types of partial transformation. We next define a few attributes of a partial transformation which will be used at various points.

Definition 1.71. Let τ be a partial transformation from $\mathcal{PT}_n$. We define the following attributes of τ.

- $\operatorname{dom}\tau$, the **domain** of τ, is the set of points which are mapped by τ;
- $\operatorname{rank}\tau$, the **rank** of τ, is size of $\operatorname{im}\tau$;
- $\ker\tau$, the **kernel** of τ, is the equivalence relation on $\operatorname{dom}\tau$ which contains all pairs (a, b) such that $(a)\tau = (b)\tau$;
- $\operatorname{im}\tau$, the **image** of τ, is the set of points which are mapped onto by τ.

Note that these attributes are also well-defined for transformations, permutations, and partial permutations, since they are all subsets of partial transformations. A partial permutation has rank equal to the size of its domain. A permutation always has rank equal to its degree, and image equal to its domain.

1.11.3 Order-preserving elements

Partial transformations (including transformations) can map points in their domain to points in their image in any order. We now define an interesting property a partial transformation may have that will be important when we come to consider *planar* bipartitions later (Definition 5.2). Recall that $\mathcal{PT}_n$ is the monoid of all partial transformations on the set $\{1, \dots, n\}$ for some $n \in \mathbb{N}$.

Definition 1.72. A partial transformation $\tau \in \mathcal{PT}_n$ is **order-preserving** if for all $i, j \in \operatorname{dom}\tau$, we have $i \leq j$ if and only if $(i)\tau \leq (j)\tau$.

Each of the monoids $\mathcal{PT}_n$, $\mathcal{T}_n$, $\mathcal{I}_n$ and $\mathcal{S}_n$ contains a submonoid consisting of the order-preserving elements: respectively, $\mathcal{PO}_n$, $\mathcal{O}_n$, $\mathcal{POI}_n$ and the group consisting of just the identity map id_n. The containment of these monoids is shown in Figure 1.73.

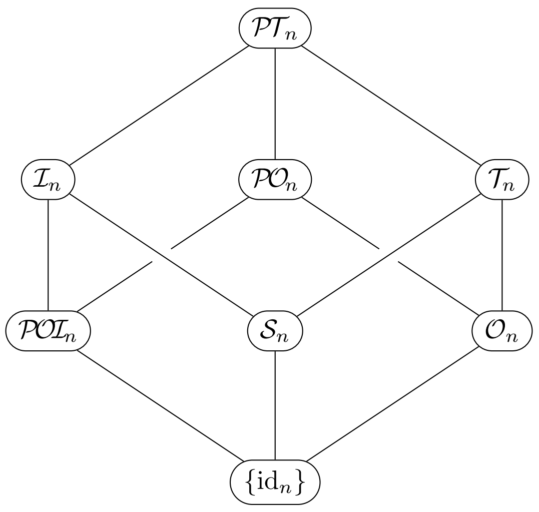


Figure 1.73: Hasse diagram showing containment of some important monoids of partial transformations, along with their order-preserving submonoids.

Note that the definition of *order-preserving* only makes sense for partial transformations that act on a set with a natural total ordering, such as the set $\{1, \dots, n\}$. It is not well-defined in $\mathcal{PT}_X$ for an arbitrary set X.

These order-preserving monoids will be considered in Section 6.1.3, where we will classify the congruences on their principal factors.

1.11.4 Bipartitions

Another important type of element discussed in this thesis is a *bipartition* (sometimes referred to just as a *partition*). Bipartitions are formally defined as a class of equivalence relations that form semigroups under an interesting composition operation; but we will often understand them in a graphical way, as in Figure 1.76. Bipartition semigroups are thus included in the class of *diagram semigroups* [EENFM15]. The study of bipartition semigroups is born out of the study of diagram algebras, for example Temperley–Lieb algebras and the bipartition algebra [Mar94]. However, they are of independent interest in semigroup theory, since the bipartition monoid (defined below) contains copies of important algebras such as $\mathcal{S}_n$, $\mathcal{T}_n$ and $\mathcal{I}_n$. It also has other interesting features such as being an example of a *regular $\star$-semigroup*, also defined below. For more information about bipartition semigroups, see [DEG17, §1].

We begin with the formal definition.

Definition 1.74. A **bipartition** is an equivalence relation on the set $\mathbf{n} \cup \mathbf{n}'$, where $\mathbf{n} = \{1, \dots, n\}$ and $\mathbf{n}' = \{1', \dots, n'\}$ for some $n \in \mathbb{N}$.

The equivalence classes of a bipartition are called **blocks**. A block is called an **upper block** if it only contains points from $\mathbf{n}$, a **lower block** if it only contains points from $\mathbf{n}'$, or a **transversal** if it contains points from both $\mathbf{n}$ and $\mathbf{n}'$.

The number n is called the **degree** of the bipartition. Two bipartitions α and β of the same degree can be composed in the following way to make another bipartition, $\alpha\beta$: let $\mathbf{n}'' = \{1'', \dots, n''\}$, let $\alpha^\vee$ be obtained from α by changing every point $i' \in \mathbf{n}'$ to i'', and let $\beta^\wedge$ be obtained from β be changing every point $i \in \mathbf{n}$ to i''. Now let Π be the equivalence on $\mathbf{n} \cup \mathbf{n}' \cup \mathbf{n}''$

given by $(\alpha^{\vee} \cup \beta^{\wedge})^{e}$. We define $\alpha\beta$ as the bipartition $\Pi \cap \big((\mathbf{n} \cup \mathbf{n}') \times (\mathbf{n} \cup \mathbf{n}')\big)$. This composition is more easily understood visually, as will be seen in Example 1.75. The operation can be seen to be associative, and so we can use it to form semigroups of bipartitions.

A bipartition can be displayed visually by plotting the points 1 to n in order in a horizontal line, with the corresponding points $1'$ to n' underneath, and drawing edges between points to make a spanning skeleton of the equivalence relation. The product of two bipartitions can then be found by concatenating the two diagrams top-to-bottom and deleting points in the middle line. Consider the following example.

Example 1.75. Let α be the bipartition of degree 5 with blocks $\{1, 1', 2'\}$, $\{2\}$, $\{3, 4, 3', 4'\}$, $\{5\}$ and $\{5'\}$. Let β be the bipartition of degree 5 with blocks $\{1, 2, 5, 1', 2'\}$, $\{3, 4', 5'\}$, $\{4\}$ and $\{3'\}$. The diagrams of these two figures, along with their product $\alpha\beta$, are shown in Figure 1.76. Note that two different diagrams are shown for $\alpha\beta$.

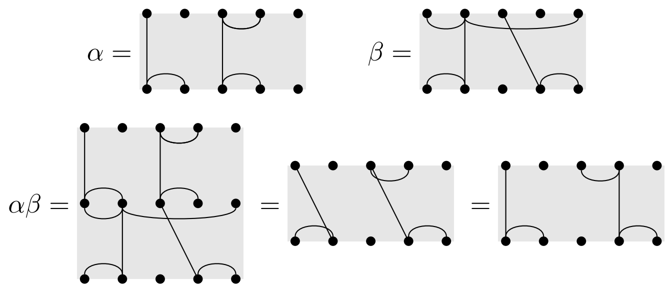


Figure 1.76: Diagrams of the bipartitions in Example 1.75.

Since a given equivalence may have several spanning skeletons, a bipartition may be represented by several different diagrams; for example, note the two different representations of $\alpha\beta$ in Figure 1.76. In general, it does not matter which diagram is used, only that it represents the appropriate bipartition – we will usually choose the diagram that illustrates the bipartition most clearly. In particular, for each block that contains points from both $\mathbf{n}$ and $\mathbf{n}'$, we will usually draw only one line crossing the diagram from top to bottom.

Next we define some attributes of bipartitions, which will be important when we come to consider certain bipartition semigroups later.

Definition 1.77. Let α be a bipartition.

- The **rank** of α, denoted $\operatorname{rank} \alpha$, is the number of transversals in α;
- The **domain** of α, denoted $\operatorname{dom} \alpha$, is the set of points $i \in \mathbf{n}$ such that i lies in a transversal of α;
- The **codomain** of α, denoted $\operatorname{codom} \alpha$, is the set of points $i \in \mathbf{n}$ such that i' lies in a transversal of α;
- The **kernel** of α, denoted $\ker \alpha$, is the equivalence relation on $\mathbf{n}$ such that two points $i, j \in \mathbf{n}$ lie in the same block of $\ker \alpha$ if and only if they lie in the same block of α (equivalently, $\ker \alpha = \alpha \cap (\mathbf{n} \times \mathbf{n})$);

- The **cokernel** of α, denoted $\operatorname{coker}\alpha$, is the equivalence relation on $\mathbf{n}$ such that two points $i, j \in \mathbf{n}$ lie in the same block of $\operatorname{coker}\alpha$ if and only if the corresponding points $i', j' \in \mathbf{n}'$ lie in the same block of α.

We illustrate these attributes by continuing our example.

Example 1.78. Let α and β be the bipartitions described in Example 1.75. The attributes of α are

$$\operatorname{dom}\alpha = \{1,3,4\}, \qquad \ker\alpha = \big\{\{1\},\{2\},\{3,4\},\{5\}\big\},$$

$$\operatorname{codom}\alpha = \{1,2,3,4\}, \qquad \operatorname{coker}\alpha = \big\{\{1,2\},\{3,4\},\{5\}\big\},$$

and the attributes of β are

$$\operatorname{dom}\beta = \{1,2,3,5\}, \qquad \ker\beta = \big\{\{1,2,5\},\{3\},\{4\}\big\},$$

$$\operatorname{codom}\beta = \{1,2,4,5\}, \qquad \operatorname{coker}\beta = \big\{\{1,2\},\{3\},\{4,5\}\big\},$$

where a kernel or cokernel is identified with the set of its blocks. We also have $\operatorname{rank}\alpha = \operatorname{rank}\beta = 2$.

Since drawing a diagram for a bipartition consumes a lot of space, and since writing out the blocks is unwieldy and difficult to read, we have another way of representing a bipartition: a modified two-row notation. In this notation, the blocks of the bipartition's kernel are written across the top row, separated by vertical lines, and the blocks of the cokernel are written across the bottom row, omitting prime symbols. Transversals are written first, with the appropriate kernel block written above its corresponding cokernel block. Non-transversal blocks are written afterwards, with upper and lower blocks separated by horizontal lines. A generic bipartition could thus be represented by

$$\alpha = \left[\begin{array}{c|c|c|c|c|c} A_1 & \dots & A_q & C_1 & \dots & C_r \\ \cline{4-6} B_1 & \dots & B_q & D_1 & \dots & D_s \end{array}\right],$$

where α has q transversals of the form $A_i \cup B_i'$ with $A_i \subseteq \mathbf{n}$ and $B_i' \subseteq \mathbf{n}'$, r upper blocks labelled C_i, and s lower blocks labelled D_i'.

Example 1.79. The bipartitions from Example 1.75 can be written in the form

$$\alpha = \left[\begin{array}{c|c|c|c} 1 & 3,4 & 2 & 5 \\ \cline{3-4} 1,2 & 3,4 & \multicolumn{2}{c}{5} \end{array}\right], \qquad \beta = \left[\begin{array}{c|c|c} 1,2,5 & 3 & 4 \\ \cline{3-3} 1,2 & 4,5 & 3 \end{array}\right].$$

We should also mention the $\star$ operation. To each bipartition α is assigned another bipartition $\alpha^\star$, which is found by swapping each point $i \in \mathbf{n}$ with its opposite point i'. Hence if $\alpha = \left[\begin{array}{c|c|c|c|c|c} A_1 & \dots & A_q & C_1 & \dots & C_r \\ \cline{4-6} B_1 & \dots & B_q & D_1 & \dots & D_s \end{array}\right]$, then we have $\alpha^\star = \left[\begin{array}{c|c|c|c|c|c} B_1 & \dots & B_q & D_1 & \dots & D_s \\ \cline{4-6} A_1 & \dots & A_q & C_1 & \dots & C_r \end{array}\right]$. This operation has the property that $\alpha\alpha^\star\alpha = \alpha$, and that $(\alpha^\star)^\star = \alpha$. Hence $\alpha^\star$ is a semigroup inverse for α.

We now consider the semigroup of all bipartitions of a given degree.

Definition 1.80. The **bipartition monoid** $\mathcal{P}_n$ is the semigroup of all bipartitions of degree n under composition, where $n \in \mathbb{N}$.

The bipartition monoid has many interesting features. First of all, a number of other important semigroups embed into $\mathcal{P}_n$ as subsemigroups. For example, consider the following way of embedding the full transformation monoid $\mathcal{T}_n$, the symmetric inverse monoid $\mathcal{I}_n$, and the symmetric group $\mathcal{S}_n$.

Example 1.81. Let $f : \mathcal{PT}_n \to \mathcal{P}_n$ be the map that sends a partial transformation τ to the bipartition $\big\{\big(i, (i\tau)'\big) : i \in \operatorname{dom}\tau\big\}^e$. For example, the partial transformation $\left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & - & 4 & 3 & - \end{smallmatrix}\right)$ would be mapped to the bipartition $\left[\begin{array}{c|c|c|c|c} 1,4 & 3 & \multicolumn{2}{c|}{2} & 5 \\ \cline{3-5} 3 & 4 & 1 & 2 & 5 \end{array}\right]$. It would be easy to mistake f for a homomorphism; however, consider the following simple counter-example:

$$a = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 & 2 \\ - & - \end{pmatrix}, \quad ab = \begin{pmatrix} 1 & 2 \\ - & - \end{pmatrix},$$

$$(a)f = \left[\begin{array}{c|c} 1,2 & \\ \cline{2-2} 1 & 2 \end{array}\right], \quad (b)f = \left[\begin{array}{c|c} 1 & 2 \\ \hline 1 & 2 \end{array}\right],$$

$$(a)f(b)f = \left[\begin{array}{cc} \multicolumn{2}{c}{1,2} \\ \hline 1 & \multicolumn{1}{|c}{2} \end{array}\right] \neq \left[\begin{array}{c|c} 1 & 2 \\ \hline 1 & 2 \end{array}\right] = (ab)f,$$

showing that f does not respect composition. Still, f is useful as an embedding for several submonoids of $\mathcal{PT}_n$. The restricted maps $f|_{\mathcal{T}_n}$ and $f|_{\mathcal{I}_n}$ are monomorphisms [Eas11, §3.1–3.2] and hence, since $\mathcal{S}_n = \mathcal{T}_n \cap \mathcal{I}_n$, so is $f|_{\mathcal{S}_n}$. Thus, we can see that $\mathcal{P}_n$ contains copies of $\mathcal{T}_n$, $\mathcal{I}_n$ and $\mathcal{S}_n$.

Note that, since each bipartition $\alpha \in \mathcal{P}_n$ has an inverse $\alpha^\star \in \mathcal{P}_n$, we have that $\mathcal{P}_n$ is a regular semigroup. Furthermore, consider the following definition.

Definition 1.82 ([NS78, Definition 1.1])**.** A **regular $\star$-semigroup** is a semigroup S together with a unary operation $^\star : S \to S$ such that the following hold:

(i) $(x^\star)^\star = x$;

(ii) $(xy)^\star = y^\star x^\star$;

(iii) $x = xx^\star x$;

for all $x, y \in S$.

We can see that the * operation described above fulfils all three of these criteria, and therefore that $(\mathcal{P}_n, ^\star)$ is a regular $\star$-semigroup. For more information on regular $\star$-semigroups, see [NS78].

As with any semigroup, it will be helpful to describe the Green's relations of the bipartition monoid. The following proposition describes the Green's relations, the containment of $\mathscr{J}$-classes and the ideals of $\mathcal{P}_n$ very simply in terms of domain, codomain, kernel, cokernel, and rank.

Proposition 1.83. *Let α and β be bipartitions in $\mathcal{P}_n$. The following hold:*

(i) *$\alpha \mathrel{\mathscr{R}} \beta$ if and only if $\operatorname{dom}\alpha = \operatorname{dom}\beta$ and $\ker\alpha = \ker\beta$;*

(ii) *$\alpha \mathrel{\mathscr{L}} \beta$ if and only if $\operatorname{codom}\alpha = \operatorname{codom}\beta$ and $\operatorname{coker}\alpha = \operatorname{coker}\beta$;*

(iii) *$\alpha \mathrel{\mathscr{J}} \beta$ if and only if $\operatorname{rank}\alpha = \operatorname{rank}\beta$;*

(iv) *$J_\alpha \leq J_\beta$ if and only if $\operatorname{rank}\alpha \leq \operatorname{rank}\beta$.*

(v) *the ideals of $\mathcal{P}_n$ are precisely the sets $I_r = \{\alpha \in \mathcal{P}_n : \operatorname{rank}\alpha \leq r\}$ for $r \in \{0, \ldots, n\}$.*

Proof. Parts (i) to (iv) are from [FL11], and (v) follows immediately from (iv). □

The bipartition monoid $\mathcal{P}_n$ grows very quickly as n increases: its size is the Bell number B_{2n} [OEIS, A000110]. It therefore grows much faster than the symmetric group $\mathcal{S}_n$ with $n!$ elements, the symmetric inverse monoid $\mathcal{I}_n$ with $\sum_{k=0}^{n} \binom{n}{k} \frac{n!}{(n-k)!}$ elements, and even the full transformation monoid $\mathcal{T}_n$ with n^n elements. The degree of this difference is illustrated in Table 1.84.

n	$\lvert\mathcal{S}_n\rvert$	$\lvert\mathcal{I}_n\rvert$	$\lvert\mathcal{T}_n\rvert$	$\lvert\mathcal{P}_n\rvert$
1	1	2	1	2
2	2	7	4	15
3	6	34	27	203
4	24	209	256	4 140
5	120	1 546	3 125	115 975
6	720	13 327	46 656	4 213 597
7	5 040	130 922	823 543	190 899 322
8	40 320	1 441 729	16 777 216	10 480 142 147
9	362 880	17 572 114	387 420 489	682 076 806 159
10	3 628 800	234 662 231	10 000 000 000	51 724 158 235 372

Table 1.84: Sizes of $\mathcal{S}_n$, $\mathcal{I}_n$, $\mathcal{T}_n$ and $\mathcal{P}_n$ for small values of n.

1.12 Computation & decidability

Since this thesis will deal with many computational issues, it may be helpful to make precise some computational terms. We start with a definition of "algorithm", a term which is generally well understood, but whose precise definition is debatable. In this thesis, we opt for a definition in line with the Church–Turing thesis, which has been favoured by a variety of authors since it was established [Min67, Gur00].

The Church–Turing thesis evolved from work by Gödel, Church and Turing in the 1930s, in which they established three different models of computation: *general recursive functions* [Göd31], *λ-calculus* [Chu36], and *Turing machines* [Tur37]. These three models were soon shown to be equivalent, with any method computable on one being computable on both the others. This led to the Church–Turing thesis: the opinion that the informal notion of an algorithm is accurately characterised by each of these three models, and therefore that they should be used as a definition of "algorithm". This gives rise to our chosen definition.

Definition 1.85. An **algorithm** is a computational method which can be simulated by a Turing machine.

A Turing machine is a conceptual machine based on a finite state automaton which interacts with an infinite tape. There are several different formulations of a Turing machine, all of which are equivalent in terms of the set of computational methods that they can run. Though this definition is not central to this thesis, we present one such formulation here for completeness.

Definition 1.86. A **Turing machine** is a tuple $(Q, \Sigma, q_0, \delta, H)$ where

- Q is a set, called the set of *states*;
- Σ is a set, called the *alphabet*;

- $q_0 \in Q$ is a state known as the *initial state*;
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$ is a function known as the *transition function*, which uses the current state and a character from the tape to process a step in the computation;
- $H \subseteq Q$ is a subset of states known as the *halting states*.

This machine is paired with a conceptual infinite *tape* – a sequence of characters from Σ which continues infinitely in both directions. This tape is provided as an input to the algorithm. The machine starts in state q_0 with its read/write head pointed to a given position on the tape. On each step of computation, it starts in a state q, reads a symbol σ from the read/write head's position on the tape, and uses $(q, \sigma)\delta$ to produce a triple (q', σ', d) from $Q \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$: it then changes state from q to q', replaces the character σ on the tape with σ', and moves the read/write head one character to the left or right according to d. This process is repeated until the machine enters a state in H, at which time it halts, having completed its operation. The program's output is the resulting tape.

This definition encompasses every computation which can be run on today's electronic computers, and is therefore certainly applicable to any practical implementation of the algorithms described in this thesis.

Now that we have a definition of an algorithm, we can define decidability. Decidability is not a core topic of this thesis, but it will be worth going into at least some detail, particularly around ideas related to semigroup presentations. A deeper discussion of decidability can be found in, for example, [End01].

Definition 1.87. A class of problems is **decidable** if there exists a single algorithm which is guaranteed to return a correct answer to any instance of one of those problems in a finite amount of time.

Note that although an algorithm may be guaranteed to complete in finite time, the length of time might be unbounded; that is, an actual run of the algorithm, though gururanteed to complete, might take an arbitrarily long time.

Decidability is always something that should be considered when designing an algorithm that may act on infinite objects. In this thesis, particularly in Chapter 2, we encounter semigroup presentations, which have an interesting decidability feature.

Definition 1.88. Let $\langle X \mid R \rangle$ be a semigroup presentation. The **word problem** for $\langle X \mid R \rangle$ is the following question: given two words $u, v \in X^+$, do u and v represent the same semigroup element?

For many individual semigroup presentations, the word problem is decidable. For example, consider the following presentation.

Example 1.89. Let S be defined by the presentation

$$\langle a, b \mid ab = ba \rangle .$$

The word problem for this presentation is decidable: two words represent the same element of S if and only if they contain the same number of occurrences of a and the same number of occurrences of b.

In Example 1.89, there is an algorithm to answer the word problem; hence, we say that $\langle\, a, b \,|\, ab = ba \,\rangle$ has decidable word problem. However, we have not shown the word problem in general to be decidable, since our algorithm does not apply to every semigroup presentation, only to the one considered in the example. It turns out that there is no single algorithm which can be applied to every presentation to solve its word problem. In fact, we can make a stronger statement: there are some presentations for which there is not even a specific algorithm to solve the word problem. Consider the following example from Makanin, which has only three generators.

Example 1.90 (Makanin, 1966)**.** The presentation

$$\langle a, b, c \mid c^2b^2 = b^2c^2,\ bc^3b^2 = cb^3c^2,\ ac^2b^2 = b^2a,$$
$$abc^3b^2 = cb^2a,\ b^2c^2b^4c^2 = b^2c^2b^4c^2a\rangle$$

has undecidable word problem [Mak66].

Furthermore, consider the following example from Cijtin, perhaps the simplest undecidable presentation, with 5 generators but only 33 occurrences of those generators in its relations.

Example 1.91 (Cijtin, 1957)**.** The presentation

$$\langle a, b, c, d, e \mid ac = ca,\ ad = da,\ bc = cb,\ bd = db,$$
$$ce = eca,\ de = edb,\ c^2e = c^2ae\rangle$$

has undecidable word problem [Cij57, C$^+$86].

These two examples show that even relatively simple presentations can give rise to semigroups with undecidable word problem. Hence, no algorithm we give for solving the word problem can be guaranteed to finish in finite time. Furthermore, since words can be arbitrarily long, and presentations can have an arbitrarily large set of relations, solving the word problem can take an unbounded finite length of time even if it is decidable. This means that it may be impossible to tell whether an algorithm for the word problem will complete or not, since it is not clear in advance whether a given semigroup has decidable word problem. At a given stage while such an algorithm is running, it may be that an answer is about to be returned, but it may instead be that it will run forever, and it may be impossible to tell the difference between those two possibilities.

Aside from the word problem, there are a great many other properties of finite presentations which are undecidable in general. For example, there is no algorithm which can take an arbitrary finite presentation and decide whether the semigroup it describes:

- is finite;
- has an identity;
- has a zero;
- has idempotents;
- is a group;

- has a nontrivial subgroup;
- is an inverse semigroup;
- is regular; or
- is simple.

References for these facts, and a survey of many other decidability problems for finitely presented semigroups, can be found in [CM09].

1.13 Union–find

This thesis deals with congruences, and a congruence is a particular type of equivalence. It will therefore be useful for us to discuss methods of computing with equivalences, in order to help us describe other algorithms later, particularly the pair orbit enumeration algorithm in Section 2.6.1. We begin by recalling a few definitions, and stating a problem we wish to solve.

Let X be a set. Recall that an *equivalence* on X is a relation (a subset of $X \times X$) which is reflexive, symmetric, and transitive – that is, a partition of X into disjoint subsets. Also recall from Definition 1.32 the relation $\mathbf{R}^e$, the least equivalence containing a given relation $\mathbf{R}$. It may be that, given a set of pairs $\mathbf{R}$, we wish to compute the equivalence $\mathbf{R}^e$. This is where the union–find method can be useful.

A **union–find** table, also known as a disjoint-set data structure, is a data structure that stores and modifies an equivalence relation by viewing it as a partition and using trees to keep track of which elements lie in which class. This approach was first described in 1964 in [GF64], and its time complexity has since been improved in various ways. A few of these ways will be described after the main description of the algorithm, but see [GI91] for a detailed survey of different improvements and their possible advantages and drawbacks.

Assume we wish to compute $\mathbf{R}^e$ for a relation $\mathbf{R}$ on a set X. A union–find table, for us, is a pair (Λ, τ) consisting of a set Λ of elements from X and a function $\tau : \Lambda \to \Lambda$. The set Λ will contain all elements not in singletons, and the function τ will be used to keep track of which elements of Λ are in which equivalence class, in a way described below. Initially Λ is empty, and τ is the empty function $\varnothing \to \varnothing$. This initial state represents the diagonal relation Δ_X. We proceed by iterating through the pairs in $\mathbf{R}$, and updating Λ and τ using the following three operations:

- $\textsc{AddElement}$ takes an element $x \in X$ and starts tracking it using the union–find table;
- $\textsc{Union}$ takes two elements $x, y \in X$ and alters the table to indicate that they lie in the same class in $\mathbf{R}^e$;
- $\textsc{Find}$ takes an element $x \in X$ and returns a canonical representative x' of the equivalence class in which x lies. For two elements $x, y \in X$, we have $\textsc{Find}(x) = \textsc{Find}(y)$ if and only if $(x, y) \in \mathbf{R}^e$.

At the beginning of the algorithm, every $\mathbf{R}^e$-class is assumed to be a singleton. The set Λ only needs to contain the elements that are in non-singletons, so it starts empty; the function τ is defined over Λ, so it is also empty. As the algorithm progresses, at various times we will find that two distinct elements (say x and y) are $\mathbf{R}^e$-related, and we will wish to record this. If either x or y is not already in Λ, we call AddElement on it to start tracking it in the union–find table; AddElement simply adds an element – for example, x – to the set Λ, and redefines τ such that $(x)\tau = x$. See Algorithm 1.92 for pseudo-code. Then we call Union(x, y) to combine the two classes as described below.

Algorithm 1.92 The AddElement algorithm (union–find)

```
Require: $x \notin \Lambda$
1: procedure AddElement($x$)
2:     $\Lambda \leftarrow \Lambda \cup \{x\}$
3:     $(x)\tau := x$
```

A simple way of tracking classes would be for τ to be a function from X to $\mathbb{N}$, where $(x)\tau$ would be the index of the equivalence class in which x lies: distinct elements x and y would be in the same equivalence class if and only if $x, y \in \Lambda$ and $(x)\tau = (y)\tau$. By this method, Union(x, y) would need go through the whole of τ, finding every z such that $(z)\tau = (y)\tau$, and updating it so that $(z)\tau = (x)\tau$, thus making the two classes equal. However, this operation has high time complexity – in the worst case, $O(|X|)$ – and would cause any implementation of this algorithm to take a long time to complete. Instead, the union–find algorithm treats τ as a pointer to a parent element in a *forest* structure, as follows.

Formally, we say that a forest (that is, a set of rooted trees) describes an equivalence relation on a set X if each element of X appears as a node in precisely one tree, and the set of nodes in each tree in the forest is equal to one equivalence class. The arrangement of nodes in each tree is not important, but it should be noted that each tree will have a single root, which will be one element in the equivalence class the tree defines. We use τ to describe such a forest as follows.

Rather than treating τ as a simple function such that elements are $\mathbf{R}^e$-related if and only if they have the same τ output, we instead have τ map an element in Λ to its parent node in the forest of $\mathbf{R}^e$. If x is an element in Λ, then $(x)\tau$ is the parent of x in the tree that contains all the elements in its class. Each class contains a single element r such that $(r)\tau = r$; this is the root of the tree. Hence the Find function takes an element x, and traverses the tree all the way back to the root by calling τ on it again and again until we reach the root. Pseudo-code is given in Algorithm 1.93.

Algorithm 1.93 The Find algorithm (union–find)

```
1: procedure Find($x$)
2:     repeat
3:         $x \leftarrow (x)\tau$              ▷ Set $x$ to the parent of the old $x$
4:     until $x = (x)\tau$                ▷ Check whether the new $x$ is the root
5:     return $x$
```

Now we may view the operation of finding an element's class as traversing a tree from a node up to its root, and we can view the entire connected tree as the class itself. In order to combine two classes, therefore, we have the function Union, which simply finds the roots of the two trees and changes one to point to the other. Its pseudo-code is shown in Algorithm 1.94. Note that a total ordering $<$ of elements is used; this ordering can be arbitrary, but since it is only ever used for elements in non-singletons, we choose to use the order in which elements were added to Λ. In lines 2 and 3, we find the roots x' and y' of the trees of x and y, by using Find. Whichever of these is higher ($x' < y'$ or $y' < x'$) is set so that its τ output is equal to the one which is lower: $(y')\tau \leftarrow x'$ or $(x')\tau \leftarrow y'$. Hence, a future call to Find(x) will return the same result as a call to Find(y) – the result will be whichever is the lower of x' and y'. Example 1.95 shows this algorithm in action.

Algorithm 1.94 The Union algorithm (union–find)

```
1: procedure Union($x, y$)
2:     $x' :=$ Find($x$)
3:     $y' :=$ Find($y$)
4:     if $x' < y'$ then
5:         $(y')\tau \leftarrow x'$
6:     else if $y' < x'$ then
7:         $(x')\tau \leftarrow y'$
```

Note that the algorithms described only ever make z an output of τ if $(z)\tau = z$. This ensures that when Find traverses a tree, it always moves towards the root, and never gets caught in a cycle. Neither AddElement nor Union contain any loops, so if Λ is finite and τ was created using the methods described, all three algorithms will always halt in finite time.

These three algorithms allow us to use a simple data structure (Λ, τ) to describe any equivalence relation on a semigroup. Whenever a pair (x, y) is found in ρ, we call AddElement(x) and AddElement(y) if necessary, and then call Union(x, y). This combines the congruence classes of x and y, and forces Find(x) = Find(y).

Note that this union–find method has automatically removed the problem of transitivity, as well as those of reflexivity and symmetry: if we relate the element x to y, and then y to z, we have combined all three elements into a single class, and so we will see that Find(x) = Find(z), so we have added the pair (x, z) with no additional effort; similarly every element x is related to itself from the very beginning; and relating x to y is precisely the same as relating y to x. In other words, if we perform Union on all the pairs of $\mathbf{R}$ one by one, we produce Λ and τ which describe the equivalence $\mathbf{R}^e$.

Other descriptions of union–find do not always include AddElement. The union–find algorithm is generally used to calculate equivalences on finite sets, but it is possible to use it with infinite sets as well. The method described here allows for X to be an infinite set, and stores information only about elements that are not in singletons, by calling AddElement on each element only when it is found to be in the same class as another element. If there are infinitely many elements in non-singleton classes, or if there are infinitely many pairs in $\mathbf{R}$, then of course $\mathbf{R}^e$ cannot be computed with this method.

Example 1.95. Let $X = \{a, b, c, d, e, f, g, h\}$ and let $\mathbf{R}$ be the set of pairs

$$\{(f, b), (d, c), (e, b), (b, d)\}.$$

We can calculate the classes of $\mathbf{R}^e$ by applying Union to each pair in $\mathbf{R}$ in turn, calling AddElement on any appropriate elements to add them to Λ first.

First we call AddElement(f) and AddElement(b), then Union(f, b). After these operations we have $\Lambda = \{f, b\}$ and τ has the results $(f)\tau = f$ and $(b)\tau = f$, representing just one non-singleton class containing both elements. Next we call AddElement on d and c, and then Union(d, c), after which we have $\Lambda = \{f, b, d, c\}$ and new results $(d)\tau = d$ and $(c)\tau = d$. This state is shown in forest form in the first diagram of Figure 1.96.

Next we AddElement(e) and call Union(e, b). Union follows b to its parent f in the tree, and sets e to point to it; hence $\Lambda = \{f, b, d, c, e\}$ and $(e)\tau = f$. This is represented in forrest form in the second diagram of Figure 1.96.

Finally we process the last pair by calling Union(b, d). This finds the root of b, which is f, and the root of d, which is d itself, and unites the two roots. f was added to Λ before d was, so $(d)\tau$ is set equal to f. At this final stage we have $\Lambda = \{f, b, d, c, e\}$ as before, and τ maps the elements f, b, d, c, e to f, f, f, d, f respectively, representing the forest structure shown in the third diagram of Figure 1.96.

$$\begin{tikzcd} f & d \\ b \arrow[u] & c \arrow[u] \end{tikzcd} \qquad \begin{tikzcd} & f & & d \\ b \arrow[ur] & & e \arrow[ul] & c \arrow[u] \end{tikzcd} \qquad \begin{tikzcd} & f & \\ b \arrow[ur] & e \arrow[u] & d \arrow[ul] \\ & & c \arrow[u] \end{tikzcd}$$

Figure 1.96: Diagrams of union–find table in Example 1.95.

This represents a single tree, and therefore a single equivalence class consisting of all the elements $\{b, c, d, e, f\}$. However, note that a, g and h have not been added to Λ, so they are in singleton classes of $\mathbf{R}^e$.

The simple description we have given so far is sufficient to implement a working version of the algorithm, but has complexity that can be easily reduced. The height of a tree created by repeated applications of Union can be as great as the size of the set X, which means that the worst-case time complexity of both Find and Union is $O(|X|)$. But we may consider the following improvements to both Find and Union, which limit the height of trees and thus lower complexity.

The Find operation descends all the way from a node to the root of its tree, but does not do anything with the final value that is found. Hence, if Find is later called on the same element, all the work is likely to be repeated. One possible improvement is to change the element's τ-entry to be equal to the result of Find, before returning. This way, a future call to Find will reach the root of the tree in a single step. Furthermore, it is possible to change every node in the tree along the way, essentially flattening the entire path each time Find is called. This improvement is known as *path compression* [HU73]. Alternatives have been proposed which do less up-front work, for example *path splitting* which points each node to its grandparent, or even *path halving* which points alternate nodes to their grandparents [vLW77]. These all improve complexity in a way which we will describe shortly.

The Union operation combines two trees by making one root the parent of the other. In the method described above, we choose the root that was added to Λ earlier to be the new parent, but we might choose the parent differently. The *union by size* method keeps track of the size of each tree, and makes the smaller tree point to the larger [GI91]; the *union by rank* method instead keeps track of the depth of each tree (the length of the longest path from the root) and makes the shallower tree point to the deeper [TvL84]. Either of these methods curbs the height of trees in the table, preventing any tree from growing to a height greater than $\lceil \log_2 |X| \rceil$ [GI91, Lemma 1.1.2].

These improvements are enough to give us the following statement about complexity.

Theorem 1.97 ([GI91, Theorem 1.1.1])**.** *Choose any* FIND *method from path compression, path splitting or path halving. Choose either union by size or union by rank as a* UNION *method. A sequence of $n-1$ calls to* UNION *and m calls to* FIND *completes in $O(n + m\alpha(m+n, n))$ time, where α is a functional inverse of Ackermann's function.*

Ackermann's function is a function which grows extremely quickly, and the functional inverse used in Theorem 1.97 (defined explicitly in [TvL84]) therefore grows extremely slowly. In fact, we have $\alpha(m, n) \leq 3$ for any $n < 2^{16}$, so in practice it can be treated as constant and the complexity stated in Theorem 1.97 is close to $O(n+m)$, meaning that over several calls, UNION and FIND have close to constant time complexity.

Part I

Computational techniques

Chapter 2

Parallel method for generating pairs

A congruence is a binary relation, and therefore is formally described as a set of pairs. In a computational setting, it is rarely practical to keep track of every pair in a congruence; a congruence on a semigroup of size n contains n^2 pairs in the worst case, and on an infinite semigroup contains an infinite number of pairs. A congruence can be described in more concise ways: for example, taking advantage of it being an equivalence relation and recording only its equivalence classes; or in the case of a Rees congruence, storing a generating set for the ideal which defines it. A variety of different ways to describe a congruence are explained in Chapter 3, along with ways to convert from one to another. However, a congruence is still just a set of pairs, and by reducing the number of pairs we store, we can often describe a congruence very concisely using them.

Let S be a semigroup and let R be a subset of $S \times S$. Recall from Definition 1.30 that the *congruence generated by* R is the least congruence (with respect to containment) that contains R as a subset. This definition is based on a simple intuitive idea: a congruence ρ is generated by a set of pairs R if it consists of only the pairs in R along with the pairs required by the axioms of a congruence (reflexivity, symmetry, transitivity and compatibility). Thus a congruence can be described completely by storing only a few pairs. Indeed, many congruences are *principal*, requiring only one pair to generate them: see, for example, the congruences studied in Chapter 5, most of which are principal.

Another justification for the use of generating pairs is that it is a completely generic representation. Some special types of semigroup have their own abstract representations of congruences – for inverse semigroups, one can study kernel–trace pairs [How95, §5.3]; for groups, normal subgroups [War90, Theorem 11.5]; for completely simple or completely 0-simple semigroups, linked triples [How95, §3.5] – but generating pairs can represent a congruence on any semigroup whatsoever. Furthermore, one might be interested in what pairs are implied by a given pair or set of pairs in a congruence, and this representation can answer such questions. Left congruences and right congruences can also be described using generating pairs, and some algorithms designed for two-sided congruences can be used with minor modifications to compute information about left and right congruences.

Algorithms for computing a congruence defined by generating pairs have existed in the GAP library for many years [GAP18, `lib/mgmcong.gi`], but lack sophistication and perform slowly (see Section 2.8.2 for benchmarks). The approach taken in the library is based on an algorithm in [AHT84] for finding the blocks of a transitive permutation group: essentially it consists of repeatedly left- and right-multiplying the generating pairs by generators of the semigroup, and storing the generated relation in a union–find table (see Section 1.13). This approach is essentially the same as the *pair orbit enumeration* algorithm that is described in detail in Section 2.6.1.

This chapter describes a new parallelised approach for computing a congruence from a set of generating pairs, as implemented in libsemigroups [MT$^+$18]. First we will give a general outline of the system and what questions it hopes to answer; then we will describe in detail each algorithm used, its advantages and disadvantages, and when it can be applied. Next we will explain how the different algorithms are executed together, and consider their implementation in libsemigroups; and finally we will show the results of some benchmarking tests which compare its performance to the code in the GAP library.

2.1 Reasons for parallelisation

Parallel processing has seen major advances in the last ten years, with multi-core processors becoming the norm in many types of computers, and processors with 4, 8, or even 16 cores becoming common on a desktop PC. This being the case, it is desirable to parallelise mathematical algorithms wherever possible, and take advantage of the ability to execute multiple threads of instructions concurrently. Some algorithms are "embarrassingly parallel" – that is, they can be split into independent threads which require almost no communication with each other. Examples of these algorithms would be brute force searches, or rendering of computer graphics. These are suited so well to parallelisation that splitting the operation into n parallel threads reduces the expected run-time to barely more than $\frac{1}{n}$ times the expected run-time in a single thread. Other algorithms do not parallelise so well: sometimes threads have to communicate, or use shared resources, causing significant slowdown and severely limiting the improvements that can be made by parallelising.

When it comes to computing information about a congruence from generating pairs, there are various different approaches that can be taken: in Sections 2.6.1, 2.6.2 and 2.6.3, we describe three possible algorithms: pair orbit enumeration, the Todd–Coxeter algorithm, and the Knuth–Bendix algorithm. Depending on what sort of semigroup is given as an input (see Section 2.4), several or all of these might be appropriate. However, depending on certain properties of the congruence, one might perform far better than another. For example, the pair orbit algorithm works well on congruences that contain few non-reflexive pairs, while the Todd–Coxeter algorithm tends to work well on congruences with few classes (i.e. very many pairs). For a detailed analysis of which algorithms perform well on which inputs, see Section 2.8. Given only a set of generating pairs, these properties are likely to be unknown in advance, which makes it difficult to choose a good algorithm.

The natural answer to this problem is the core concept of this chapter: a parallel approach which does not attempt to parallelise individual algorithms, but which runs several known algorithms at the same time, each in a different thread, and simply halts all threads as soon as any one completes. Since these algorithms do not interact with each other in any way, the total run-time will be close to the minimum run-time of all the different algorithms. This is particularly important, since for certain semigroups and congruences some algorithms will never terminate, while others may terminate in a very short time.

2.2 Applicable representations of a semigroup

A semigroup can be represented computationally in different ways. For example, a semigroup of transformations could be specified by a set of transformations that generates it, or alternatively by a finite presentation. Which representation is used affects which methods will be most effective, or even which methods will be applicable. For this purpose, we consider two different categories of semigroup representation: finite presentations, and *concrete* representations. Recall Section 1.7 for the definitions of terms surrounding finite presentations.

Definition 2.1. Let S be a semigroup. A **concrete representation** for S is one of the following:

- the Cayley table of S, where S is finite;
- a finite generating set for S, whose elements are any of the following:
  - partial transformations of finite degree (which might include transformations, partial permutations, and permutations);
  - bipartitions of finite degree;
  - partitioned binary relations of finite degreee (as described in [EENFM15, §2.1]);
  - $n \times n$ boolean matrices for some $n \in \mathbb{N}$ (as described in [BH97]).

The motivation behind Definition 2.1 is to characterise a data structure that has certain convenient computational properties. Observe firstly that only finite semigroups can have concrete representations. Note also that we can use any concrete representation to produce a list of all the elements of S, and to find the product of any two elements, in a finite amount of time, as we will now describe.

Let us first consider the most trivial type of concrete representation: a finite semigroup's Cayley table (see Definition 1.8). Our Cayley table has a finite number of rows and columns, and therefore we know immediately that it must describe a finite semigroup. A list of elements of the semigroup can be taken directly from the indices of the table, and the product xy of two elements x and y can be found by simply reading the entry in row x and column y. Hence Cayley tables have the convenient properties we describe above, but in practice in computation they are used very little, since they require a great deal of space to store – order $O(|S|^2)$.

It will be more common for us to consider generating sets of partial transformations. If S is a semigroup of finite-degree partial transformations, then a finite set of generators for S is a concrete representation of S. Two elements can be multiplied and compared without reference to the semigroup as a whole: for example, if $x = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 3 & 4 & 2 \end{pmatrix}$ and $y = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 4 & 2 & 2 \end{pmatrix}$ then we can calculate $xy = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 4 & 2 & 4 \end{pmatrix}$ without knowing anything else about S. Since all elements in this semigroup have finite degree, and since there are only a finite number of generators, we know immediately that the semigroup in question is finite, and we can produce a list of its elements using, for example, the Froidure–Pin algorithm (see Section 2.5). Hence, a generating set for a semigroup of partial transformations also has the properties described above. Note that bipartitions, partitioned binary relations, and boolean matrices have similar properties.

A finite presentation, on the other hand, is an example of a semigroup representation that is not concrete, and which may not possess the convenient properties above. If $\langle X \mid R \rangle$ is a finite presentation, and S is the semigroup it defines, then it may be that S is infinite, and so $\langle X \mid R \rangle$ would not have the attractive properties of a concrete representation – consider, for example, the presentation $\langle a \mid \varnothing \rangle$ with one generator and no relations, which presents the free semigroup $\{a\}^+$; it has an infinite number of elements, and so an attempt to produce a full list of them will never complete in finite time. Certain finite presentations do define finite semigroups – indeed, in Section 2.5 we give an algorithm for finding a presentation for a finite semigroup. However, it cannot even be determined that a presentation defines a finite semigroup without performing some processing, for example running the Todd–Coxeter algorithm. In this way, finite presentations may not be as useful as concrete representations, and so will be treated differently in some of the algorithms below.

Since some finite presentations define infinite semigroups, the algorithm described in this chapter, whose outputs are described in Section 2.4, cannot be guaranteed to complete in all cases. The only guarantee that can be given is that an answer will be returned if S happens to be finite. In the concrete case, semigroups are always finite and so an answer is guaranteed; but in the case of a finite presentation, it is unknown in advance whether the semigroup it defines is finite or not. Hence, the user of this algorithm might not know whether a particular run is guaranteed to terminate, since a run that is about to finish is indistinguishable from one that will run forever. In some cases, however, we may be able to return an answer even if S is infinite. For a fuller explanation of presentations and decidability issues, see Sections 1.7 and 1.12.

Finitely presented monoids are treated as equivalent to finitely presented semigroups, since a monoid presentation $\langle X \mid R \rangle$ can be easily converted into an equivalent semigroup presentation: an extra generator ε should be added to X, and relations $(x\varepsilon, x)$ and $(\varepsilon x, x)$ added to R for each generator $x \in X$.

2.3 Program inputs

Our algorithm determines the properties of a single left, right, or two-sided congruence defined by generating pairs, over a semigroup S. For the remainder of this chapter, the word "congruence" will be used to refer to left, right, and two-sided congruences equally, without having the default meaning of "two-sided congruence".

If S has a concrete representation (as described in Section 2.2) then we will certainly have a generating set for S (which may be all the elements in S); in this case, it is quick to use these generators to calculate a list of elements in S, along with left and right Cayley graphs for S, using the Froidure–Pin algorithm (see Section 2.5). The Froidure–Pin algorithm also gives us, for each element $s \in S$, a word $w \in X^+$ which *represents* s in the sense of Figure 1.48. We use these words to define a **factorisation function** $f : S \to X^+$ which maps each s to its corresponding word w.

If, on the other hand, we only have a finite presentation for S, then the elements will not be known in advance. In either case, a finite presentation $\langle X \mid R \rangle$ can be given – a technique for efficiently finding a presentation from a concrete representation is given in Section 2.5. The exact parameters supplied to the algorithm are therefore as follows:

- A set of generators X;
- A finite set of relations $R \subseteq X^+ \times X^+$;
- A finite set of generating pairs $W \subseteq X^+ \times X^+$;
- A record of whether we are computing a left, right, or two-sided congruence.

The following are also available only in the case of a concrete representation:

- A list of elements of S;
- Left and right Cayley graphs for S (see Definition 1.12);
- A factorisation function $f : S \to X^+$.

We shall now make clear the meanings of these different parameters, by giving a complete description of the system, starting with the commutative diagram in Figure 2.2.

Here X is our alphabet, and X^+ is the free semigroup it defines. We have a set of relations $R \subseteq X^+ \times X^+$, which generates the two-sided congruence $R^\sharp$ on X^+. This two-sided congruence gives rise to a quotient semigroup $X^+/R^\sharp$; this is isomorphic to the semigroup S, which is described by the presentation $\langle X \mid R \rangle$. The congruence also gives us its natural homomorphism $\pi : X^+ \to S$ (see Definition 1.25). We also have a set of generating pairs $\mathbf{P} \subseteq S \times S$, which defines a left, right, or two-sided congruence $\mathbf{P}^\lhd$, $\mathbf{P}^\rhd$, or $\mathbf{P}^\sharp$. The aim of the algorithm described in this chapter is to obtain a data structure describing this congruence, where the precise meaning of "data structure" is defined in Section 2.4. If we are calculating a two-sided congruence, then it gives rise to the quotient semigroup $S/\mathbf{P}^\sharp$.

$$\begin{tikzcd}
X \arrow[d, hook] \arrow[dr] & & \\
X^+ \arrow[r, two heads, "\pi"] & \frac{X^+}{R^\sharp} \cong S \arrow[r, two heads] & \frac{S}{\mathbf{P}^\sharp}
\end{tikzcd}$$

Figure 2.2: How input objects relate to each other.

The generating pairs $\mathbf{P}$ are not given by the user. Since the elements of S might be unknown (for example if S was specified by a finite presentation), it would be impractical for the user to specify them precisely. Instead, the user specifies a set W consisting of pairs of words from $X^+ \times X^+$, which can be evaluated to pairs of elements in $S \times S$, giving the set of generating pairs $\mathbf{P}$. More formally, let $\Pi : X^+ \times X^+ \to S \times S$ be defined by $\Pi : (w_1, w_2) \mapsto (w_1\pi, w_2\pi)$, where π is the natural homomorphism from X^+ to S mentioned above. The generating pairs $\mathbf{P}$ of the congruence are given by $\mathbf{P} = W\Pi$. This relationship is summarised in Figure 2.3.

$$\begin{tikzcd} W \arrow[r, two heads, "\Pi|_W"] \arrow[d, hook] & \mathbf{P} \arrow[d, hook] \\ X^+ \times X^+ \arrow[r, two heads, "\Pi"] & S \times S \end{tikzcd}$$

Figure 2.3: How generating pairs are specified.

2.4 Program outputs

Each method we are about to explain can provide a variety of different pieces of information, but it is important to consider which questions we aim to answer. Our system should be able to return the following information about a given congruence when requested:

(i) An algorithm to determine whether a given pair (x, y) is in the congruence;

(ii) The number of congruence classes;

(iii) An algorithm that takes an element x and returns the index of the congruence class to which it belongs (it should return the same index for elements x and y if and only if (x, y) is in the congruence);

(iv) A list of the elements in each non-trivial congruence class (only if all such classes are finite).

Each of our algorithms will produce a data structure that can be used to compute these four pieces of information. However, note that not every algorithm can produce all four pieces of information in all cases. For example, if S is infinite, the Knuth–Bendix algorithm cannot be used to compute (ii), and the Todd–Coxeter algorithm cannot be used to produce (iv). After each algorithm is described in Section 2.6, we will explain how that algorithm can be used to produce each one of these four outputs, and we will also explain the situations in which a given algorithm cannot produce a certain output, while pointing out which alternative algorithm will be successful instead.

Note that item (iv) can only be produced by any algorithm if the list happens to be finite – that is, only if all but finitely many elements of the semigroup lie in singletons.

2.5 Finding a presentation

Recall from Section 2.3 that a concrete representation for a semigroup may not include a finite presentation. In order to use the Todd–Coxeter and Knuth–Bendix algorithms, a finite presentation is required, and so a presentation $\langle\, X \mid R \,\rangle$ must be calculated for S. For the purposes of the algorithm described in this chapter, it is not important how this presentation is obtained. However, for the sake of completeness, we will briefly discuss how a presentation could be computed.

We will start by describing a very simple way of producing a presentation from a concrete representation: by using its Cayley table directly.

Method 2.4. Let S be a semigroup with concrete representation, and assume we have access to its Cayley table. We can produce a presentation as follows. Let $\bar{S}$ be a set with the same cardinality as S, containing an element $\bar{x}$ for each element $x \in S$, and let $R \subseteq \bar{S} \times \bar{S}$ be equal to $\{(\bar{x}\bar{y}, \overline{xy}) : (x, y) \in S \times S\}$. The resulting presentation $\langle \bar{S} \mid R \rangle$ defines the semigroup S.

Proof. Consider the natural homomorphism $\pi : \bar{S}^+ \to S$ which maps a word $\bar{x}_1\bar{x}_2 \ldots \bar{x}_n$ to the semigroup element $x_1x_2 \ldots x_n$. Note that π is surjective, since for any element $x \in S$ we have $(\bar{x})\pi = x$. To prove that $\langle \bar{S} \mid R \rangle$ defines S we must show that for any two words $u, v \in \bar{S}^+$, $(u, v) \in R^\sharp$ if and only if $(u)\pi = (v)\pi$.

Let $u, v \in \bar{S}^+$ such that $(u, v) \in R^\sharp$. As in Theorem 1.39, there must be a chain of words

$$u = w_1 \to w_2 \to \cdots \to w_k = v$$

such that $w_i = xay$ and $w_{i+1} = xby$ for some words $x, y \in \bar{S}^*$ and $a, b \in \bar{S}^+$ where either (a, b) or (b, a) is in R, for each $i \in \{1, \ldots, k-1\}$. In each of these steps, (a, b) or (b, a) being in R implies that $(a)\pi = (b)\pi$, because of the way in which R was created. Since π is a homomorphism, this tells us that $(xay)\pi = (xby)\pi$, and therefore that $(u)\pi = (v)\pi$, proving this implication.

For the converse, let $u, v \in \bar{S}^+$ such that $(u)\pi = (v)\pi$. Let $u = \bar{u}_1\bar{u}_2 \ldots \bar{u}_k$ and $v = \bar{v}_1\bar{v}_2 \ldots \bar{v}_l$, with each $\bar{u}_i$ and $\bar{v}_j$ being from $\bar{S}$. Since R has a relation $(\bar{x}\bar{y}, \overline{xy})$ for each pair $(\bar{x}, \bar{y}) \in \bar{S} \times \bar{S}$, there exists a relation $(\bar{u}_1\bar{u}_2, \overline{u_1u_2}) \in R$, and so by repetition we have $(u, u') = (\bar{u}_1\bar{u}_2 \ldots \bar{u}_k, \overline{u_1u_2 \ldots u_k}) \in R^\sharp$, relating the word u with length k to a word u' with length 1, such that $(u)\pi = (u')\pi$. We can perform a similar process on v to produce a word v' also with length 1, with $(v)\pi = (v')\pi$. Since $(u')\pi = (u)\pi = (v)\pi = (v')\pi$ and both u' and v' have length 1, we must conclude that $u' = v'$, and so we find that

$$u \; R^\sharp \; u' = v' \; R^\sharp \; v,$$

so $(u, v) \in R^\sharp$ as required. ∎

Consider the following example.

Example 2.5. Let $\mathcal{T}_2$ be the full transformation semigroup on 2 points. It has 4 elements,

$$\left\{ \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 2 & 2 \end{pmatrix} \right\},$$

which we will relabel as $\{a, b, c, d\}$. The Cayley table is

	a	b	c	d
a	a	b	c	d
b	b	a	c	d
c	c	d	c	d
d	d	c	c	d

Method 2.4 converts this Cayley table to the presentation

$$\begin{aligned} \langle\, a, b, c, d \mid aa = a,\ ab = b,\ ac = c,\ ad = d,& \\ ba = b,\ bb = a,\ bc = c,\ bd = d,& \\ ca = c,\ cb = d,\ cc = c,\ cd = d,& \\ da = d,\ db = c,\ dc = c,\ dd = d\,\rangle,& \end{aligned}$$

which has 4 generators and 16 relations.

This approach is simple to describe, but of course results in a large, unwieldy presentation which will be difficult to use in computations: it will have $|S|$ generators and $|S|^2$ relations, which is likely to be far more than necessary, and will therefore slow down the Todd–Coxeter and Knuth–Bendix algorithms badly. If S is $\mathcal{T}_5$, the full transformation semigroup on 5 points, it has only 3125 elements, but the presentation produced would have 3125 generators and $9,765,625$ relations, an absurdly large representation for a semigroup which can be generated by just three transformations of degree 5. We therefore might consider an alternative.

If S has a known generating set X, we can use it to produce a presentation which is likely to be much smaller. Consider the following approach, which is adapted from the simplified version of the algorithm shown in [FP97, §3.1], where its correctness is proven in more detail.

Method 2.6. Let S be a finite semigroup with a concrete representation, and let $X \subseteq S$ be a generating set for S. We can produce a presentation $\langle\, X \mid R \,\rangle$ consisting of the known generating set X together with some relations R which are computed as follows, with complete pseudo-code in Algorithm 2.7.

Let R begin empty (line 2), and start enumerating the elements of the monoid $S^{(1)}$. We keep a list L_S of elements that have been found in $S^{(1)}$ (initially just the identity id, as in line 3), and we keep another list L_{X^*} of words made up of generators from X (initially just the empty word ε, as in line 4). We also define a function $\nu : X^* \to S^{(1)}$ which maps a word $w = x_1x_2\dots x_n$ to the element $x_1 \cdot x_2 \cdots x_n$ found by multiplying the generators (and $\varepsilon \mapsto \mathrm{id}$ as a special case). We now loop over the words w that have been found and added to L_{X^*}; initially this list contains only one word, but in each iteration of the loop, we may discover new words, which we add to the end of the list, and iterate over in course, until we have considered all words (line 18).

In each iteration of the loop, we take the next word $w \in L_{X^*}$ (line 8) and for each generator $x \in X$ in turn we consider the new word wx made by appending x to the end of w. If this new word represents an element of $S^{(1)}$ that has already been found – that is, if $(wx)\nu \in L_S$ – then we add a new relation (wx, w') to R, where w' is the word we have already stored that represents $(wx)\nu$ (lines 10–12); we do not add anything to L_{X^*}, ensuring that all the words in L_{X^*} describe distinct elements (as asserted in line 16). If, on the other hand, this new word does not represent an element that we already know, then we have encountered a new element which must be added to L_S (line 14) and we have a word that represents it, which must be added to L_X* (line 15); this ensures that $(wx)\nu \in L_S$ for all $x \in X$ at the end of this run of the repeat-loop, which is therefore true for all $u \in L_{X^*}$ considered so far (as asserted in line 17).

Once the loop is run to the end, this if–else statement (lines 10–15) ensures that every word $w \in X^*$ can be rewritten by relations in R to a word in L_{X^*}, and that every element in S has precisely one word in L_{X^*} which represents it. Hence the resulting presentation $\langle\, X \mid R \,\rangle$ defines the semigroup S.

We can also see that this algorithm always terminates for finite X and finite S: the repeat-loop is only run once for each element of S, and the for-loop inside is only run once for each generator in X. Hence the loop must complete, and the algorithm halts.

We can improve further on this method. The libsemigroups implementation of this chapter's algorithm uses the Froidure–Pin algorithm [FP97], a method which is essentially a more advanced variation of Algorithm 2.7. The Froidure–Pin algorithm takes a concrete set of gen-

Algorithm 2.7 The PresentationFromGenerators algorithm

```
 1: procedure PresentationFromGenerators($X$)
 2:     $R := \varnothing$
 3:     $L_S := \{\mathrm{id}\}$
 4:     $L_{X^*} := \{\varepsilon\}$
 5:     $i := 0$                                    ▷ Number of words we have looped over
 6:     repeat
 7:         $i \leftarrow i + 1$
 8:         $w :=$ the $i$th element of $L_{X^*}$
 9:         for $x \in X$ do
10:             if $(wx)\nu \in L_S$ then                   ▷ New word for known element
11:                 $w' :=$ the unique element in $L_{X^*}$ such that $(w')\nu = (wx)\nu$
12:                 Add $(wx, w')$ to $R$
13:             else                                  ▷ Previously unknown element
14:                 Add $(wx)\nu$ to $L_S$
15:                 Add $wx$ to $L_{X^*}$
16:             ▷ $(u)\nu \neq (v)\nu$ for all $u, v \in L_{X^*}$
17:         ▷ $(ux)\nu \in L_S$ for all $x \in X$ and the first $i$ elements $u$ in $L_{X^*}$
18:     until $i = |L_{X^*}|$                          ▷ There are no words left to consider
19:     return $\langle X \mid R \rangle$
```

erators X for a semigroup S, and returns several useful pieces of information:

- a left Cayley graph for S with respect to X;
- a right Cayley graph for S with respect to X;
- a confluent terminating rewriting system R describing the elements of S as words in X^+ (see Section 2.6.3);
- and for each element $s \in S$, a word $w \in X^+$ representing one possible factorisation of s.

The right Cayley graph can be used by the Todd–Coxeter procedure to pre-fill its table (see Section 2.6.2). But more importantly, the rewriting system R is a set of pairs which can be used as the relations in a finite presentation $\langle X \mid R \rangle$ for the semigroup S. Rewriting systems will be defined later in Section 2.6.3, along with the terms "confluent" and "terminating". The fact that R is confluent and terminating may also be useful when it is used as part of a rewriting system in the Knuth–Bendix process, as we will see in Theorem 2.42.

A full description of the Froidure–Pin algorithm is outside the scope of this thesis, but for more information about the algorithm and its implementation in libsemigroups, see [FP97] and [JMP18].

The presentation produced by the algorithms above defines a finite semigroup, and therefore has decidable word problem. Hence, when we have a concrete representation for a semigroup S, we can always find a presentation in which we can compare two words and say whether they represent the same element of S. However, if we started with a finite presentation instead of a concrete representation, we may not have this guarantee. There are many examples of presentations for which the word problem is undecidable (see Examples 1.90 and 1.91). Many presentations do have decidable word problem (see Example 1.89), but many do not, and there is no algorithm to decide whether a given presentation does. Besides, finite presentations, even if their word problem is decidable, can describe infinite semigroups, and the finiteness of S is also not known in advance – nor, in general, is it decidable.

2.6 The methods

2.6.1 Pair orbit enumeration

The first method we will describe is *pair orbit enumeration.* This rather simple algorithm consists of taking the pairs in $\mathbf{R}$, and for each pair (a, b) finding all pairs (as, bs) and (sa, sb) for all $s \in S$. We might refer to this set of pairs as the *orbit* of $\mathbf{R}$ in S, which justifies the name. Although this algorithm is simple and in some ways inefficient, there are cases in which it out-performs the other algorithms in this chapter, so it is worth including in our parallelised method.

We will now give a brief description of pair orbit enumeration – a pseudo-code description is shown in Algorithm 2.8. We start with a semigroup S, a set of generators X for S, and a set of generating pairs $\mathbf{R}$ for the congruence ρ we are trying to compute. It will also be important to remember whether we are calculating a left, right, or two-sided congruence, a piece of information encoded in Algorithm 2.8 as $\sigma \in \{L, R, T\}$. We start with a list of pairs $\mathbf{R}'$ which we initialise to be equal to the set of generating pairs $\mathbf{R}$ (line 3); this list $\mathbf{R}'$ will hold all the pairs that have been found so far, except those we infer from reflexivity, symmetry and transitivity. We will also create a union–find table (Λ, τ) for storing the classes (line 4), and we will be using the three operations AddElement, Union and Find to modify them. See Section 1.13 for a full description of the union–find method.

Now we can describe the overall structure of the pair orbit enumeration method. We begin iterating through the pairs in $\mathbf{R}'$ (the repeat loop starting on line 6), and as we do so we will add further pairs to $\mathbf{R}'$ which will also need to be iterated on, until we reach the end of the list (line 21). For each pair (a, b) we first need to merge the congruence classes of a and b in the union–find table using Union (line 13), ensuring that Find$(a) =$ Find(b) (line 20). However, if either a or b is not already in Λ, then it will need to be added first using AddElement (lines 9–12). Now, for each generator x, we can find the two pairs (xa, xb) and (ax, bx). Depending on whether we are calculating a left, right, or two-sided congruence (that is, depending on the value of σ) we add one or both of these pairs to $\mathbf{R}'$ to be processed in its own turn (lines 16 and 18); note that no pair is ever added from outside $\mathbf{R}^c$ (line 19).

Once this procedure is finished, we have a complete table (Λ, τ) which describes the congruence's non-trivial classes. To check whether a pair (a, b) lies in the congruence, we now just need to look up a and b in the table using Find, and if they lie in the same class we return true. If either element has not been added to Λ, then it lies in a singleton, and (a, b) is in the congruence if and only if $a = b$.

In order to prove that this method is valid, we will recall some facts from Chapter 1, as well as citing several sources for some theory. Let S be a semigroup, let $\mathbf{R} \subseteq S \times S$, let $\sigma \in \{L, R, T\}$, and let ρ be the left, right or two-sided (according to σ) congruence on S generated by $\mathbf{R}$. We can now state the following theorem.

Theorem 2.9. *Let* $(\Lambda, \tau) =$ PairOrbit$(S, \mathbf{R}, \sigma)$. *Distinct elements x and y in Λ lie in the same class of ρ if and only if* Find$(x) =$ Find(y).

Algorithm 2.8 The PairOrbit algorithm

```
Require: $S$ a semigroup, $\mathbf{R} \subseteq S \times S$, $\sigma \in \{L, R, T\}$
 1: procedure PairOrbit($S, \mathbf{R}, \sigma$)
 2:     Let $X$ be a generating set for $S$
 3:     $\mathbf{R}' := \mathbf{R}$
 4:     $(\Lambda, \tau) := (\varnothing, \varnothing)$                ▷ An initialised union–find table
 5:     $i := 0$                                      ▷ Number of pairs we have looped over
 6:     repeat
 7:         $i \leftarrow i + 1$
 8:         $(a, b) :=$ the $i$th pair in $\mathbf{R}'$
 9:         if $a \notin \Lambda$ then
10:             AddElement($a$)
11:         if $b \notin \Lambda$ then
12:             AddElement($b$)
13:         Union($a, b$)
14:         for $x \in X$ do
15:             if $\sigma \in \{L, T\}$ then
16:                 Add $(xa, xb)$ to $\mathbf{R}'$ if not already present
17:             if $\sigma \in \{R, T\}$ then
18:                 Add $(ax, bx)$ to $\mathbf{R}'$ if not already present
19:             ▷ $\mathbf{R}' \subseteq \mathbf{R}^c$ (also, $\mathbf{R}' \subseteq \mathbf{R}^l$ if $\sigma = L$, and $\mathbf{R}' \subseteq \mathbf{R}^r$ if $\sigma = R$)
20:         ▷ Find($a$) = Find($b$) for the first $i$ pairs $(a, b)$ in $\mathbf{R}'$
21:     until $i = |\mathbf{R}'|$                        ▷ There are no pairs left to process
22:     return $(\Lambda, \tau)$
```

Proof. Recall from Definition 1.34 the relations $\mathbf{R}^c$, $\mathbf{R}^l$ and $\mathbf{R}^r$, and recall that they are respectively the smallest compatible, left-compatible, and right-compatible relations which contain $\mathbf{R}$ (see Lemma 1.35).

Let us start by considering the case when $\sigma = L$, that is the case where we are computing the left congruence. For each pair $(a, b) \in \mathbf{R}$, the pair orbit enumeration procedure finds all the pairs (xa, xb) where x is a generator of S. That pair is then added to $\mathbf{R}'$ (line 16), and so it is kept in line for processing. On a later run through the repeat loop, when i is incremented in line 7 and reaches the position of (xa, xb) in $\mathbf{R}'$, (xa, xb) is considered as a pair in its own right, and all of its left multiples are found in their turn: (yxa, yxb) for every generator $y \in X$. In this way, (sa, sb) is found for every $s \in S$, so the set of pairs found by the PairOrbit algorithm is

$$\{(sa, sb) \mid (a, b) \in \mathbf{R}, s \in S\},$$

which is equal to the relation $\mathbf{R}^l$. Similarly, if $\sigma = R$, the algorithm finds all the pairs in $\mathbf{R}^r$, and if $\sigma = T$, the algorithm finds all the pairs in $\mathbf{R}^c$.

The remainder of the proof considers the union–find method. As described above, since union–find stores pairs as a partition, it automatically takes care of reflexivity, symmetry, and transitivity. That is, if a set of pairs Q is fed into the union–find table using Union on every pair in Q, then the resulting table describes Q^e, the least equivalence relation containing Q. Hence, if $\sigma = L$, the table produced by PairOrbit will describe the relation $(\mathbf{R}^l)^e$; if $\sigma = R$, it will describe $(\mathbf{R}^r)^e$; and if $\sigma = T$, it will describe $(\mathbf{R}^c)^e$. As we know from Theorem 1.39, these relations are respectively the least left, right, and two-sided congruence containing $\mathbf{R}$, so the output of PairOrbit describes ρ accurately, and Find may be used as described to determine whether two elements lie in the same congruence class. □

In PairOrbit, we call AddElement, and thus add elements to Λ, only when an element is found in a pair. This approach opens up the possibility of applying this method to infinite semigroups. Our implementation in libsemigroups [MT+18] includes such a possibility, by taking a semigroup presentation $\langle X \mid R \rangle$ and using the Knuth–Bendix algorithm (see Section 2.6.3) as a way of comparing the semigroup's elements. The pair orbit enumeration procedure, as described above, can then be used, and will complete in finite time if and only if the number of elements in non-trivial congruence classes is finite. It should be noted that we can rely on PairOrbit terminating in a finite number of steps so long as S and X are finite. There are only two loops in the algorithm: the inner for-loop is limited by the finite length of X, so it has only a finite number of steps; and the outer repeat loop can only be run at most $|S|^2$ times, since it runs once for each pair added to $\mathbf{R}'$, which is a subset of $S \times S$ – note that new pairs are only added if they are not already present (lines 16 and 18).

This is the simplest of the algorithms discussed in this section, and generally does not perform as well as the others in practice. However, it has certain advantages such as its small overhead and quick setup, and therefore on certain examples performs better than the others. Furthermore, it is applicable to any problem discussed in this chapter, whether we have a concrete representation or a finite presentation, and whether ρ is a left, right or two-sided congruence. Uniquely among the methods described in this chapter, it does not even require a presentation in order to be run.

Now that we have explained the pair orbit algorithm, we should consider how it can be used to answer the questions in Section 2.4. Assume we have a congruence ρ on a semigroup S, and the pair orbit algorithm has been completed. For (i), we can determine whether a given pair (x, y) lies in ρ fairly easily using the resultant union–find table: $(x, y) \in \rho$ if and only if $\textsc{Find}(x) = \textsc{Find}(y)$. For (ii), to find the number of congruence classes of ρ, we first find the number of different blocks in the union–find table (that is, the number of different values returned by Find when given elements in Λ), and then we add the number of singletons (that is, the number of elements not in Λ). Hence the number of congruence classes of ρ is equal to

$$|\{\textsc{Find}(x) : x \in \Lambda\}| + |S \setminus \Lambda|.$$

For (iii), we require an algorithm that takes an element and returns a unique integer corresponding to the congruence class in which it lies; we can simply use the function Find and create a map of outputs to integers over successive calls; for a fuller description, see the ClassNo function described in Section 2.6.3 under "Computing program outputs" (here we use Find instead of a rewriting system). Finally for (iv) we require a list of the elements in each non-trivial congruence class; this can be produced by calling Find on each element x in Λ, and putting x into a list labelled L_i where $i = \textsc{Find}(i)$, creating new lists as necessary – once all elements have been considered, these lists are what is required.

This algorithm has rather high complexity. As we saw in the proof of Theorem 2.9, the algorithm adds pairs to $\mathbf{R}'$ until it is equal to $\mathbf{R}^c$ (for a two-sided congruence – substitute $\mathbf{R}^l$ or $\mathbf{R}^r$ for a left or right congruence). The repeat-loop will be therefore be executed once for each pair in $\mathbf{R}^c$, and it contains a for-loop that iterates over X. Hence, even if we treat calls to Union as close to constant time (see Theorem 1.97) we can only say that PairOrbit has time complexity order $O(|\mathbf{R}^c|\cdot|X|)$. Hence, the algorithm can complete quite quickly for congruences which have a small generating set X and only a few pairs in $\mathbf{R}^c$ (that is, with many classes). However, in the worst case, when the generating set is large and there are many pairs in $\mathbf{R}^c$, the time complexity is order $O(|S|^3)$, since $\mathbf{R}^c = S \times S$ and $X = S$. Similarly, to store all the pairs that are found in $\mathbf{R}^c$ the space complexity in this worst case is $O(|S|^2)$. This makes the algorithm unattractive for congruences with many pairs, but in some cases the algorithm can still out-perform the others in this chapter (see Section 2.8).

Some improvements can be made to the basic algorithm shown above. For example, the libsemigroups implementation makes use of the symmetry of pairs: a pair (a, b) is not added to $\mathbf{R}'$ if the pair (b, a) has previously been added. Tweaks like this can make the algorithm slightly faster, though they do not address the high complexity of the method.

2.6.2 The Todd–Coxeter algorithm

The Todd–Coxeter algorithm was originally described in 1936 in [TC36]. It was an algorithm to enumerate the cosets of a finitely generated subgroup of a finitely presented group. Arriving before the advent of electronic computers, the algorithm was originally intended to be carried out by hand. Perhaps the earliest automatic implementation was on the EDSAC II computer in Cambridge [Lee63]. Since then, a wide variety of efficient, optimised versions have been implemented, one notable example being ACE [RH09].

A variation of the Todd–Coxeter algorithm for semigroups was described in 1967 [Neu67]. The algorithm takes a presentation $\langle X \mid R \rangle$ for a semigroup S and computes the right regular representation of $S^{(1)}$ with respect to the generators X – that is, it computes all the elements of $S^{(1)}$ and the result of right-multiplying each element by each generator (see Section 1.11.1). Since the original algorithm makes very little use of those properties unique to groups, the method applied to semigroups is essentially the same. Other descriptions of the Todd–Coxeter algorithm for semigroups can be found in [Ruš95, Chapter 12] and [Wal92, Chapter 1.2], and a variation specific to inverse semigroups can be found in [Cut01]. Our version of the algorithm is based closely on an implementation by Götz Pfeiffer, found in [GAP18, `lib/tcsemi.gi`], itself based on [Wal92].

We will now describe the Todd–Coxeter method as used in the context of this chapter. Though the Todd–Coxeter method itself does not represent new work, it is an important part of the overall parallel approach, and in order to understand its uses and limitations, it is described here in full. The idea of pre-filling the table is original work (see "Pre-filling the table" below), as is the integration into the overall parallel algorithm this chapter describes.

Setup

The Todd–Coxeter algorithm is based on a table, where each row corresponds to a single congruence class (or equivalently in the case of a two-sided congruence, a single element of the quotient semigroup). The columns of the table correspond to the generators of the semigroup, and the entry in row i, column j represents the element found by taking element i and right-multiplying it by generator j. These entries may be blank, and two different rows may be found to describe the same element. Mathematically, we can view this table as a triple $(n, \mathbf{N}, \tau)$ consisting of:

- an integer $n \in \mathbb{N}$ representing the number of rows in the table;
- a set $\mathbf{N} \subseteq \{1, \dots, n\}$ containing the indices of the *undeleted* rows; and
- a function $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$, where $(i, x)\tau$ is equal to the entry in row i and the column corresponding to generator x – with 0 representing a blank entry.

Suppose that we have a semigroup presentation $\langle X \,|\, R \rangle$ for a semigroup S. The table is initialised with a single row, numbered 1. This row corresponds to the empty word ε, or the adjoined identity of the monoid $S^{(1)}$. The row is empty, containing a blank entry in all $|X|$ columns. In our mathematical notation, we define $n = 1$ and $(1, x)\tau = 0$ for all $x \in X$.

We can naturally extend the function $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$ to a function $\bar{\tau} : \mathbf{N} \times X^* \to \mathbf{N} \cup \{0\}$ which is described as follows. If $w \in X^*$ and $w = w_1 \dots w_n$, where $w_1, \dots, w_n \in X$, then we can define $\bar{\tau}$ recursively by

$$(i, w)\bar{\tau} = \begin{cases} i & \text{if } w = \varepsilon, \\ 0 & \text{if } (i, w_1)\tau = 0, \\ \big((i, w_1)\tau, w_2 \dots w_n\big)\bar{\tau} & \text{otherwise.} \end{cases}$$

The effect of $\bar{\tau}$ is to trace an entire word through the table, starting at a given row.

Elementary operations

We now describe three operations which may be applied to the table. These operations will be described in turn to give an understanding of what they are designed to do, along with their description in pseudo-code (Algorithms 2.10, 2.11 and 2.12). We will then describe the overall Todd–Coxeter procedure which uses these operations to find all the elements of a semigroup from a presentation (Algorithm 2.13). The three operations are:

- Add: Fill in a blank entry and add a row to the table;
- Trace: Trace a relation from a row;
- Coinc: Process a coincidence.

The first operation, Add, is the simplest of the three, and is shown in pseudo-code in Algorithm 2.10. Calling Add(i, x) should fill in a blank cell in the table in row i and column x – that is, a position such that $(i, x)\tau = 0$. It fills it in with the address of a new row, which must first be created. We add a new row at the bottom of the table by incrementing the number of rows n (line 2), adding its new value to the list of active rows $\mathbf{N}$ (line 3), and filling in all its entries with blanks – that is, setting its τ-outputs to 0 (lines 4–5). Finally, the address of the new row is written into the blank cell that was specified – that is, we set $(i, x)\tau$ to the new row's address n (line 6). Now the blank cell has been filled with the address of a new row, as required.

Trace takes two arguments: a row e in the table, and a relation $v = w$ from R. Its goal is to ensure that, starting at e, applying the word v has the same result as applying the word w – in other words, to ensure that $(e, v)\bar{\tau} = (e, w)\bar{\tau}$. We will now describe how this is done, referring to Algorithm 2.11 for pseudo-code.

Algorithm 2.10 The Add algorithm (Todd–Coxeter)

```
Require: $(i, x)\tau = 0$
1: procedure Add$(i, x)$
2:     $n \leftarrow n + 1$
3:     $\mathbf{N} \leftarrow \mathbf{N} \cup \{n\}$
4:     for $x \in X$ do
5:         $(n, x)\tau := 0$
6:     $(i, x)\tau \leftarrow n$
```

First we follow both words through the table, one letter at a time, up to and including their penultimate letter. For v, we use a variable s, which starts at e (line 4) and we go through v one letter v_i at a time up to the second-last letter (line 5). At each step, we consider $(s, v_i)\tau$, the entry in the table which we must follow next to continue going through the word. If it is not set, we call Add to create a new row to point it towards (lines 6–7). Then we follow it, setting s to the new value (line 8). At each stage of the loop, $s = (e, v_1 \dots v_i)\bar{\tau}$ (as asserted on line 9). Hence, at the end of this loop, we only need to follow the last letter v_m to complete the word – that is, $(s, v_m)\tau = (e, v)\bar{\tau}$.

We follow a similar process for w in lines 10–15, going through all letters of w except the last one, and finding a row t such that $(t, w_n)\tau = (e, w)\bar{\tau}$. Now in order to satisfy the objective $(e, v)\bar{\tau} = (e, w)\bar{\tau}$, we just need to ensure that two specific cells in the table are equal: we need to ensure that $(s, v_m)\tau = (t, w_n)\tau$. We do this by considering four different cases:

- if the two cells are both empty, then we apply Add to (s, v_m) to create a new row for $(s, v_m)\tau$ to point to, and then we copy that entry into $(t, w_n)\tau$ (lines 16–18);
- if just one of the entries is empty, then the filled entry is copied into the empty one (lines 19–22);
- if both entries are filled and equal, we do not need to do anything;
- if both entries are filled and are distinct, then we need to force the two rows they point towards to be equal, by applying Coinc to the two entries (lines 23–24).

After each of these cases, the result is that $(s, v_m)\tau = (t, w_n)\tau$, and hence that $(e, v)\bar{\tau} = (e, w)\bar{\tau}$ as required.

Coinc is used when two rows in the table are found to refer to the same element of $S^{(1)}$; it modifies the table to delete one row and use the other instead. Pseudo-code for Coinc can be found in Algorithm 2.12. First, the higher-numbered row s is deleted from the list of active rows ($\mathbf{N} \leftarrow \mathbf{N} \setminus \{s\}$), and all occurrences of the higher number s are replaced by the lower number r (lines 5–6), in every active row (line 3) and every column (line 4) of the table. Next, the two rows are combined into one, with all known information being preserved: any columns that are filled in row s but empty in row r are copied (lines 9–10) and if there is any column that has different non-empty entries in rows r and s (line 11), we know that those two entries refer to the same element, and the coincidence needs to be processed with another call to Coinc (line 12). Hence, for each column considered, a cell in row r can only be empty if that cell in row s is also empty (as asserted in line 13). After the algorithm is finished, all references to s have been removed from the table, and all information from row s has been incorporated into row r.

Algorithm 2.11 The TRACE algorithm (Todd–Coxeter)

```
procedure TRACE($e, v = w$)
    Write $v = v_1 \dots v_m$                                ▷ ($v_i \in X$ for $1 \le i \le m$)
    Write $w = w_1 \dots w_n$                                ▷ ($w_i \in X$ for $1 \le i \le n$)
    $s \leftarrow e$
    for $i \in \{1, \dots, m-1\}$ do
        if $(s, v_i)\tau = 0$ then
            ADD($s, v_i$)
        $s \leftarrow (s, v_i)\tau$
        ▷ $s = (e, v_1 \dots v_i)\bar{\tau}$
    $t \leftarrow e$
    for $i \in \{1, \dots, n-1\}$ do
        if $(t, w_i)\tau = 0$ then
            ADD($t, w_i$)
        $t \leftarrow (t, w_i)\tau$
        ▷ $t = (e, w_1 \dots w_i)\bar{\tau}$
    if $(s, v_m)\tau = (t, w_n)\tau = 0$ then
        ADD($s, v_m$)
        $(t, w_n)\tau \leftarrow (s, v_m)\tau$
    else if $(s, v_m)\tau = 0$ then
        $(s, v_m)\tau \leftarrow (t, w_n)\tau$
    else if $(t, w_n)\tau = 0$ then
        $(t, w_n)\tau \leftarrow (s, v_m)\tau$
    else if $(s, v_m)\tau \neq (t, w_n)\tau$ then
        COINC($(s, v_m)\tau, (t, w_n)\tau$)
```

Algorithm 2.12 The COINC algorithm (Todd–Coxeter)

```
Require: $r < s$
procedure COINC($r, s$)
    $\mathbf{N} \leftarrow \mathbf{N} \setminus \{s\}$
    for $e \in \mathbf{N}$ do
        for $x \in X$ do
            if $(e, x)\tau = s$ then
                $(e, x)\tau \leftarrow r$
        ▷ $(i, x)\tau \neq s$ for all $x \in X$ and $i \in \mathbf{N}$ such that $i \le e$
    for $x \in X$ do
        if $(r, x)\tau = 0$ then
            $(r, x)\tau \leftarrow (s, x)\tau$
        else if $(r, x)\tau \neq (s, x)\tau$ and $(s, x)\tau \neq 0$ then
            COINC($(r, x)\tau, (s, x)\tau$)
        ▷ $(r, x)\tau \neq 0$ unless $(s, x)\tau = 0$
```

Now that we have these three operations, it is simple to describe the overall ToddCoxeter procedure, as shown in Algorithm 2.13. First we set up the table as described above: the number of rows n is set to 1, the set of active rows $\mathbf{N}$ contains just this row 1, and the table τ is initialised to have a blank cell in each column, and just one row (lines 2–6). Now we go through all the rows e in the table, starting with row 1. For each row, we check whether it is active – that is, we check whether e is in $\mathbf{N}$ (line 10). If it is not, we do nothing and proceed with the next row; if it is, we apply relations to it. We go through all the relations from R (line 11), and apply each one to the current row using Trace (line 12). Each call to Trace may, of course, invoke calls to Add and Coinc, so rows will be appended to the table as the algorithm progresses, and it may take many iterations of the repeat-loop before the bottom of the table is reached. When the end is reached (line 15), the table should completely describe the multiplication for the finitely presented semigroup: each row in $\mathbf{N} \setminus \{1\}$ represents one element of S, and $(i, x)\tau$ represents the element denoted by i right-multiplied by the generator x.

Algorithm 2.13 The ToddCoxeter algorithm (for semigroups)

```
 1: procedure ToddCoxeter($\langle X \mid R \rangle$)
 2:     $n := 1$
 3:     $\mathbf{N} := \{1\}$
 4:     $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$
 5:     for $x \in X$ do
 6:         $(1, x)\tau := 0$
 7:     $e := 0$
 8:     repeat
 9:         $e \leftarrow e + 1$
10:         if $e \in \mathbf{N}$ then
11:             for $(u, v) \in R$ do
12:                 Trace($e, u = v$)
13:                 ▷ $(e, u)\bar{\tau} = (e, v)\bar{\tau}$
14:         ▷ $(i, u)\bar{\tau} = (i, v)\bar{\tau}$ for all $(u, v) \in R$ and $i \in \mathbf{N}$ such that $i \leq e$
15:     until $e = n$                                  ▷ There are no rows left to process
16:     return $(n, \mathbf{N}, \tau)$
```

Note that there is no guarantee that the end of $\mathbf{N}$ will ever be reached: if the given presentation defines an infinite semigroup, the table will grow forever and the procedure will never terminate. On the other hand, the procedure is guaranteed to terminate in a finite number of steps if and only if the presentation defines a finite semigroup (see [HEO05, Theorem 5.5] and [BC76, Theorem 3]). This number of steps is, however, unbounded; and since we may not know in advance whether a presentation defines a finite or infinite semigroup, it is impossible to know, while the procedure is running, whether it will end – indeed, see Example 2.25.

Example 2.14. We now give an example of the Todd–Coxeter algorithm running on the semigroup presentation

$$\langle\, a, b \,\big|\, ba = ab,\ b^2 = b,\ a^3 = ab,\ a^2b = a^2 \,\rangle\,.$$

We initialise the table to look like Table 2.15.

	a	b
1		

Table 2.15: Initial position.

The list **N** of undeleted rows contains only a single entry, 1. We begin by tracing each relation on the row 1, starting with $ba = ab$. The left-hand side of this relation makes us call ADD on the cell $(1, b)$, creating a new row, 2, which is added to **N**. For the right-hand side, we must call ADD on the cell $(1, a)$, creating a row 3. At the end of the TRACE, we must set $(1, ba)\bar{\tau}$ equal to $(1, ab)\bar{\tau}$, so we set both $(2, a)\tau$ and $(3, b)\tau$ to 4 (as in Table 2.16).

	a	b
1	3	2
2	4	
3		4
4		

Table 2.16: Position after TRACE$(1, ba = ab)$.

Next, we apply TRACE$(1, b^2 = b)$. Since $(1, b)\bar{\tau}$ is already set, we just set $(1, b^2)\bar{\tau}$ equal to it: $(2, b)\tau \leftarrow 2$. See Table 2.17.

	a	b
1	3	2
2	4	2
3		4
4		

Table 2.17: Position after TRACE$(1, b^2 = b)$.

Still on row 1, we apply TRACE to the third relation, $a^3 = ab$. This creates a new row for $(1, a^2)\bar{\tau} = (3, a)\tau = 5$. The new row's a entry is set to be the same as $(1, ab)\bar{\tau}$, which is 4 (see Table 2.18).

	a	b
1	3	2
2	4	2
3	5	4
4		
5	4	

Table 2.18: Position after TRACE$(1, a^3 = ab)$.

The final relation for row 1 is $a^2b = a^2$. $(1, a^2b)\bar{\tau}$ is currently blank, and is set to the current value of $(1, a^2)\bar{\tau}$, which is 5. Hence $(5, b)\tau \leftarrow 5$ (as in Table 2.19).

	a	b
1	3	2
2	4	2
3	5	4
4		
5	4	5

Table 2.19: Position after TRACE$(1, a^2b = a^2)$.

We have now finished with row 1, and we proceed to the next row in $\mathbf{N}$, which is 2. Accordingly, we apply the first relation, TRACE$(2, ba = ab)$. The value of $(2, ba)\bar{\tau}$ is 4, whereas the value of $(2, ab)\bar{\tau}$ has not yet been set. We set it by applying $(4, b)\tau \leftarrow 4$. See Table 2.20.

	a	b
1	3	2
2	4	2
3	5	4
4		4
5	4	5

Table 2.20: Position after TRACE$(2, ba = ab)$.

Proceeding with TRACE$(2, b^2 = b)$, we find that $(2, b^2)\bar{\tau} = (2, b)\bar{\tau}$ already, so we make no modifications to the table. Next, TRACE$(2, a^3 = ab)$ discovers that $(2, a^2)\bar{\tau}$ is not set, and so we call ADD$(4, a)$, creating a new row 6. Now $(6, a)\tau$ is set to $(2, ab)\bar{\tau}$ which is equal to 4. See Table 2.21.

	a	b
1	3	2
2	4	2
3	5	4
4	6	4
5	4	5
6	4	

Table 2.21: Position after TRACE$(2, a^3 = ab)$.

The final relation for row 2 is $a^2b = a^2$, setting $(6, b)\tau \leftarrow 6$ (see Table 2.22).

	a	b
1	3	2
2	4	2
3	5	4
4	6	4
5	4	5
6	4	6

Table 2.22: Position after all relations on row 2.

Next we move onto row 3, and we apply TRACE$(3, ba = ab)$. Inspecting the table shows $(3, ba)\bar{\tau} = 6$ but $(3, ab)\bar{\tau} = 5$, giving rise to a coincidence. We apply COINC$(5, 6)$, which deletes row 6, rewrites any occurrences of 6 in the table to 5, and copies row 6 into row 5 (yielding no new information). The result is shown in Table 2.23. The rest of the relations are applied to row 3, and to the remaining rows in the table, but no changes are made to the table, so Table 2.23 is the final state.

	a	b
1	3	2
2	4	2
3	5	4
4	~~6~~5	4
5	4	5
~~6~~	~~4~~	~~6~~

Table 2.23: Final position.

We can now delete row 1, which acts as an appended identity, and we find a description of the semigroup's multiplication, with relation to its generators. This description can be represented as a Cayley graph, as shown in Figure 2.24.

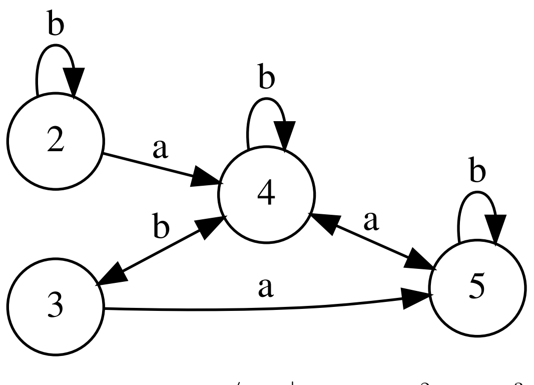


Figure 2.24: Right Cayley graph of $\langle\, a, b \,|\, ba = ab,\ b^2 = b,\ a^3 = ab,\ a^2b = a^2 \,\rangle$.

It is worth noting that the columns of the table now give a right representation of S. That is, S is isomorphic to the semigroup generated by the transformations $\left(\begin{smallmatrix}1&2&3&4&5\\3&4&5&5&4\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}1&2&3&4&5\\2&2&4&4&5\end{smallmatrix}\right)$, as we can see from Theorem 1.63: we have $(a)\phi = \left(\begin{smallmatrix}1&2&3&4&5\\3&4&5&5&4\end{smallmatrix}\right)$ and $(b)\phi = \left(\begin{smallmatrix}1&2&3&4&5\\2&2&4&4&5\end{smallmatrix}\right)$, and these two transformations generate all the elements of S.

We now consider another example, which shows that the Todd–Coxeter procedure can take an unbounded length of time, even when the semigroup it computes is relatively small.

Example 2.25. Let $\langle X \mid R \rangle$ be the semigroup presentation

$$\langle a, b \mid a^{100} = b,\ a = a^2 \rangle .$$

By inspection we can see that any string a^n is equal to a by repeated application of the second relation, and therefore that $a = a^{100} = b$ by the first relation. Hence any string in X^+ is equal to a, showing that $\langle X \mid R \rangle$ defines the trivial semigroup. However, if we apply the Todd–Coxeter procedure to this example, a table with 100 rows will need to be constructed before any coincidences are found, and 99 of these will need to be deleted before the algorithm terminates.

The number 100 in this example can be replaced by any arbitrarily large value; the presentation will still define the trivial semigroup, but computation time will be arbitrarily long. In this way, we can see that although the Todd–Coxeter algorithm is guaranteed to terminate in a finite number of steps when the input defines a finite semigroup, this number of steps is unbounded.

Left, right, and two-sided congruences

Next we will describe how to apply the given Todd–Coxeter method to left, right and two-sided congruences. The description of the Todd–Coxeter algorithm given above is a method for finding the elements of the semigroup S given by the presentation $\langle X \mid R \rangle$, and describing their multiplication. More precisely, the Todd–Coxeter algorithm produces a table in which each row represents one element $s \in S^{(1)}$, each column represents one generator $x \in X$, and the cell in the row of s and the column of x contains the row number of the element sx. Hence its columns describe a right regular representation of S, as described in Section 1.11.1. In the language of congruences, this is equivalent to finding the classes of the trivial congruence on the semigroup $S^{(1)}$, or of the two-sided congruence $R^\sharp$ on the free monoid X^*.

The original problem we wanted to solve, as described in Section 2.3, is to find the classes of the congruence ρ defined by a set of pairs $W \subset X^+ \times X^+$ over the semigroup defined by $\langle X \mid R \rangle$. If we are considering a two-sided congruence, we can simply apply the Todd–Coxeter method described above to the semigroup presentation $\langle X \mid R, W \rangle$, and the result will represent the classes of ρ. We call this method ToddCoxeterTwoSided, and give pseudo-code for it in Algorithm 2.26. Note that this pseudo-code is precisely the same as ToddCoxeter in Algorithm 2.13, except for the addition of lines 11–12 which trace all the relations in W, before doing the same for those in R.

The correctness of the ToddCoxeterTwoSided algorithm is inherited from the correctness of the ToddCoxeter algorithm, which is shown in, for example, [TC36] and [BC76]. What we have precisely computed is the elements of the semigroup presented by $\langle X \mid R, W \rangle$, or in other words, the quotient semigroup S/ρ. Each class of ρ is thus represented by one row in the resulting data structure. Since this algorithm does not produce, for example, a list of all the elements in a given class, we cannot yet say that we have "computed" the congruence in the sense of the program outputs we specified in Section 2.4. However, see the "Computing

Algorithm 2.26 The ToddCoxeterTwoSided algorithm (for congruences)

```
1: procedure ToddCoxeterTwoSided($\langle X \mid R \rangle$, $W$)
2:     $n := 1$
3:     $\mathbf{N} := \{1\}$
4:     $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$
5:     for $x \in X$ do
6:         $(1, x)\tau := 0$
7:     $e := 0$
8:     repeat
9:         $e \leftarrow e + 1$
10:        if $e \in \mathbf{N}$ then
11:            for $(u, v) \in W$ do
12:                Trace($e, u = v$)
13:                ▷ $(e, u)\bar{\tau} = (e, v)\bar{\tau}$
14:            for $(u, v) \in R$ do
15:                Trace($e, u = v$)
16:                ▷ $(e, u)\bar{\tau} = (e, v)\bar{\tau}$
17:        ▷ $(i, u)\bar{\tau} = (i, v)\bar{\tau}$ for all $(u, v) \in R \cup W$ and $i \in \mathbf{N}$ such that $i \leq e$
18:    until $e = n$                                ▷ There are no rows left to process
19:    return $(n, \mathbf{N}, \tau)$
```

program outputs" section below for a description of the additional work required to produce all the information we require about the congruence.

If we are considering a left or right congruence we must alter the method slightly. We shall now describe how to modify the Todd–Coxeter algorithm to compute a table for the right congruence ρ (see Algorithm 2.27 for pseudo-code).

Let $\langle X \mid R \rangle$ and W be as described in Section 2.3, and let ρ be the right congruence they specify. Since $\langle X \mid R \rangle$ specifies the semigroup over which ρ is defined, we must trace all the relations R on every row in the table, as usual (lines 11–18). This ensures that, for a relation $(a, b) \in R$, we have $(i, a)\bar{\tau} = (i, b)\bar{\tau}$ for every $i \in \mathbf{N}$, which is equivalent to the left congruence rule that for any pair $(a, b) \in R$ we have $(sa, sb) \in \rho$ for every $s \in S$. However, such a condition is not required for a right congruence, and so we do not need to enforce the pairs from W so strictly. In fact, we only need to trace the pairs from W from row 1, and not any other row, as we do in lines 7–8 of Algorithm 2.27. The rest of the algorithm is the same as ToddCoxeter (Algorithm 2.13), which is explained in detail above.

To see that this algorithm is correct, consider the following theorem.

Theorem 2.28. *Let $\langle X \mid R \rangle$ and W be as in Section 2.3 and ρ be the right congruence they specify.* ToddCoxeterRight($\langle X \mid R \rangle$, W) *returns a table which describes the classes of ρ (after the identity row 1 is removed).*

Proof. We may safely assume that the basic Todd–Coxeter algorithm is correct [TC36, BC76] and therefore that ToddCoxeterTwoSided($\langle X \mid R \rangle$, W) gives the correct answer for a two-sided congruence, since it is equivalent to running the Todd–Coxeter algorithm on the presentation $\langle X \mid R, W \rangle$. We must now consider how the ToddCoxeterRight algorithm differs from ToddCoxeterTwoSided, and prove that the resulting table does indeed define the right congruence ρ.

Algorithm 2.27 The ToddCoxeterRight algorithm (for right congruences)

```
1: procedure ToddCoxeterRight($\langle X \mid R \rangle$, $W$)
2:     $n := 1$
3:     $\mathbf{N} := \{1\}$
4:     $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$
5:     for $x \in X$ do
6:         $(1, x)\tau := 0$
7:     for $(u, v) \in W$ do
8:         Trace$(1, u = v)$
9:         ▷ $(1, u)\bar{\tau} = (1, v)\bar{\tau}$ for every pair $(u, v)$ in $W$ considered so far
10:    $e := 0$
11:    repeat
12:        $e \leftarrow e + 1$
13:        if $e \in \mathbf{N}$ then
14:            for $(u, v) \in R$ do
15:                Trace$(e, u = v)$
16:                ▷ $(e, u)\bar{\tau} = (e, v)\bar{\tau}$
17:        ▷ $(i, u)\bar{\tau} = (i, v)\bar{\tau}$ for all $(u, v) \in R$ and $i \in \mathbf{N}$ such that $i \leq e$
18:    until $e = n$                                  ▷ There are no rows left to process
19:    return $\tau$
```

ToddCoxeterTwoSided treats pairs from R and pairs from W in essentially the same way: each pair (u, v) is traced starting at *every* active row $e \in \mathbf{N}$, using Trace$(e, u = v)$. ToddCoxeterRight follows this method for pairs in R, but traces pairs in W only from row 1.

For a pair $(u, v) \in W$ we must certainly have $(u, v) \in \rho$. Hence we run Trace$(1, u = v)$ on line 8 to ensure that $(1, u)\bar{\tau} = (1, v)\bar{\tau}$ and therefore the row corresponding to u is the same as the row corresponding to v. However, we must not run Trace$(e, u = v)$ for any numbers e higher than 1, since this would be enforcing the *left* congruence criterion – that $(su, sv) \in \rho$ for all $s \in S$ – which does not apply to the *right* congruence ρ.

The right congruence criterion, that for any pair $(u, v) \in W$ we have $(us, vs) \in \rho$ for all $s \in S$, is automatically enforced without additional work, in the following way. The words us and vs should be in the same congruence class of ρ if and only if $(1, us)\bar{\tau} = (1, vs)\bar{\tau}$; but we already know that $(1, u)\bar{\tau} = (1, v)\bar{\tau}$, because it was enforced using Trace on line 8, and so we have

$$(1, us)\bar{\tau} = ((1, u)\bar{\tau}, s)\bar{\tau} = ((1, v)\bar{\tau}, s) = (1, vs)\bar{\tau},$$

which is enough to show that $(us, vs) \in \rho$ as required. □

In summary, when computing a right congruence with the Todd–Coxeter algorithm, each relation $u = v$ from R must be applied to every single row $e \in \mathbf{N}$ (Trace$(e, u = v)$ as on line 15), while the relations from W only need to be applied to the identity row (Trace$(1, u = v)$ as on line 8).

If, instead of a right congruence, we are considering a left congruence, we may apply the same method but reversing all multiplications. This is shown in pseudo-code as ToddCoxeterLeft in Algorithm 2.29. The row given by $(1, w_1 w_2 \dots w_k)\bar{\tau}$ does not correspond to the word $w_1 w_2 \dots w_k$ but to the word $w_k w_{k-1} \dots w_1$. The words in every relation in R and W should be reversed (lines 7 and 12) to make new sets of relations $\bar{R}$ and $\bar{W}$, and then we should study the right congruence defined by the resulting relations – that is, the result of ToddCoxeterRight($\left\langle X \,\middle|\, \bar{R} \right\rangle, \bar{W}$). The Todd–Coxeter method then works in exactly the same way as described above, but it should be remembered on completion that the resulting table describes the semigroup and congruence with left multiplication instead of right multiplication.

Algorithm 2.29 The ToddCoxeterLeft algorithm (for left congruences)

```
1: procedure ToddCoxeterLeft($\langle X \mid R \rangle$, $W$)
2:     $\bar{R} := \varnothing$
3:     $\bar{W} := \varnothing$
4:     for $(u, v) \in R$ do
5:         Let $u = u_1 u_2 \dots u_m$                         ▷ where each $u_i$ is in $X$
6:         Let $v = v_1 v_2 \dots v_n$                         ▷ where each $v_j$ is in $X$
7:         $\bar{R} \leftarrow \bar{R} \cup \{(u_m \dots u_1, v_n \dots v_1)\}$
8:         ▷ $(\bar{u}, \bar{v}) \in \bar{R}$ for every $(u, v)$ in $R$ processed so far
9:     for $(u, v) \in W$ do
10:        Let $u = u_1 u_2 \dots u_m$                         ▷ where each $u_i$ is in $X$
11:        Let $v = v_1 v_2 \dots v_n$                         ▷ where each $v_j$ is in $X$
12:        $\bar{W} \leftarrow \bar{W} \cup \{(u_m \dots u_1, v_n \dots v_1)\}$
13:        ▷ $(\bar{u}, \bar{v}) \in \bar{W}$ for every $(u, v)$ in $W$ processed so far
14:    return ToddCoxeterRight($\left\langle X \,\middle|\, \bar{R} \right\rangle, \bar{W}$)
```

Theorem 2.30. *Let $\langle X \mid R \rangle$ and W be as in Section 2.3 and ρ be the left congruence they specify.* ToddCoxeterLeft($\langle X \mid R \rangle$, W) *returns a table which describes the classes of ρ (after the identity row 1 is removed, and with respect to left multiplication).*

Proof. For a word $w \in X^*$, let $\bar{w}$ be the reverse of that word, i.e. if $w = w_1 w_2 \dots w_n$ where $w_i \in X$ for all $i \in \{1 \dots n\}$, then let $\bar{w} = w_n w_{n-1} \dots w_1$. Let $\bar{R}$ and $\bar{W}$ be produced from R and W as shown in Algorithm 2.29, by reversing all the words in all the pairs in each of the two sets. Note that for two words $u, v \in X^*$, $\overline{uv} = \bar{v}\bar{u}$, and that $\bar{\bar{u}} = u$.

Let S be defined by the presentation $\langle X \mid R \rangle$, and let $\bar{S}$ be defined by the presentation $\left\langle X \,\middle|\, \bar{R} \right\rangle$. If u and v are two words in X^+, then $(u, v) \in R$ if and only if $(\bar{u}, \bar{v}) \in \bar{R}$. This gives rise to a natural anti-isomorphism $\bar{\cdot}$ from S to $\bar{S}$: if $w \in X^+$ represents some $s \in S$, then $\bar{s}$ is the element of $\bar{S}$ represented by $\bar{w}$. This anti-isomorphism is the basis for ToddCoxeterLeft.

Since ρ is a left congruence, we have the rule that $(sa, sb) \in \rho$ for every element $s \in S$ and every pair $(a, b) \in \rho$. We define

$$\bar{\rho} = \{(\bar{a}, \bar{b}) \mid (a, b) \in \rho\},$$

a relation on $\bar{S}$. The reflexivity, symmetry and transitivity of $\bar{\rho}$ follow from ρ. But note that $(\overline{sa}, \overline{sb}) = (\bar{a}\bar{s}, \bar{b}\bar{s}) \in \bar{\rho}$ for every $(\bar{a}, \bar{b}) \in \bar{\rho}$, and so $\bar{\rho}$ is a right congruence. In fact, it is the right congruence generated by $\bar{W}$.

Now when we call ToddCoxeterRight($\left\langle X \,\middle|\, \bar{R} \right\rangle, \bar{W}$) we know we are producing a table which represents $\bar{\rho}$. We know that S is anti-isomorphic to $\bar{S}$, and that $(a, b) \in \rho$ if and only $(\bar{a}, \bar{b}) \in \bar{\rho}$, so the table can be used directly to describe ρ itself. We only need to take care to remember that all the multiplications shown by the Todd–Coxeter table $(n, \mathbf{N}, \tau)$ are actually left-multiplications. □

Computing program outputs

So far we have described how the Todd–Coxeter procedure operates and the output it produces. However, on completion of ToddCoxeterLeft or Right or TwoSided, it may not be immediately obvious how information about the congruence can be retrieved from the resultant table. In this section we will explain how the output of these algorithms can be used to produce the four program outputs (i) to (iv) required by Section 2.4. Assume that we have taken inputs X, R, W and applied them to one of the three algorithms; let ρ be the left, right, or two-sided congruence described by these inputs, over the semigroup S.

Firstly, we require (i), an algorithm to determine whether a given pair (x, y) of elements in the semigroup lies in ρ. If x and y are elements of S, then we can find words w_x and w_y over the alphabet X which represent them (see Section 2.3). We can then simply compute Trace$(1, w_x)$ and Trace$(1, w_y)$, which will be equal to each other if and only if $(x, y) \in \rho$.

For (ii), the number of congruence classes of ρ, we simply observe that each congruence class is represented by one unique row in the Todd–Coxeter table. Each row corresponds to a single congruence class, except row 1 which acts as an appended identity. Hence the number of classes is equal to one less than the number of rows in the table.

For (iii) we require an algorithm that takes an element x and returns the index of the congruence class to which it belongs. As described above for (i), we can simply take a word w_x that represents x, and call Trace$(1, w_x)$: the result is the index of the congruence class.

Finally, output (iv) is a list of the elements in each non-trivial congruence class. This is not so immediately available, but may still be computed if S is finite. Firstly, we require a list of the elements of S: if we have a concrete representation, then this is given; if we have only a finite presentation $\langle X \mid R \rangle$ then ToddCoxeter can be called on the presentation, and a representative word for each element can be found during this run. Once this list of words is found, we can use Trace on each one, as for (iii), to find which congruence class its element lies in. These results can then be compiled, and singletons discarded, to produce the desired output.

Note that output (iv) cannot be produced using the Todd–Coxeter algorithm when S is infinite. However, the pair orbit algorithm will succeed for infinite S, so long as there are only finitely many elements in non-trivial classes. See Section 2.6.1 for more details.

Pre-filling the table

In the ordinary Todd–Coxeter procedure as described above, we begin with just a single row representing the empty word, and we add rows to represent the different congruence classes as we go along, merging rows together if they are found to describe the same congruence class. However, in the case that we are calculating a congruence over a concretely represented semigroup – or indeed, over a finitely presented semigroup on which the Todd–Coxeter algorithm has already been run – we have certain information which we can use to help us with our calculation. We now describe how to use this information in a process called *pre-filling*, which represents new work in this area.

If we have a semigroup S, then we may be able to find a right Cayley graph Γ for S with respect to its generators X: if we have a concrete representation, then we can find this graph using the Froidure–Pin algorithm (see Section 2.5); or if the Todd–Coxeter algorithm has been run on S, then the resulting table contains all the information required for a right Cayley graph. Either way, this graph tells us, for each element $s \in S$ and each generator $x \in X$, the value of $s \cdot x$. To **pre-fill** the table is to convert this Cayley graph into a Todd–Coxeter table $(n, \mathbf{N}, \tau)$ with rows that represent the elements of $S^{(1)}$, instead of starting with just a single row representing the identity. Then we only need to process the pairs in W in order to calculate the equivalence classes of ρ, and we do not need to use the pairs in R at all. We now describe how this works in more detail, with pseudo-code in Algorithm 2.31.

We initialise the table with $|S| + 1$ rows (line 2), all of them active (line 3), where row 1 represents the identity as usual, and rows 2 to $|S| + 1$ represent the elements of S. We put these elements in some order such that $S = \{s_1, s_2, \dots, s_k\}$. We now fill in the table, not with empty cells as in previous Todd–Coxeter algorithms, but with the values we know they should have according to the Cayley graph of S.

First, row 1 represents the identity. When we multiply the identity by a generator x, we get x itself. Hence, in lines 5–6, for each $x \in X$, we fill in $(1, x)\tau$ with the row that corresponds to x (that is, one more than the position of x in S). Now, each of the rows beyond row 1 corresponds to an element in S: row i corresponds to the element s_{i-1}. For each row i and each generator x, we therefore need to fill the table in with the row number of the element $s_{i-1} \cdot x$. Hence, in lines 7–8, we fill in $(i, x)\tau$ with the row that corresponds to $s_{i-1} \cdot x$ (that is, one more than the position of $s_{i-1} \cdot x$ in S).

Recall the notation we defined in Section 2.3: our semigroup S is presented by $\langle X \mid R \rangle$, and we wish to calculate a congruence ρ defined by the generating pairs W. **Pre-filling** the table with the right Cayley graph as described in lines 1–8 above is equivalent to processing all the relations R and running the Todd–Coxeter algorithm to completion: we have a table which describes the elements of S, or in other words, the classes of the trivial congruence Δ_S on S. In fact, the state of $(n, \mathbf{N}, \tau)$ on line 9 is the same as the output of $\text{ToddCoxeter}(\langle X \mid R \rangle)$, but may have taken much less time to compute.

Since we wish to calculate information about the classes of ρ, we now have to consider the pairs W, and find out which Δ_S-classes should be combined to make ρ-classes. Since there are no blanks in the table (i.e. we never have $(i, x)\tau = 0$), we will never be forced to use the Add operation and create new rows; the procedure from now on will simply be about tracing relations from W and combining rows together. As we can see, the remainder of the algorithm (lines 10–18) are the same as the final lines of ToddCoxeterTwoSided, with the one difference that the two lines which trace relations from R have been removed – these relations are now unnecessary, since they are already incorporated into the table by prefilling from the Cayley graph of S. This is the main source of time-saving that occurs in this algorithm, and is the key to its speed advantage over the non-prefilled version (see Figures 2.54, 2.55 and 2.56 for performance benchmarks).

Theorem 2.32. *Let $\langle X \mid R \rangle$ and W be as in Section 2.3, let S be a semigroup presented by $\langle X \mid R \rangle$, and assume we have a concrete representation for S.* ToddCoxeterPrefill*($\langle X \mid R \rangle$, W, S) has output identical to* ToddCoxeterTwoSided*($\langle X \mid R \rangle$, W), up to relabelling of the rows.*

Proof. The only difference between Prefill and TwoSided is that Prefill initialises the Todd–Coxeter table with a right Cayley graph for S, while TwoSided computes one from

Algorithm 2.31 The ToddCoxeterPrefill algorithm

```
Require: $S = \{s_1, s_2, \ldots, s_k\}$ is a semigroup with its elements known in a defined order
 1: procedure ToddCoxeterPrefill($\langle X \mid R \rangle, W, S$)
 2:     $n := |S| + 1$
 3:     $\mathbf{N} := \{1, \ldots, n\}$
 4:     $\tau : \mathbf{N} \times X \to \mathbf{N} \cup \{0\}$
 5:     for $x \in X$ do
 6:         $(1, x)\tau :=$ (position of $x$ in $S$) + 1
 7:         for $i \in \{2, \ldots, n\}$ do
 8:             $(i, x)\tau :=$ (position of $s_{i-1} \cdot x$ in $S$) + 1
 9:     ▷ At this point $(i, u)\bar{\tau} = (i, v)\bar{\tau}$ for all $(u, v) \in R$ and $i \in \mathbf{N}$
10:     $e := 0$
11:     repeat
12:         $e \leftarrow e + 1$
13:         if $e \in \mathbf{N}$ then
14:             for $(u, v) \in W$ do
15:                 Trace($e, u = v$)
16:                 ▷ $(e, u)\bar{\tau} = (e, v)\bar{\tau}$
17:         ▷ $(i, u)\bar{\tau} = (i, v)\bar{\tau}$ for all $(u, v) \in W$ and $i \in \mathbf{N}$ such that $i \leq e$
18:     until $e = n$                                   ▷ There are no rows left to process
19:     return $(n, \mathbf{N}, \tau)$
```

scratch using R.

In lines 14 and 15 of TwoSided (Algorithm 2.26), we are doing the equivalent of the original ToddCoxeter algorithm, which produces a right Cayley graph for S. Instead, in lines 2–8 of Prefill (Algorithm 2.31) we input a right Cayley graph for S directly, without consulting R. These two operations produce the same result – a Todd–Coxeter table filled with a right Cayley graph for S – so it does not matter which method we use to do so.

The only other part of the algorithm is to process the pairs in W, something which both algorithms do identically. It might be noticed that in TwoSided this is done before the pairs in R. Again, since the operation is equivalent to processing the presentation $\langle X \mid R, W \rangle$, it does not matter in which order relations are processed. □

It is worth noting that, unlike the other Todd–Coxeter algorithms described in this thesis, ToddCoxeterPrefill is guaranteed to halt in finite time. Since X and S are finite, the loops in lines 1–8 of the algorithm are guaranteed to have a finite number of iterations. In order to see that the repeat-loop in lines 11–18 will also have a finite number of iterations, observe that there are no blanks in the table – that is, that there are no values of (i, x) such that $(i, x)\tau = 0$. Since there are no blanks, Trace can never cause a call to Add, and therefore the number of rows n will never go above $|S| + 1$. Hence the repeat-loop will go through a maximum of $|S| + 1$ iterations, and since the for-loop inside iterates over a finite set W, we are guaranteed completion in a finite number of steps.

Implementation

In the methods described above, rows may be added to the table, and deleted from it. A list must be kept of rows which are in use; when a row is added, its position in the table should be appended to this list at the end, and when a row is deleted it should be removed from its position in the list and added to a list of "free rows" which can be reused later. The "rows in use" list is best implemented as a doubly-linked list, so that single entries can be added and removed with as little processor work as possible.

2.6.3 Rewriting systems

Another approach for solving the word problem in a finite presentation is using rewriting systems. Hence, given a semigroup S with presentation $\langle X \mid R \rangle$ and a congruence ρ over S with generating pairs given by W (see Section 2.3) we may be able to find a rewriting system which converts any word $w \in X^+$ to a canonical word representing the same element of $\langle X \mid R,\ W \rangle$; that is, a word representing a semigroup element in the same ρ-class as the semigroup element of S represented by w.

For ease of notation and understanding, this section will describe an algorithm for the word problem on a finitely presented semigroup. We understand that this is the same as the problem of whether a given pair of words represent semigroup elements related to each other by a two-sided congruence ρ. The current implementation of this method in libsemigroups does not include left and right congruences as an option (but see Section 2.9.4).

In order to describe the process, we must first explain some background theory. A full description of these ideas can be found in [HEO05, Section 12.2]. Note that we shall again consider monoid presentations instead of semigroup presentations, since it is easy to change between the two by appending or removing an identity (the empty string ε).

Definition 2.33. Let X be an alphabet. A **rewriting system R** on X^* is a set of ordered pairs (u, v) where $u, v \in X^*$.

A pair $(u, v) \in \mathbf{R}$ is called a *rule*, and can be viewed as an operation which transforms an occurrence of u in a word into an occurrence of v. For this section, we will assume that $\mathbf{R}$ is finite. A rewriting system $\mathbf{R}$ extends to relations $\to_{\mathbf{R}}$, $\overset{*}{\to}_{\mathbf{R}}$, and $\overset{*}{\leftrightarrow}_{\mathbf{R}}$ which describe how words are rewritten, and which are defined as follows.

Let $u, v \in X^*$ and let $\mathbf{R}$ be a rewriting system. We write $u \to_{\mathbf{R}} v$ if there exist $(w_1, w_2) \in \mathbf{R}$ and $s, t \in X^*$ such that $u = sw_1t$ and $v = sw_2t$. That is, $u \to_{\mathbf{R}} v$ if a rule rewrites a contiguous subword of u to turn u into v. The relation $\overset{*}{\to}_{\mathbf{R}}$ is simply the reflexive transitive closure of $\to_{\mathbf{R}}$; that is, $u \overset{*}{\to}_{\mathbf{R}} v$ if and only if $u = v$ or

$$u = u_0 \to_{\mathbf{R}} u_1 \to_{\mathbf{R}} \cdots \to_{\mathbf{R}} u_n = v,$$

for some $u_0, \ldots, u_n \in X^*$. Finally, $\overset{*}{\leftrightarrow}_{\mathbf{R}}$ is the symmetric closure of $\overset{*}{\to}_{\mathbf{R}}$. It is easy to see that $\overset{*}{\leftrightarrow}_{\mathbf{R}}$ is an equivalence relation whose classes we may write as $[w]_{\mathbf{R}}$. Where there is no chance of ambiguity, we omit the subscript in these operations, just writing $\to$, $\overset{*}{\to}$ and $\overset{*}{\leftrightarrow}$.

This definition of a rewriting system does not guarantee that a word can be rewritten in a useful way. A rewriting system could allow an endless loop of rewriting; for example, a system over the alphabet $\{a, b\}$ could contain rules (aa, b) and (b, aa) which would allow the rewrite sequence

$$aa \to b \to aa \to b \to aa \to b \to aa \to \cdots$$

to go on forever. It could also be possible to rewrite one word in two different ways; for example, a system over the alphabet $\{a, b, c\}$ could contain rules (aa, b) and (aa, c).

In order to solve the word problem for a semigroup, we require a rewriting system with certain properties. We will describe these properties, and then explain how to produce a rewriting system which satisfies them.

Definition 2.34. A string $u \in X^*$ is **R-irreducible** if there is no string $v \in X^*$ such that $u \to v$; that is, u cannot be rewritten by any rule in $\mathbf{R}$. [HEO05, Def 12.13]

Definition 2.35. A rewriting system is **terminating** if there is no infinite chain of words $u_1, u_2, \ldots \in X^*$ such that $u_i \to u_{i+1}$ for all $i > 0$.

If a rewriting system is terminating, this is good news computationally. It means that any word can be transformed by rules only a finite number of times before it reaches an irreducible state, so the task of finding an irreducible form of a word is guaranteed to be achievable in finite time. But note that we can still only talk about *an* irreducible word, not *the* irreducible word. We could still have a word $u \in X^*$ and irreducible words $v, w \in X^*$ such that $u \xrightarrow{*} v$ and $u \xrightarrow{*} w$ but $v \neq w$. To avoid this, we must ensure that the system is *confluent*, as follows.

Definition 2.36. A rewriting system is **confluent** if, for any words $u, v_1, v_2 \in X^*$ such that $u \xrightarrow{*} v_1$ and $u \xrightarrow{*} v_2$, there exists a word $w \in X^*$ such that $v_1 \xrightarrow{*} w$ and $v_2 \xrightarrow{*} w$.

The intuition behind this definition is that, as the name suggests, different paths "flow together". The result is that, in a confluent terminating rewriting system, rules can be applied to a word in any order, and a canonical irreducible word will be found in a finite number of steps.

Another definition will help us to determine whether a rewriting system is confluent: *local confluence*. This is a weaker condition than confluence, but the two are strongly linked by Lemma 2.38.

Definition 2.37. A rewriting system is **locally confluent** if, for any words $u, v_1, v_2 \in X^*$ such that $u \to v_1$ and $u \to v_2$, there exists a word $w \in X^*$ such that $v_1 \xrightarrow{*} w$ and $v_2 \xrightarrow{*} w$.

Lemma 2.38 (Newman's diamond lemma [HEO05, Lemma 12.15])**.** *A terminating rewriting system is confluent if and only if it is locally confluent.*

Lemma 2.38 gives us an idea of how to check computationally whether a system is confluent: rather than checking every possible transitive rewriting of a word (its neighbours under $\xrightarrow{*}$), it suffices to check a word's immediate children (its neighbours under $\to$). This lemma will help us later, with Theorem 2.41 and the Knuth–Bendix procedure.

We can now see an application of rewriting systems to the word problem in a finitely presented monoid. Indeed, given an alphabet X and a rewriting system $\mathbf{R}$, the quotient monoid $X^*/ \overset{*}{\leftrightarrow}_{\mathbf{R}}$ is described by the monoid presentation $\langle\, X \mid \mathbf{R} \,\rangle$. Hence, given a monoid M with a presentation $\langle\, X \mid R \,\rangle$, if there is a confluent terminating rewriting system $\mathbf{R}$ such that the word equality relation $=_M$ is the same as the relation $\overset{*}{\leftrightarrow}_{\mathbf{R}}$, then the word problem can be solved simply by rewriting two words using $\mathbf{R}$ until their irreducible representatives are found, and then comparing them. The only difficulty is in finding a rewriting system which is confluent and terminating – but we can find one, by starting with the set of relations R, and then using the Knuth–Bendix completion algorithm.

Let M be a monoid with finite presentation $\langle\, X \mid R \,\rangle$. We start with no rules in our rewriting system, $\mathbf{R} = \varnothing$, and we begin to add relations from R. However, in order to ensure our system is *terminating*, we must reorder each relation to ensure we do not create any loops. For this purpose, we define a total ordering $\leq$ on X^* and reorder each relation so that a rule (u, v) has the property that $u > v$. Our chosen ordering $\leq$ must be a *reduction ordering*, meaning that if we have $u \leq v$ then we must also have $wux \leq wvx$ for all $w, x \in X^*$ [HEO05, §12.2]. For our purposes, it will suffice to use the *shortlex ordering*: $u < v$ if and only if u is shorter than v or they have equal length and u is less than v lexicographically. Hence the first few words over the alphabet $\{a, b\}$ are

$$\varepsilon < a < b < aa < ab < ba < bb < aaa < aab < aba < \cdots$$

Note that this requires a well-understood total order on the alphabet X itself. Another possible reduction ordering is *wreath product ordering* [Sim94, §2.1], which has a particular application to polycyclic groups; however, this has no particular advantage when applied to semigroups, so the simpler shortlex ordering is preferable.

The use of a reduction ordering justifies the use of the words “reducible” and “irreducible” – words are always replaced with lesser words. We must ensure, at all points during the algorithm, that every rule $(u, v) \in \mathbf{R}$ satisfies $u > v$.

Example 2.39 (Exercise 3.1 in [Sim94, §2.3])**.** Let $X = \{a, b\}$, and let $\mathbf{R}$ be a rewriting system on X^* given by

$$\mathbf{R} = \Big\{ \big(a^5, \varepsilon\big),\ \big(b^5, \varepsilon\big),\ \big(b^4a^4, (ab)^4\big),\ \big((ba)^4, a^4b^4\big) \Big\}.$$

We can apply any sequence of these rules to any word we wish. For example, we can rewrite the word b^5a^6 using the last two rules as follows:

$$b^5a^6 = b\underline{bbbbaaaa}aa \rightarrow_{\mathbf{R}} ba\underline{babababa}a \rightarrow_{\mathbf{R}} baaaaabbbba.$$

Hence $b^5a^6 \overset{*}{\rightarrow}_{\mathbf{R}} ba^5b^4a$. Alternatively we could rewrite it using the first two rules:

$$b^5a^6 \rightarrow_{\mathbf{R}} \varepsilon a^6 = a^5a \rightarrow_{\mathbf{R}} \varepsilon a = a.$$

Hence $b^5a^6 \overset{*}{\rightarrow}_{\mathbf{R}} a$. Using both these results together with transitivity, we have $a \overset{*}{\leftrightarrow}_{\mathbf{R}} ba^5b^4a$.

Let us consider the shortlex ordering described above. The first two rules in $\mathbf{R}$ shorten a word by 5 characters, and the last two rules replace a subword of length 8 beginning with b by one beginning with a. Hence all four rules (u, v) have $u > v$ with respect to shortlex ordering. Hence $\mathbf{R}$ is a terminating rewriting system.

Once each relation from R is added – possibly reordered – to $\mathbf{R}$, we will have a terminating rewriting system such that $X^*/\overset{*}{\leftrightarrow}_{\mathbf{R}} = \langle X \mid R \rangle$, as required. It only remains to add rules to $\mathbf{R}$ to make it confluent, without altering the relation $\overset{*}{\leftrightarrow}$. This is where we use the Knuth–Bendix completion process.

First described by Knuth and Bendix in [KB83], the completion process adds rules to $\mathbf{R}$ by finding and resolving *critical pairs*. A critical pair is a pair of rules (u_1, v_1) and (u_2, v_2) from $\mathbf{R}$ such that u_1 and u_2 overlap in a certain way. They are defined as follows, as in [HEO05, Lemma 12.17].

Definition 2.40. Let $\mathbf{R}$ be a terminating rewriting system over an alphabet X^*. A **critical pair** is a pair of rules (u_1, v_1) and (u_2, v_2) in $\mathbf{R}$ such that one of the following is true:

(i) $u_1 = rs$ and $u_2 = st$ with $r, s, t \in X^*$ and $s \neq \varepsilon$;

(ii) $u_1 = ru_2t$ for some $r, t \in X^*$ and $u_2 \neq \varepsilon$.

A critical pair results in a choice when applying rewriting rules, which leads to different results depending on which rule is applied. A critical pair of type (i) allows us to rewrite rst to either v_1t or rv_2, depending on which rule is applied; and a critical pair of type (ii) allows us to rewrite u_1 to either v_1 or rv_2t, depending on which rule is applied. In order for $\mathbf{R}$ to be confluent, we have to ensure that whichever option is chosen, a word is still rewritten to a fixed word later on. This notion is made precise in the following theorem, which is key to the Knuth–Bendix completion process.

Theorem 2.41 ([HEO05, Lemma 12.17])**.** *A terminating rewriting system $\mathbf{R}$ over X is confluent if and only if, for each critical pair of rules (u_1, v_1) and (u_2, v_2), the following hold:*

(i) *if it is a critical pair of type (i), there exists some word $w \in X^*$ such that $v_1t \xrightarrow{*}_{\mathbf{R}} w$ and $rv_2 \xrightarrow{*}_{\mathbf{R}} w$;*

(ii) *if it is a critical pair of type (ii), there exists some word $w \in X^*$ such that $v_1 \xrightarrow{*}_{\mathbf{R}} w$ and $rv_2t \xrightarrow{*}_{\mathbf{R}} w$.*

Now we have all the concepts required to describe the Knuth–Bendix completion process. The process searches through rules in $\mathbf{R}$ looking for critical pairs; when a critical pair is found which does not satisfy the condition stated in Theorem 2.41, an appropriate word w is chosen and the rules required in Theorem 2.41 are added to $\mathbf{R}$ in order to ensure confluence.

Though, as mentioned in Section 1.12, the word problem for a semigroup presentation is undecidable in general, we do have a result about the Knuth–Bendix process which ensures completion in some cases.

Theorem 2.42. *Let S be a semigroup with a finite presentation $\langle X \mid R \rangle$, and let $\mathbf{R}$ be the rewriting system produced from R by applying a shortlex ordering to all pairs, as described above.*

If S is finite, then the Knuth–Bendix process applied to $\mathbf{R}$ will eventually halt with a finite, confluent and terminating set of rules.

Proof. Let $u, v \in X^*$. We know that $u \overset{*}{\leftrightarrow}_{\mathbf{R}} v$ if and only if u and v describe the same element of S. In particular, if S is finite, then the $\overset{*}{\leftrightarrow}_{\mathbf{R}}$ relation will have finitely many classes – one for each element of S. Finally, [HEO05, Corollary 12.21] tells us that if the Knuth–Bendix algorithm is run on any rewriting system R such that $\overset{*}{\leftrightarrow}_R$ has only finitely many classes, it will halt with a finite, confluent and terminating set of rules. This applies to our rewriting system $\mathbf{R}$, and so we have the result we need. $\square$

Let us consider an example of each of the two types of critical pair in Definition 2.40.

Example 2.43. Let $X = \{a, b, c\}$, and let $\mathbf{R}$ be a rewriting system on X, containing two rules (ab, c) and (bb, a). Here the word abb could be rewritten by either rule: $abb = (ab)b \to cb$, but also $abb = a(bb) \to aa$. Hence (cb, aa) is a critical pair of type (i).

If there exists some $w \in X^*$ such that $cb \xrightarrow{*} w$ and $aa \xrightarrow{*} w$, then confluence is not violated; otherwise, we must add a new rule to $\mathbf{R}$ to make sure confluence holds. The Knuth–Bendix process adds the rule (cb, aa), since $aa < cb$ in our shortlex order. Now Theorem 2.41 is satisfied.

Example 2.44. Let $X = \{a, b, c\}$ and let $\mathbf{R}$ be a rewriting system on X with rules (abc, c) and (b, a). The word abc can now be written by either rule, $abc \to c$ or $abc \to aac$. Hence (c, aac) is a critical pair of type (ii).

Again, if there are other rules allowing both words to be rewritten to a word $w \in X^*$, then no rules need to be added; if however there are no such rules, we must add (aac, c) to ensure the theorem is satisfied, and confluence can be guaranteed.

The Knuth–Bendix completion process consists of searching for these critical pairs and adding rules where necessary. For a more detailed description of precisely how these tasks are done, see [Sim94, §2.6].

A large part of the computational work involved in the Knuth–Bendix process consists of rewriting words using rules from $\mathbf{R}$. If the rewriting system is confluent and terminating, then the rules can be applied in any order and an irreducible word will eventually be reached. However, the order in which the rules are applied can have a great effect on the overall runtime of the algorithm. In Algorithm 2.45 we present one possible strategy, known as *rewriting from the left*, which is based on the REWRITE_FROM_LEFT procedure described in [Sim94, §2.4], where the issues of termination and correctness are addressed formally. The algorithm takes two arguments – a word u and a confluent terminating rewriting system $\mathbf{R}$ – and returns the irreducible word produced from u by applying rules in $\mathbf{R}$.

The REWRITE algorithm starts with the original word u, and an empty word v. We enter a loop that constitutes the rest of the algorithm. On each iteration, we start by removing the first character from u and putting it onto the end of v (lines 4–6). Next we go through each rule $p \to q$ in $\mathbf{R}$ (line 7) and attempt to apply it to a suffix of v – that is, we check whether v ends in p (line 8). If we find a rule $p \to q$ such that v does end in p, then we remove the occurrence of p from the end of v (line 9) and we add the rewritten version q to the beginning of u (line 10) for further processing in a later iteration of the repeat-loop. After finding just one of these rules to apply, we stop going through rules (line 11) and if appropriate, go back to the beginning of the while-loop (line 4). When the whole word has been processed and u is therefore empty (which will happen in finite time so long as $\mathbf{R}$ is terminating and confluent) we return v, which is the irreducible result of u after a series of rules from $\mathbf{R}$ have been applied.

Algorithm 2.45 The REWRITE algorithm

```
 1: procedure REWRITE($u, \mathbf{R}$)
 2:     $v := \varepsilon$
 3:     while $u \neq \varepsilon$ do
 4:         Let $u_1$ be the first character of $u$
 5:         $v \leftarrow vu_1$
 6:         Remove the first character of $u$
 7:         for $(p, q) \in \mathbf{R}$ do
 8:             if $v = rp$ for some $r \in X^*$ then
 9:                 $v \leftarrow r$
10:                 $u \leftarrow qu$
11:                 break
12:             ▷ $vu$ has been rewritten using a rule of $\mathbf{R}$
13:         ▷ $vu$ is $\overset{*}{\leftrightarrow}_{\mathbf{R}}$-related to the original $u$
14:     return $v$
```

It should be noted that, like the Todd–Coxeter procedure, there are many inputs for which the Knuth–Bendix algorithm will not terminate. More rules may be added continually, which themselves need to be checked for critical pairs, without a complete set ever being found. It may also be that a confluent terminating rewriting system will eventually be found, but not for a very long time – and it may not be known in advance whether the process is going to terminate. However, this process has one clear advantage over the Todd–Coxeter algorithm, which is that it might complete even when the monoid $\langle X \mid R \rangle$ is infinite, so long as the set of rules in the rewriting system is finite (see Example 2.46). This is a strong argument in favour of trying the Knuth–Bendix process along with other methods. The Knuth–Bendix process as currently implemented cannot be used for left or right congruences, an area where the Todd–Coxeter algorithm has a clear advantage (but again, see Section 2.9.4).

We give an example of a rewriting system which solves the word problem for an infinite monoid:

Example 2.46. The *bicyclic monoid* $B = \langle b, c \mid bc = \varepsilon \rangle$ trivially admits a rewriting system $\{(bc, \varepsilon)\}$. Using this, any element can be rewritten in a finite number of steps to an irreducible word of the form $c^i b^j$. There is only one rule in the rewriting system. Since it reduces words by the *short-lex* ordering, the system is terminating; and since there are no critical pairs, the system is confluent by Theorem 2.41.

Computing program outputs

We have shown how the inputs X, R and W can be used as inputs to the Knuth–Bendix algorithm in order to produce a rewriting system. However, we have not explicitly stated how this can be used to produce the information we require about the congruence in question, in the sense of the four program outputs in Section 2.4. We will now consider each of these outputs in turn, and explain how the rewriting system $\mathbf{R}$ can be used to produce them. Let X, R and W be as described earlier, let S be the semigroup in question, and let ρ be the congruence we are computing.

For (i) we require an algorithm to determine whether a given pair (x, y) is in ρ. This is the simplest of the four: we take two words w_x and w_y over the alphabet X, which represent x and y respectively, and we rewrite them both using $\mathbf{R}$. The pair (x, y) is in the congruence if and only if the two words rewrite to the same word.

For (ii) we require the number of classes the congruence has. As just mentioned, an element's class is defined by the word produced by rewriting it with $\mathbf{R}$; hence, the number of classes is equal to the number of different words to which words can be rewritten. If S is finite, this number can be found by taking a word that represents each element of S, rewriting it with $\mathbf{R}$, and calculating the number of distinct words computed. However, if S is infinite, we do not have such a method available, and we cannot return a number in this case. Note that there could still be a finite number of classes, in which case the Todd–Coxeter algorithm could return an answer (see "Computing program outputs" in Section 2.6.2).

For (iii) we require an algorithm ClassNo that takes an element x and returns the index of the congruence class to which it belongs. That is, it should return a positive integer for any element, and it should have the same output for elements x and y if and only if $(x, y) \in \rho$. In libsemigroups we take an approach which creates indices for classes as calls are made to the algorithm. We start with an empty list L. When ClassNo(x) is called, the algorithm takes a word representing x, and rewrites it to some word w using the rewriting system $\mathbf{R}$. If w is in L, then the algorithm returns its position in the list; if not, then the algorithm adds it to the end of L, and returns its new position. This gives ClassNo the required behaviour, but it is important to note that the outputs depend on the order in which past calls were made. In two different program sessions, in which calls were made in a different order, different numbers would be returned. In this sense, ClassNo is not canonical.

Finally, for (iv), we wish for a list of the elements in each non-trivial congruence class. If S is finite and we have a list of its elements, we can use (iii) and apply ClassNo to each element, to find out which elements lie in which class. However, if S is infinite, no such method is available, and we would have to rely on the pair orbit algorithm to produce the answer.

Note that outputs (ii) and (iv) require a list of the elements of S, which might not be immediately available at the end of the Knuth–Bendix algorithm. If we have a concrete representation for S, then we do have a list of all the elements of S. However, if we only have a finite presentation for S, then we would need to run another algorithm such as the Todd–Coxeter algorithm on $\langle X \mid R \rangle$ in order to find a list of its elements.

2.7 Running in parallel

Now that we have described the various individual algorithms for computing with congruences, we can describe the overall parallel method which ties all these algorithms together.

The basic principle is not complicated: run all the described algorithms simultaneously, and whenever one of the algorithms finds an answer, kill all the other algorithms and return the answer that was given. However, the precise details depend on whether we have a concrete representation or a finite presentation, as well as whether we are considering a left, right or two-sided congruence. Table 2.47 shows which algorithms are run in which cases. A tick (✓) denotes that the algorithm is used, a cross (✘) denotes that it cannot be used, and a tilde ($\sim$) denotes that the algorithm could be applied in principle, but is so rarely the fastest option that in practice it is not worth including.

Type	Side	TC	TC (pre-fill)	P	KB
Concrete	Two-sided	✓	✓	$\sim$	$\sim$
	Left/right	✓	✓	$\sim$	✘
FP	Two-sided	✓	✘	✓	✓
	Left/right	✓	✘	✓	✘

Table 2.47: The algorithms that are used in various cases.

The positions in the table marked with a tilde are chosen based on extensive benchmarking tests with libsemigroups, which can be seen in detail in Section 2.8. It is true that in certain examples these algorithms will complete faster than the others; however, so few of these examples have been encountered that it seems more desirable to eliminate the setup cost and overheads for these algorithms by omitting them entirely, since in the vast majority of cases this will speed things up.

It should be noted that any one of these algorithms might be able to return an answer before its respective data structure has been completely evaluated. For instance, if we are running the

Todd–Coxeter algorithm in order to decide whether two words u and v are congruent, we can at any time check whether $(1,u)\bar{\tau} = (1,v)\bar{\tau}$: if this equality holds, then the two words lie in the same congruence class, and since a class is never split in two by the Todd–Coxeter algorithm, an answer of "true" can be returned immediately, without any need to run the algorithm until a complete table is found. If, on the other hand, $(1,u)\bar{\tau} \neq (1,v)\bar{\tau}$, then we cannot return an answer: it might be that the words genuinely lie in different congruence classes, or it might be that a coincidence between their rows will be found in the future. An answer of "false" can therefore not be returned until the Todd–Coxeter algorithm is run to completion.

A similar condition is true for both the Knuth–Bendix algorithm and pair orbit enumeration. Imagine we have a rewriting system $\mathbf{R}$ halfway through a run of the Knuth–Bendix algorithm; if two words u and v rewrite to the same word under $\mathbf{R}$, then we can return "true" immediately, but if they do not, then it might be that the Knuth–Bendix algorithm will add rules to $\mathbf{R}$ later which will cause u and v to rewrite to the same word. Pair orbit enumeration has this condition even more obviously: more and more pairs are found, and the pair we are looking for can turn up at any time.

In an implementation of this parallel method, it is therefore prudent to have a periodic check to see whether a desired pair is present, and return immediately if it is. This check could be done by each individual algorithm every few hundred operations, or on a timer every few milliseconds. In this way, operations can quickly return desired results, even if it would take them a very long time (or even an infinite length of time) to run to completion.

When an algorithm is terminated early after having returned an answer, the data structures involved and the work done so far could be thrown away. However, it may make sense to keep all the data structures as they are, preserving the algorithm in a suspended state. This way, if another piece of information is required later in the running of a program, and the answer is not already known, the algorithm can simply be resumed. The only alteration to the algorithm on resuming should be a periodic check for a different piece of information. This is the approach that is taken in libsemigroups.

2.8 Benchmarking

The approach described in this chapter was implemented in the C++ programming language in libsemigroups [MT+18], and this allows us to run benchmarking tests to analyse the performance of the different algorithms, compared to each other and compared to existing programs. In this section we provide some examples of such tests.

2.8.1 Examples

Firstly, it will be helpful to examine a few examples of problems which can be solved by the different algorithms, and to compare the types of problems in which each algorithm is most effective. These examples were implemented in the benchmarking section of libsemigroups, and their performance was tested with various algorithms.

Example 2.48. Consider $\mathcal{P}B_2$, the full PBR monoid of degree 2 – this is a special type of diagram monoid based on directed graphs, and is explained in detail in [EENFM15, §2.1]. It has 65536 elements and is therefore large enough to require significant processing time when considering its congruences. We take a set of pairs which we know will generate a two-sided congruence equal to the universal congruence, and we ask libsemigroups for the number of congruence classes, using the simple Todd–Coxeter algorithm in one case and the pre-filled Todd–Coxeter algorithm in the other.

On average, the simple Todd–Coxeter algorithm returns an answer in 724 milliseconds, while the pre-filled version requires an average of 3057 milliseconds, more than 4 times as long. This difference can be explained by the structure of the table at completion: a single row representing the one congruence class. In the pre-fill case, a coset table must be constructed with 65536 rows, only to have all but one row deleted. In the simple case, the number of rows may stay very small throughout the procedure, drastically reducing the amount of work which needs to be done.

Example 2.49. The virtue of the pre-filled Todd–Coxeter algorithm is shown by a different congruence on $\mathcal{P}B_2$. A much smaller set of generating pairs gives rise to a two-sided congruence ρ with 19009 congruence classes, in contrast to the universal congruence which has only one. On average, the simple Todd–Coxeter algorithm returns in around 128 seconds, while the pre-filled version completes in under 12 seconds, a tenfold increase in speed. This disparity can be explained by the large number of rows which must be in the table on completion: the simple algorithm has to compute a large amount of information from relations, and add rows one by one, while the pre-filled algorithm begins the process with much of the table's information already known, and only has to combine rows which coincide.

The two examples above show the difference in speed between two algorithms. We now consider an example where certain algorithms will never terminate, but others complete quickly. This is an even greater justification for the parallel use of a variety of algorithms.

Example 2.50. Consider the semigroup S defined by the semigroup presentation

$$\left\langle\, a, b \,\middle|\, a^3 = a,\ ab = ba \,\right\rangle .$$

We define ρ to be the two-sided congruence on S generated by the single pair (a, a^2). Since each word of the form b^i represents a different element of S, and a different class of ρ, we can conclude that S is infinite and ρ has an infinite number of congruence classes. A test asks whether (ab^{20}, a^2b^{20}) is a pair in ρ. The Todd–Coxeter methods would never complete, since they require a separate row for each congruence class and therefore an infinite amount of memory. However, the Knuth–Bendix method is able to answer the question very quickly, since it is not restricted to finite objects.

Taken together, these examples give a justification for the use of several different algorithms to solve one question. Since it may be unknown in advance which algorithms may perform well or which may return at all, it is useful to be able to execute all at once and return in whichever run-time is quickest.

2.8.2 Comparison on random examples

In order to analyse performance, a large number of example congruences were generated randomly, and tested using each different algorithm described in this chapter, along with the default implementation in GAP [GAP18]. All these tests were conducted using an Intel Core i7-4770S CPU running at 3.10GHz with 16GB of memory.

Figure 2.51 shows the results of a set of 250 tests on transformation semigroups. In each test, 4 transformations of degree 7 were chosen at random, and used to generate a semigroup S (any semigroup of size over 25000 was rejected). The elements of S were computed, and a pair of elements was chosen at random to generate a two-sided congruence ρ. Three more pairs were chosen at random, and each of the different algorithms described in this chapter was used to determine whether each pair was contained in ρ.

For greater variety, similar tests were also conducted using 3 generating pairs for ρ, and using a mixture of 1 to 10 generating pairs, as shown in 2.52 and 2.53. The time taken to return an answer was recorded in each case, and these figures were compared to one another. The algorithms used were:

- the Todd–Coxeter algorithm (`tc`),
- the Todd–Coxeter algorithm with pre-filled table (`tc_prefill`),
- the Knuth–Bendix algorithm (`kb`),
- pair orbit enumeration (`p`),
- the parallel method described in this chapter (`default`),
- and the method implemented in the library of GAP (`GAP`).

As can be seen in Figure 2.51, the pre-filled Todd–Coxeter method is the most likely to complete fastest, with the standard Todd–Coxeter algorithm winning in a sizeable minority of cases. This backs up the observations that group theorists have made that the Todd–Coxeter algorithm tends to perform faster than the Knuth–Bendix algorithm [HHKR99]. We may also observe that pair orbit enumeration sometimes completes almost instantly, which makes some sense when we consider how little work the algorithm does when there are very few non-reflexive pairs in the congruence. The Knuth–Bendix procedure lags behind badly on these examples, taking even longer than the built-in methods in GAP. These results are a justification for the decision to run only the Todd–Coxeter algorithms in the case of a concrete representation.

Figures 2.52 and 2.53 show that with a higher number of generating pairs, the pair orbit enumeration algorithm suffers badly – this can be understood, since it generally has to enumerate more pairs when the generating set is larger. However, with more generating pairs `tc` tends to perform relatively better, since there are likely to be fewer congruence classes and therefore fewer rows in the coset table.

This tendency is illustrated further in Figures 2.54 and 2.55, which show larger tests, each using 5 generators of degree 8, and of size up to 100,000. In these figures we compare the standard Todd–Coxeter algorithm with the pre-filled version, arranged according to whether the congruence in question has many or few classes relative to its size. The x axis in these figures is

$$(\text{Number of congruence classes} - 1)/(\text{Size of } S - 1),$$

a scale from 0 to 1, where 0 represents a universal congruence and 1 represents a trivial congruence. Since the size of S is a major factor in how long any algorithm takes to run, only the ratio

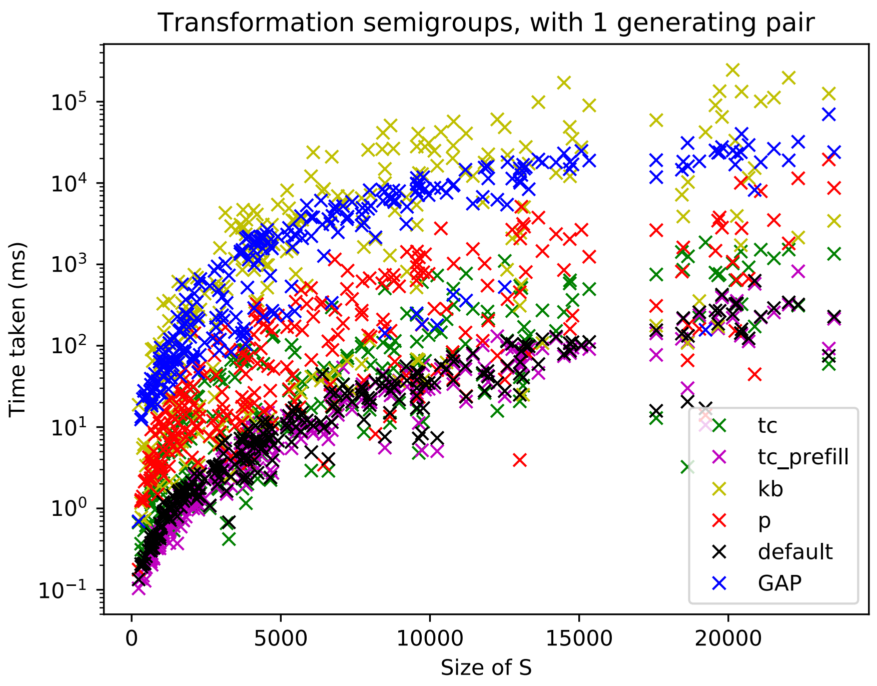


Figure 2.51: Performance of the algorithms on 250 transformation semigroups, with one generating pair.

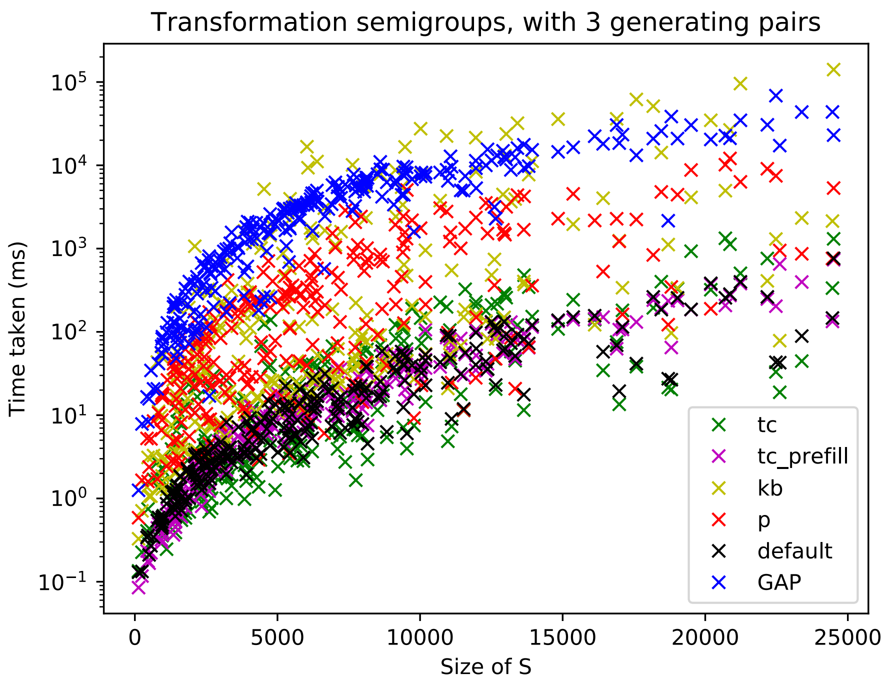


Figure 2.52: Performance of the algorithms on 250 transformation semigroups, with three generating pairs.

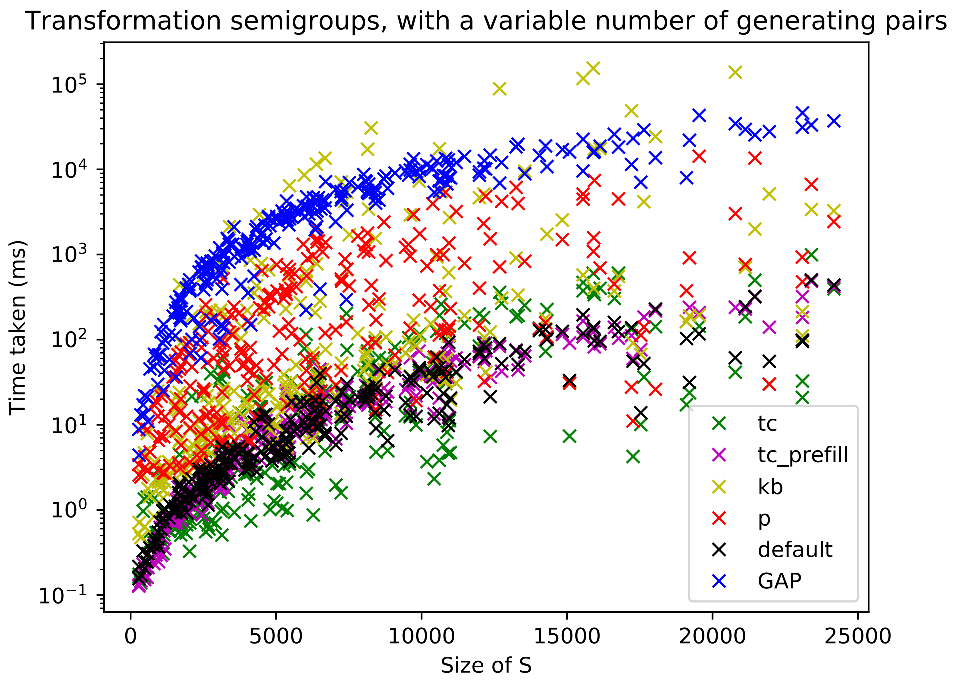


Figure 2.53: Performance of the algorithms on 250 transformation semigroups, with a variable number of generating pairs.

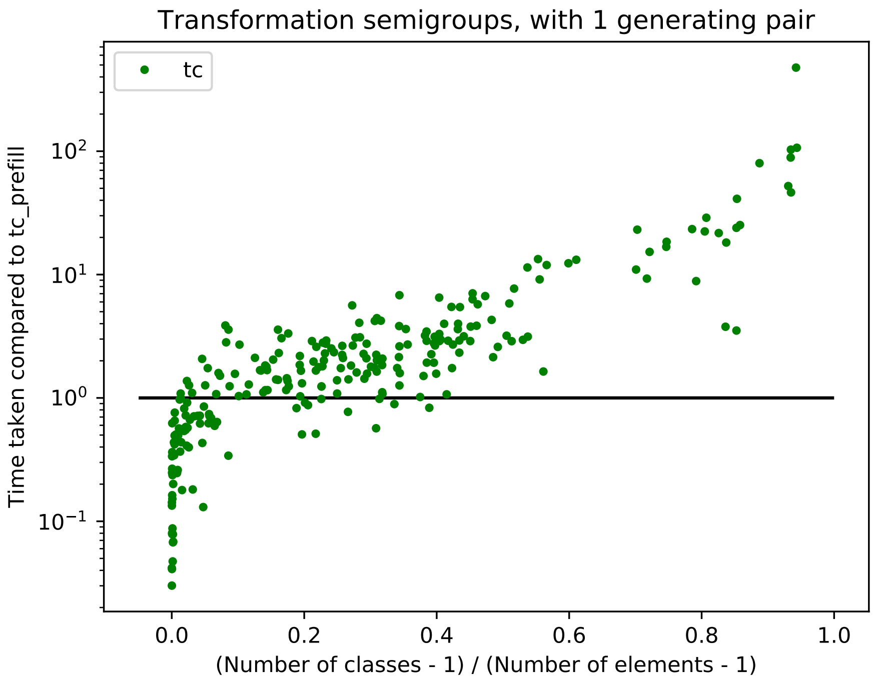


Figure 2.54: Comparison between the two Todd–Coxeter methods, for transformation semigroups with one generating pair.

of `tc` to `tc_prefill` is shown: a black line is drawn on the graphs to indicate the length of time taken by `tc_prefill`, and a data point is then plotted for each test, showing how many times as long `tc` took to complete. As can be seen, `tc` often wins when there are relatively few large classes, and `tc_prefill` is much more likely to win when there are many small classes. This reinforces the idea given in Examples 2.48 and 2.49, that the winning algorithm may depend on number of classes, and it is therefore not clear in advance which algorithm may be better.

This distinction between the performance of `tc` and `tc_prefill` is echoed in an example for right congruences, as shown in Figure 2.56. Interestingly, it appears that for right congruences `tc_prefill` is even more likely to be effective, perhaps because in ToddCoxeterRight so little time is spent applying relations from W, so the work is almost finished by the time the table has been pre-filled.

Figures 2.57 and 2.58 show data from the same tests, plotted not by the size of the semigroup, but simply in order of the ratio between GAP's runtime and that of the default method in libsemigroups. As can be seen, the libsemigroups methods run much faster than the GAP methods in almost all cases, with the GAP methods taking as much as 7000 times as long for one generating pair. Out of the 500 tests shown in total in these two figures, GAP never performed better than libsemigroups.

Further tests were carried out in a similar way, but using finite presentations instead of concrete representations. The main difference between these tests and the ones described previously is that for each transformation semigroup S that was generated, a finite presentation $\langle X \mid R \rangle$ was found for S, and that presentation was used in tests instead of the concrete representation. This is intended to test which algorithms are effective when the elements of the semigroup are not known in advance. In order to produce a further comparison, the tests for finitely presented semigroups were also run with the kbmag package for GAP [Hol19]. The results are shown on the graphs with the name `kbmag`.

As can be seen in Figures 2.62 to 2.64, the performance of the GAP library over finite presentations is far worse than it was for concrete representations, taking as much as 300,000 times as long as libsemigroups to complete. Due to the excessive times GAP took to complete some tests, the size of semigroups was restricted to 1000. In these tests, unlike for concrete representations, the Knuth–Bendix algorithm tended to outperform GAP, but in general the Todd–Coxeter methods were still faster. It should be noted, however, that these were all congruences that were guaranteed in advance to have a finite number of classes. An arbitrary congruence on a finitely presented semigroup may have an infinite number of classes, and there are many examples in which the Knuth–Bendix algorithm can return an answer but the Todd–Coxeter algorithm cannot (see Example 2.50).

kbmag generally performed worse in tests than the Todd–Coxeter algorithm, but was comparable to our implementation of the Knuth–Bendix algorithm. It generally took around 10 times as long as the complete parallel method (`default` on the graphs).

2.9 Future work

The parallel approach described in this chapter is quite open-ended, and could be extended or improved in several ways. We will now discuss some areas which could bear investigation, given more time to spend on the project.

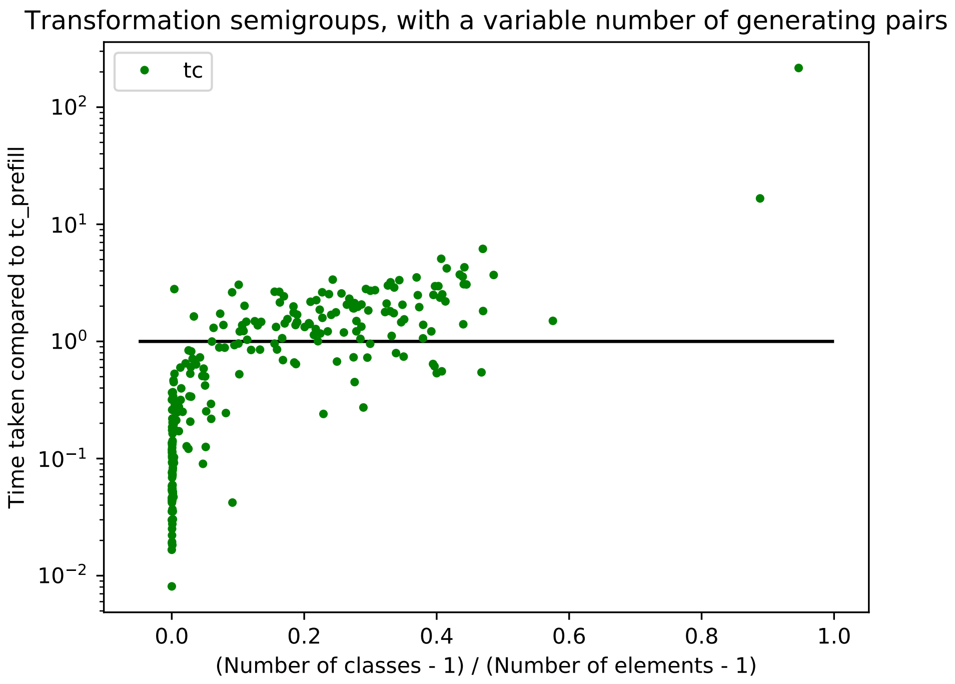


Figure 2.55: Comparison between the two Todd–Coxeter methods, for transformation semigroups with a variable number of generating pairs.

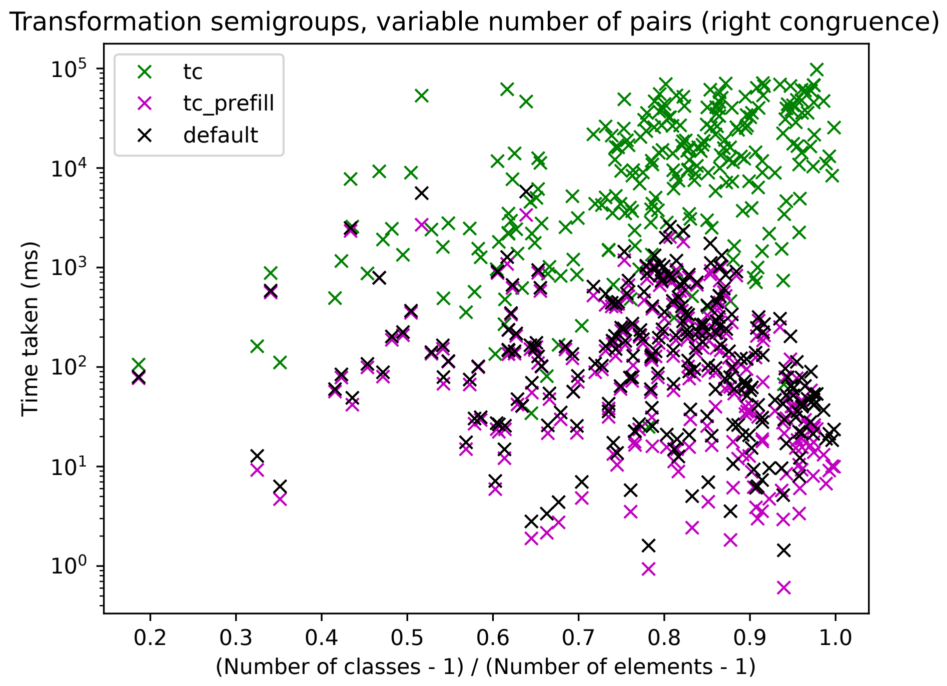


Figure 2.56: Comparison between the two Todd–Coxeter methods, for right congruences over transformation semigroups with a variable number of generating pairs.

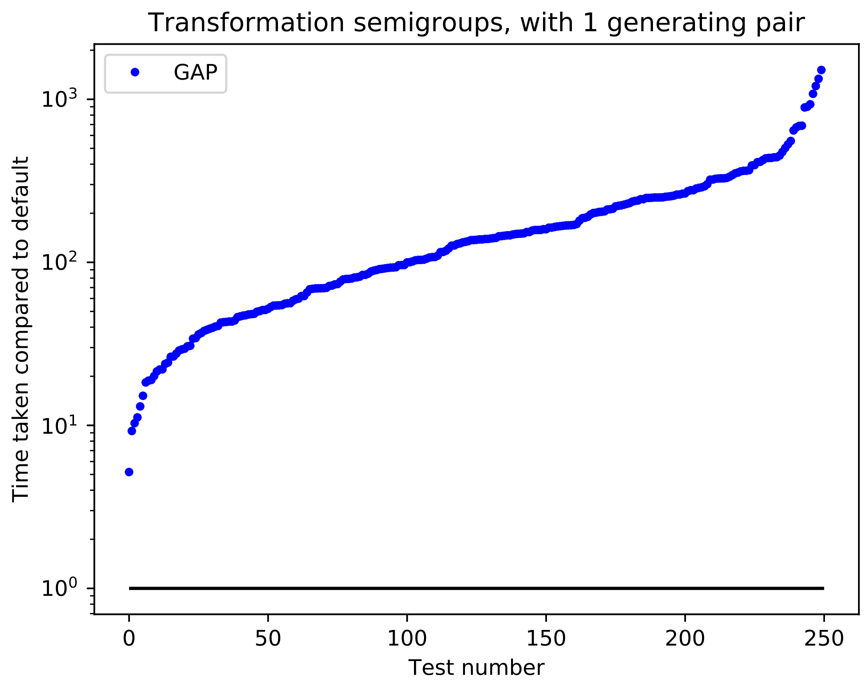


Figure 2.57: Comparison between libsemigroups and the GAP library, for transformation semigroups with one generating pair.

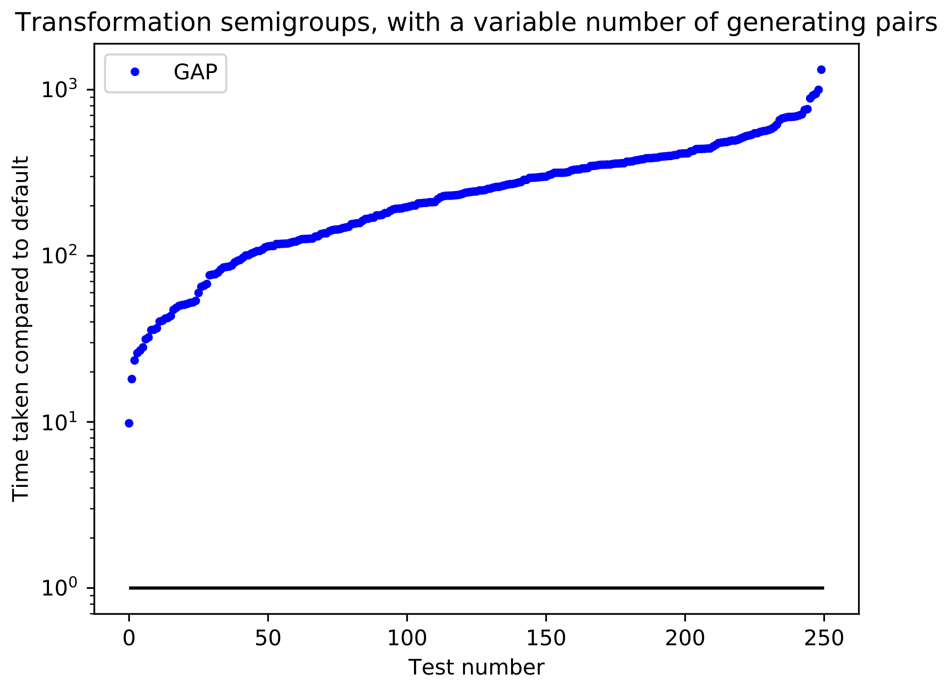


Figure 2.58: Comparison between libsemigroups and the GAP library, for transformation semigroups with a variable number of generating pairs.

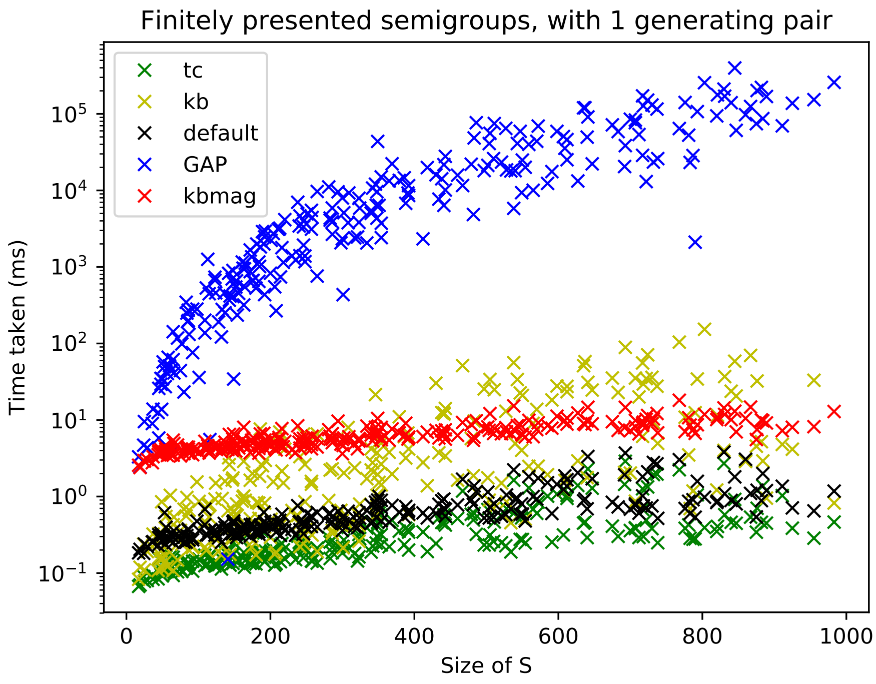


Figure 2.59: Performance of the algorithms on 250 finitely presented semigroups with one generating pair.

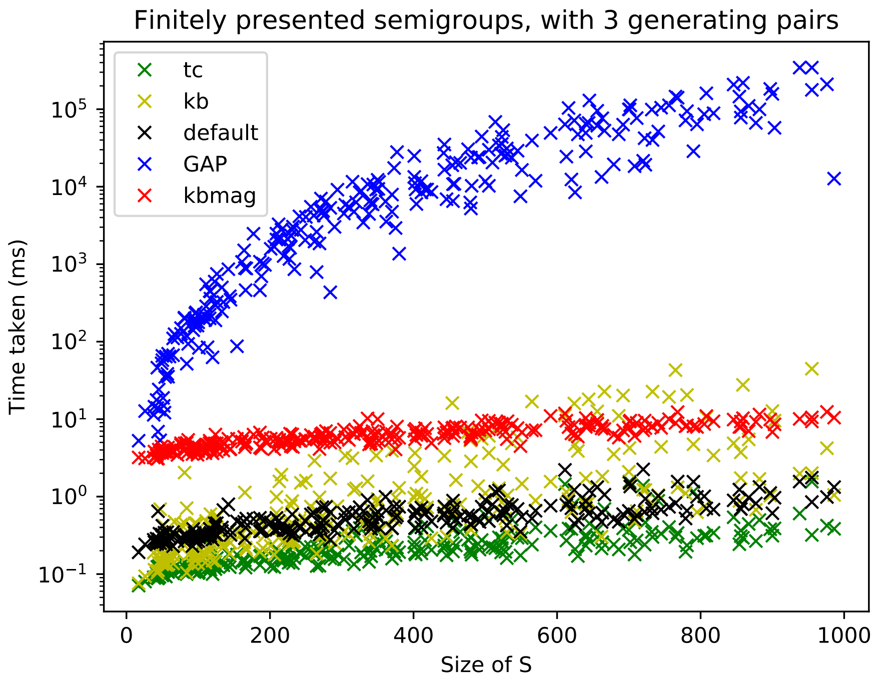


Figure 2.60: Performance of the algorithms on 250 finitely presented semigroups with three generating pairs.

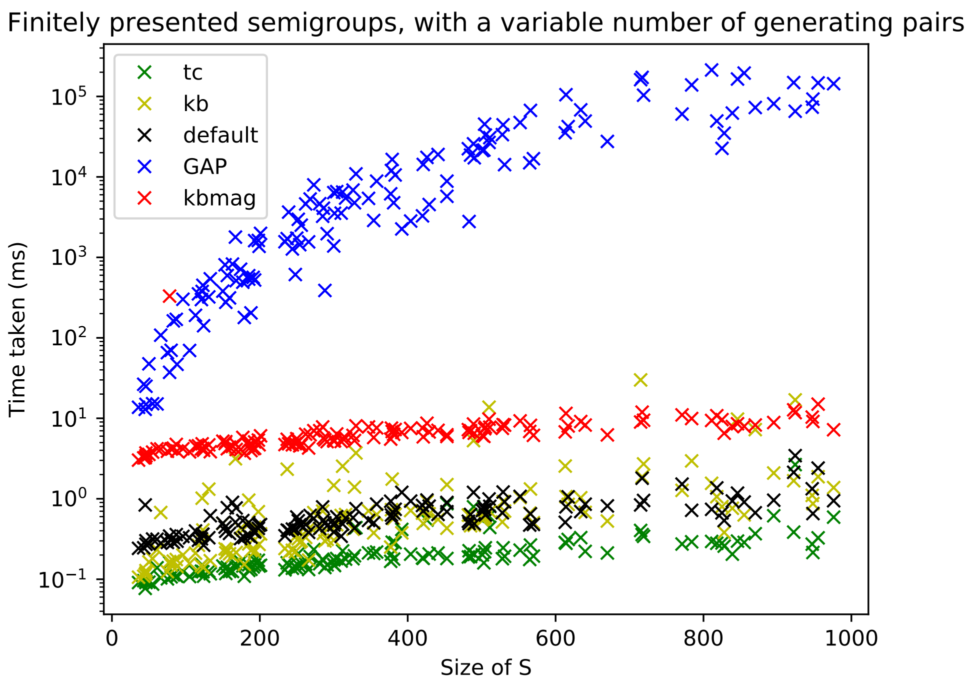


Figure 2.61: Performance of the algorithms on 250 finitely presented semigroups with a variable number of generating pairs.

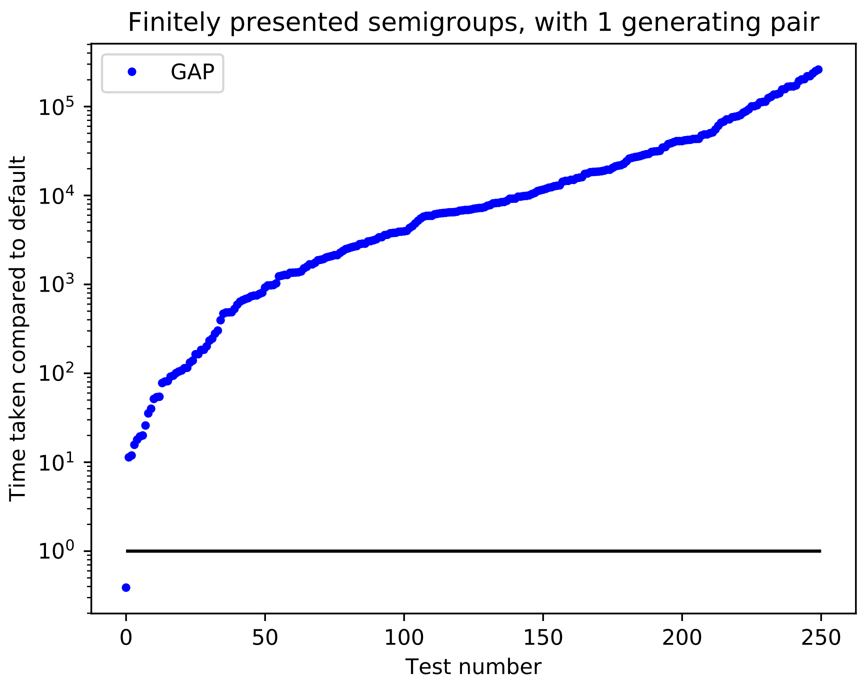


Figure 2.62: Comparison between libsemigroups and the GAP library, for finitely presented semigroups and one generating pair.

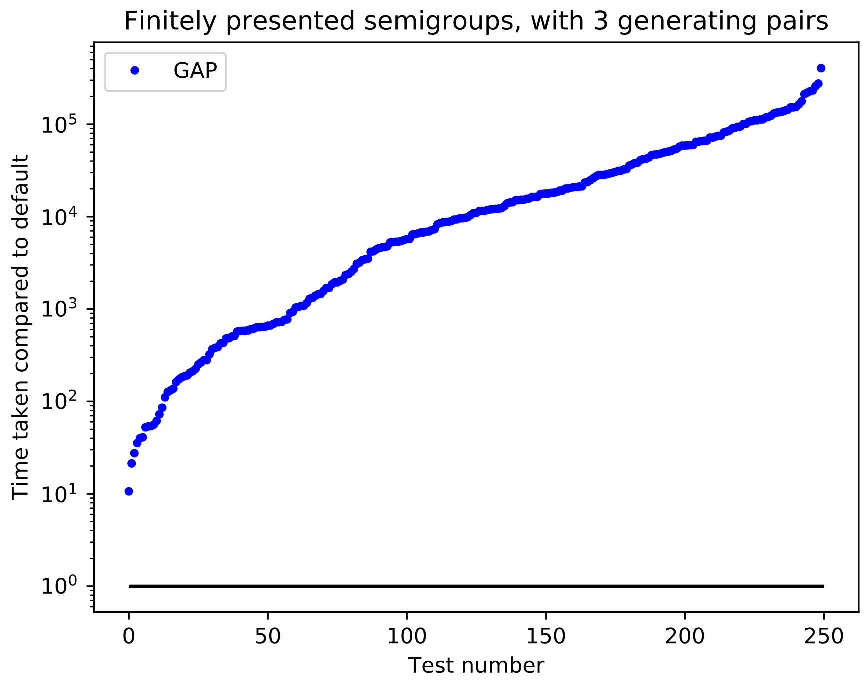


Figure 2.63: Comparison between libsemigroups and the GAP library, for finitely presented semigroups and three generating pairs.

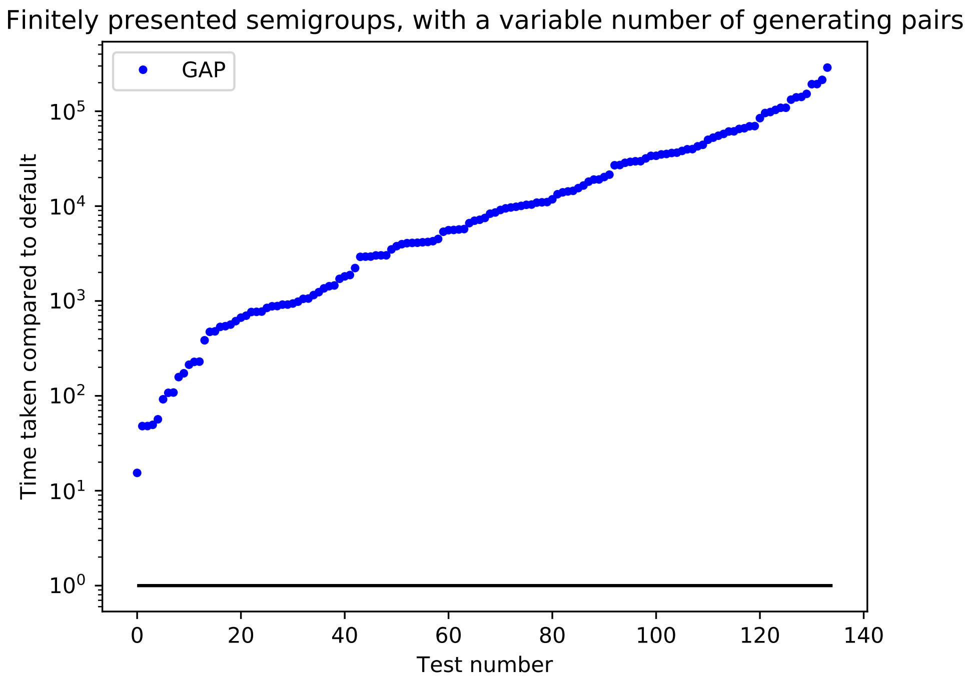


Figure 2.64: Comparison between libsemigroups and the GAP library, for finitely presented semigroups and a variable number of generating pairs.

2.9.1 Pre-filling the Todd–Coxeter algorithm with a left Cayley graph

The pre-filled Todd–Coxeter algorithm, as described in Section 2.6.2, works by starting the procedure with a right Cayley graph for S. In libsemigroups and in the Semigroups package for GAP, a right Cayley graph for a semigroup is found using the Froidure–Pin method, which also returns the corresponding left Cayley graph. Hence, we may also wish to find a way to use the left Cayley graph in the pre-filling process.

As mentioned in Section 2.6.2, it is possible to use the Todd–Coxeter algorithm with a reversed multiplication, essentially studying a semigroup anti-isomorphic to S. It would be possible to apply the same principle, reverse the multiplication of elements in S, and thus use the left Cayley graph instead of the right Cayley graph in pre-filling. This could be run as an additional thread in the parallel procedure.

In many cases, using the left Cayley graph might be very similar in terms of performance to using the right. However, some semigroups have left and right Cayley graphs which are very different. Consider, for example, the right zero semigroup $\mathcal{RZ}_n$, which has n generators, n elements, and the multiplication $xy = y$ for any $x, y \in \mathcal{RZ}_n$. Its right Cayley graph is the complete digraph, where any element can be mapped to any other element using the appropriate generator. Its left Cayley graph is totally disconnected, with each vertex v in a single trivial connected component with n edges taking v to v. With such different left and right Cayley graphs, it seems likely that one piece of information would be much more helpful than the other in calculating congruences on the semigroup.

2.9.2 Interaction between the Knuth–Bendix and Todd–Coxeter algorithms

The Knuth–Bendix and Todd–Coxeter algorithms, as described in this chapter, do not interact with each other in any way. The Knuth–Bendix process runs in one thread, and the Todd–Coxeter process runs in another. However, information from one could perhaps be shared with the other.

The main objective of the Knuth–Bendix algorithm is the addition of new rewriting rules to a rewriting system $\mathbf{R}$ to satisfy the condition of confluence: if a critical pair is found, and a new rule $u \to v$ is added to $\mathbf{R}$, then this gives us a pair of words (u, v) which represents a pair of congruent elements in the congruence we are studying. If a Todd–Coxeter procedure is running in parallel, it would be possible to send the pair of words (u, v) to the Todd–Coxeter thread, which at its next convenience would run $\textsc{Coinc}\big(\textsc{Trace}(1, u),\ \textsc{Trace}(1, v)\big)$ to identify the two corresponding rows in the table.

Conversely, the Todd–Coxeter algorithm may find information which could be used by the Knuth–Bendix algorithm. Firstly, it is trivial to record, for each row i of the table, the word w_i which was first used to describe it: row 1 is identified with the empty word ($w_1 := \varepsilon$), and if a row is created using $\textsc{Add}(i, x)$ then it is assigned the word $w_i x$. For each row i, we now have $\textsc{Trace}(1, w_i) = i$; that is, each row has a word which can act as a representative for its congruence class. If, in a normal run of the Todd–Coxeter algorithm, it is found that two rows i and j represent the same class and must be combined, then this immediately gives a pair of words (w_i, w_j) which represent the same congruence class. This pair of words can be sent to a parallel instance of the Knuth–Bendix algorithm, which can add it as a rule $w_i \to w_j$ (or

$w_j \to w_i$, as dictated by the chosen ordering).

It may be that this sharing of knowledge between the two algorithms would greatly increase the speed of certain calculations; or it may be that the time and space overhead required in the implementation of these ideas would be so great that the algorithms would not speed up at all. Experiments with this idea might show it to be useful, or might suggest that it is not worth pursuing.

2.9.3 Using concrete elements in the Todd–Coxeter algorithm

The Todd–Coxeter procedure uses a finite presentation $\langle X \mid R \rangle$ and a set of extra pairs W in order to calculate information about a congruence over a semigroup S. If S has a concrete representation, then $\langle X \mid R \rangle$ and W must be calculated before the beginning of the Todd–Coxeter algorithm, so that the procedure can use them as parameters. However, once this information has been calculated, no other information about S is used for the rest of the algorithm, which only deals with words, abstract generators and relations.

It may prove helpful to use the concrete representation from S in the the Todd–Coxeter algorithm procedure, if one is available. For instance, when $\textsc{Add}(i, x)$ for some row i and generator x, it would be possible to find an element corresponding to row i, find the element corresponding to the generator x, multiply the two, and see whether a row already exists which represents that element. In this way, we can avoid the unnecessary creation of new rows which would only be deleted by $\textsc{Coinc}$ later.

The pre-filling of Todd–Coxeter tables is one use of the concrete elements that we have already described and implemented, but they may be many more which would be effective.

2.9.4 Left and right congruences with the Knuth–Bendix algorithm

The parallel method in this chapter, and its implementation in libsemigroups, include support for left congruences and right congruences, as well as the more important two-sided congruences. Currently, the Todd–Coxeter and pair enumeration algorithms are the only methods which support left and right congruences, while the Knuth–Bendix algorithm is only applied in the two-sided case. However, there does exist a version of the Knuth–Bendix algorithm which applies to left and right congruences, and it could be considered a useful addition to the parallel algorithm described in this chapter.

The algorithm for right congruences is described in [Sim94, §2.8], and is summarised as follows. Given parameters X, R, and W, as described in Section 2.3, we wish to find the right congruence ρ defined by W on the semigroup S presented by $\langle X \mid R \rangle$. Our goal is to find a rewriting system $\mathbf{R}$ which rewrites two words u and v to the same word if and only if they represent elements of S which are in the same ρ-class. We define a new symbol, '#', which we will use as an additional generator in the new alphabet $Y = X \cup \{\#\}$. We then consider the pairs in W, and produce the set

$$\#W = \{(\#u, \#v) : (u, v) \in W\}.$$

Now, we apply the Knuth–Bendix completion procedure to the presentation $\langle Y \mid R, \#W \rangle$ in the same way as described in Section 2.6.3. The algorithm produces a rewriting system $\mathbf{R}$.

Next we consider the subset $\#X^+ \subseteq Y^+$ defined by

$$\#X^+ = \{\#w : w \in X^+\}.$$

It is shown in [Sim94, §2.8] that $\#X^+$ is a union of $\overset{*}{\leftrightarrow}_{\mathbf{R}}$-classes, and that the $\overset{*}{\leftrightarrow}_{\mathbf{R}}$-classes contained in $\#X^+$ are in one-to-one correspondence with the ρ-classes of X^+. That is, given a pair of words $(u, v) \in X^+$, the elements $[u]$ and $[v]$ in S lie in the same ρ-class if and only if the words $\#u$ and $\#v$ are rewritten to the same word by $\mathbf{R}$. A symmetric approach would work for left congruences, with the # symbols being added to the ends of words instead of to the start.

This additional method could be included in the parallel algorithm, and in libsemigroups, without too much extra work. It may be that it would perform well, returning faster than the Todd–Coxeter algorithm on some examples – but it would need to be benchmarked in a manner similar to Section 2.8 in order to establish whether it were worth running in either the concrete case or the finite presentation case. Once this were decided, we would be able to remove the two cross (✘) symbols in the 'KB' column of Table 2.47, and change each one to either a tick (✓) or a tilde ($\sim$).

Chapter 3

Converting between representations

A congruence is a binary relation, and therefore is formally described as a set of pairs – a subset of $S \times S$. In both computational and mathematical settings, it is worth thinking about how a congruence could be stored.

One approach to storing a congruence ρ on a semigroup S is simply to store every one of its pairs. In principle, it is possible to store ρ in this way if and only if S is finite. However, this could well use a lot of storage – even the trivial congruence would use $O(|S|)$ space, and in general a congruence could even use $O(|S|^2)$ space.

In Chapter 2 we looked in detail at how a congruence can be represented by a set of generating pairs. As we found there, a congruence can be described by a subset $\mathbf{R} \subseteq \rho$, which in many cases can be very small. This is one very generic way of representing congruences, in two senses: firstly that it can be used for any finite semigroup; and secondly that it can be used for left and right congruences.

However, there are other ways to view congruences in certain circumstances: some semigroups have properties such as being an inverse semigroup or being a group, which allow additional things to be said about their congruences; and some specific congruences have special properties, such as being Rees, which allows them to be represented in a certain way. In this chapter, we will describe some important ways of representing congruences, and then consider ways of converting one to another. Section numbers for the different representations and the ways they can be converted to one another are summarised in Table 3.1.

3.1 Ways of representing a congruence

We will begin by describing several different ways of representing a congruence. These representations all exist in some form in GAP [GAP18] or the Semigroups package [M$^+$19].

3.1.1 Generating pairs

Recall that a congruence ρ on a semigroup S can be stored using a subset of the pairs in ρ. If $\mathbf{R}$ is a subset of $S \times S$, then we can say that $\mathbf{R}$ *generates* a congruence. The congruence *generated*

			…to…			
		GP	NS	LT	KT	RC
	Generating pairs 3.1.1	■	3.2.7	3.2.3	3.2.5	3.3.2
	Normal subgroup (groups) 3.1.2	3.2.7	■	3.2.1	3.2.7	3.2.7
From…	Linked triple ((0-)simple) 3.1.3	3.2.4	3.2.1	■	3.2.7	3.2.7
	Kernel–trace (inverse) 3.1.4	3.3.1	3.2.7	3.2.7	■	3.2.6
	Rees congruence 3.1.5	3.2.2	3.2.7	3.2.7	3.2.6	■

Table 3.1: Section references to algorithms for converting between different congruence representations. Grey references represent open problems.

by $\mathbf{R}$ is defined as the least congruence on S containing all the pairs in $\mathbf{R}$; equivalently, it is defined as the intersection of all congruences on S containing all the pairs in $\mathbf{R}$. It is denoted by $\mathbf{R}^\sharp$ (see Theorem 1.39). We have similarly defined the left congruence generated by $\mathbf{R}$ (denoted by $\mathbf{R}^\lhd$) and the right congruence generated by $\mathbf{R}$ (denoted by $\mathbf{R}^\rhd$). A full explanation of how generating pairs can be used to represent congruences is given in Section 1.6, and an approach for computing properties of congruences using their generating pairs is given in Chapter 2.

Given a set of pairs $\mathbf{R}$, we may wish to produce the congruence $\mathbf{R}^\sharp$ and represent it using one of the other methods described in this chapter. It is of course possible to calculate the set of all pairs in $\mathbf{R}^\sharp$ and convert that to the other representation; however, in order to find other representations with as little work as possible, it is desirable to use the pairs in $\mathbf{R}$ directly, calculating as few extra pairs as possible – see, for example, Sections 3.2.3 and 3.2.5. Conversely, if we wish to convert another representation for a congruence ρ to a set of generating pairs, it is desirable to find as small a set of pairs as possible – see, for example, Sections 3.2.2 and 3.2.4. When converting between generating pairs and other representations, these will be the goals.

3.1.2 Groups: normal subgroups

In group theory, it is unusual to encounter discussion of congruences. This is because a group's congruences are closely related to another structure – its normal subgroups – and any questions we could ask about a group's congruences are easily described using normal subgroups instead. Recall that a subgroup N of a group G is *normal* if and only if $g^{-1}ng \in N$ for all $g \in G$ and $n \in N$; recall also that a *coset* of N is the set Ng or gN for some $g \in G$, and that $Ng = gN$ if N is normal. The following theorem shows how a group's normal subgroups are in bijective correspondence with its congruences.

Theorem 3.2. *Let G be a group. If ρ is a congruence on G, then the ρ-class containing the identity is a normal subgroup of G. Conversely, if N is a normal subgroup of G, then its cosets are the classes of a congruence on G.*

Proof. First, let ρ be a congruence on G, and let I be the ρ-class containing the identity 1. First we show that I is a subgroup: if $a, b \in I$ then $ab \ \rho \ 11 = 1$, so $ab \in I$. Furthermore, we have $(a, 1) \in \rho$, so $(aa^{-1}, 1a^{-1}) = (1, a^{-1}) \in \rho$, so $a^{-1} \in I$, and so I is a subgroup. To show I is normal, let $g \in G$ and $i \in I$. Observe that $g^{-1}ig \ \rho \ g^{-1}1g = g^{-1}g = 1$, so $g^{-1}ig \in I$, as required.

To show the converse, let N be a normal subgroup of G, and let ν be the equivalence on G whose classes are the cosets of N. If $(x,y),(s,t) \in \nu$, then $Nx = Ny$ and $sN = tN$. Hence $Nxs = Nys = ysN = ytN = Nyt$, so we have $(xs, yt) \in \nu$, meaning that ν is a congruence as required. □

This theorem means that any information which can be taken from a congruence can instead be taken from a normal subgroup, and so congruences on a group need never be studied directly. We even have the fortunate property that the containment of normal subgroups follows the containment of the corresponding congruences.

It is possible to calculate the normal subgroups of a finite group relatively quickly, using a variety of well-known algorithms. One method for finding the normal subgroups of a finite group is given in [Hul98]; this is the method used in the most general case by GAP [GAP18], though more specific methods are used for certain specific categories of group. In the case of an infinite group, it may be impossible to find all normal subgroups – indeed, this problem is undecidable in general [Mil92, Theorem 3.17] – but the LowIndexSubgroups algorithm [HEO05, §5.4] may be used to find all normal subgroups up to a given index, given a small modification to exclude subgroups which are not normal [HEO05, §5.5].

The other structures discussed in this section represent congruences on other categories of semigroup in a similar way.

3.1.3 Completely (0-)simple semigroups: linked triples

There is a special way of describing a congruence on a completely simple or completely 0-simple semigroup: using a linked triple. We will start by explaining the terms *completely simple* and *completely 0-simple*, then we will define a semigroup's linked triples and explain how they are related to its congruences.

Definition 3.3. A semigroup S is:

- **simple** if its only ideal is S;
- **0-simple** if it contains a zero, and has precisely two ideals.

Simple and 0-simple semigroups are closely related. Note that if S is a simple semigroup, then S^0, the semigroup created by appending a zero element to S, is 0-simple. A 0-simple semigroup's ideals are $\{0\}$ and S. Note also that the trivial semigroup is simple but not 0-simple.

Next, we consider a slightly stronger condition, after a preliminary definition relating to idempotents.

Definition 3.4. An idempotent $p \in S$ is **primitive** if it is non-zero and there is no other non-zero idempotent $i \in S$ such that $ip = pi = i$.

Definition 3.5. A semigroup is:

- **completely simple** if it is simple and contains a primitive idempotent;
- **completely 0-simple** if it is 0-simple and contains a primitive idempotent.

Definitions 3.3 and 3.5 are equivalent for finite semigroups – that is to say, a finite semigroup is completely simple if and only if it is simple, and it is completely 0-simple if and only if it is 0-simple. Some of the conversions described in this chapter will be applicable only to finite semigroups, and in those circumstances we will refer to *finite simple* or *finite 0-simple* semigroups, knowing that these are completely simple or completely 0-simple, respectively. Note that a finite semigroup is simple if and only if it is $\mathscr{J}$-trivial.

Completely simple and completely 0-simple semigroups have a strong and useful isomorphism property, which allows us to say a great deal about their structure and, in particular, their congruences. We will consider first the more complicated case, that of completely 0-simple semigroups, and then at the end of this section we will explain how this theory can be adapted for the much less complicated case, that of completely simple semigroups.

Definition 3.6 ([How95, §3.2]). A **Rees 0-matrix semigroup** $\mathcal{M}^0[T; I, \Lambda; P]$ is the set

$$(I \times T \times \Lambda) \cup \{0\}$$

with multiplication given by

$$(i, a, \lambda) \cdot (j, b, \mu) = \begin{cases} (i, ap_{\lambda j}b, \mu) & \text{if } p_{\lambda j} \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

for $(i, a, \lambda), (j, b, \mu) \in I \times T \times \Lambda$, and $0x = x0 = 0$ for all $x \in \mathcal{M}^0[T; I, \Lambda; P]$, where

- T is a semigroup,
- I and Λ are non-empty index sets,
- P is a $|\Lambda| \times |I|$ matrix with entries $(p_{\lambda i})_{\lambda \in \Lambda, i \in I}$ taken from T^0,
- 0 is an element not in $I \times T \times \Lambda$.

We will require a certain property of the matrix P, which we should define first: we call a matrix **regular** if it contains at least one non-zero entry in each row and each column.

The following theorem shows how we can use Rees 0-matrix semigroups to classify completely 0-simple semigroups.

Theorem 3.7 (Rees). *Every completely 0-simple semigroup is isomorphic to a Rees 0-matrix semigroup $\mathcal{M}^0[G; I, \Lambda; P]$, where G is a group and P is regular. Conversely, every such Rees 0-matrix semigroup is completely 0-simple.*

Proof. Theorem 3.2.3 in [How95]. □

Now we can replace any completely 0-simple semigroup with its isomorphic Rees 0-matrix semigroup when we wish to perform any isomorphism-invariant calculations – hence we can restrict our further investigations just to this type of semigroup. Note that methods exist in the Semigroups package for performing this replacement: in a session, we can decide whether a finite semigroup S is completely 0-simple using `IsZeroSimpleSemigroup`, and if the result is positive we can use `IsomorphismReesZeroMatrixSemigroup` to obtain a Rees 0-matrix semigroup isomorphic to S, as well as a map between the elements of the two semigroups [M+19].

If $\mathcal{M}^0[G; I, \Lambda; P]$ is finite, then G, I, Λ and P must all be finite, so all the components of the semigroup that we work with will also be finite.

Next we consider the congruences of a finite 0-simple semigroup.

Definition 3.8 ([How95, §3.5])**.** Let S be a finite Rees 0-matrix semigroup $\mathcal{M}^0[G; I, \Lambda; P]$ over the group G with regular matrix P. A **linked triple** on S is a triple

$$(N, \mathcal{S}, \mathcal{T})$$

consisting of a normal subgroup $N \trianglelefteq G$, an equivalence relation $\mathcal{S}$ on I and an equivalence relation $\mathcal{T}$ on Λ, such that the following are satisfied:

(i) $\mathcal{S} \subseteq \varepsilon_I$, where $\varepsilon_I = \{(i, j) \in I \times I \,|\, \forall \lambda \in \Lambda : p_{\lambda i} = 0 \iff p_{\lambda j} = 0\}$,

(ii) $\mathcal{T} \subseteq \varepsilon_\Lambda$, where $\varepsilon_\Lambda = \{(\lambda, \mu) \in \Lambda \times \Lambda \,|\, \forall i \in I : p_{\lambda i} = 0 \iff p_{\mu i} = 0\}$,

(iii) For all $i, j \in I$ and $\lambda, \mu \in \Lambda$ such that $p_{\lambda i}, p_{\lambda j}, p_{\mu i}, p_{\mu j} \neq 0$ and either $(i, j) \in \mathcal{S}$ or $(\lambda, \mu) \in \mathcal{T}$, we have $q_{\lambda\mu ij} \in N$, where

$$q_{\lambda\mu ij} = p_{\lambda i} p_{\mu i}^{-1} p_{\mu j} p_{\lambda j}^{-1}.$$

We can associate the linked triples of a finite 0-simple semigroup with its non-universal congruences, as follows.

Theorem 3.9. *Let S be a Rees 0-matrix semigroup defined with a group and a regular matrix. There exists a bijection Γ between the non-universal congruences on S and the linked triples on S.*

Proof. Theorem 3.5.8 in [How95] □

This theorem shows us an alternative way to look at congruences on completely 0-simple semigroups, just as normal subgroups show us an alternative way to look at congruences on groups. However, in order to use this at all in a computational setting, we must have a concrete function Γ which we can use to convert a congruence to a linked triple and back again, rather than just the knowledge that such a function exists – indeed, describing such a function is the purpose of this section. We define the function Γ as follows.

Definition 3.10 ([How95, §3.5])**.** Let S be a Rees 0-matrix semigroup $\mathcal{M}^0[G; I, \Lambda; P]$ over a group G and a regular matrix P. The **linked triple function** Γ of S is defined, for ρ a non-universal congruence, by

$$\Gamma : \rho \mapsto (N_\rho, \mathcal{S}_\rho, \mathcal{T}_\rho),$$

so that it maps any non-universal congruence onto a triple whose entries are defined as follows.

The relation $\mathcal{S}_\rho \subseteq I \times I$ is defined by the rule that $(i, j) \in \mathcal{S}_\rho$ if and only if $(i, j) \in \varepsilon_I$ and

$$(i, p_{\lambda i}^{-1}, \lambda) \;\rho\; (j, p_{\lambda j}^{-1}, \lambda)$$

for all $\lambda \in \Lambda$ such that $p_{\lambda i} \neq 0$ (and hence $p_{\lambda j} \neq 0$). Similarly, the relation $\mathcal{T}_\rho \subseteq \Lambda \times \Lambda$ is defined by the rule that $(\lambda, \mu) \in \mathcal{T}_\rho$ if and only if $(\lambda, \mu) \in \varepsilon_\Lambda$ and

$$(i, p_{\lambda i}^{-1}, \lambda) \;\rho\; (i, p_{\mu i}^{-1}, \mu)$$

for all $i \in I$ such that $p_{\lambda i} \neq 0$ (and hence $p_{\mu i} \neq 0$). Finally, we define the normal subgroup $N_\rho \trianglelefteq G$ as follows. First, fix some $\xi \in \Lambda$, a row of the matrix P. Since P is regular, row ξ must contain a non-zero entry – fix some $k \in I$ such that $p_{\xi k} \neq 0$. Now we can define

$$N_\rho = \{a \in G \mid (k, a, \xi) \;\rho\; (k, 1_G, \xi)\},$$

where 1_G is the identity in the group G.

The inverse of Γ is then such that, for a linked triple $(N, \mathcal{S}, \mathcal{T})$, the congruence $(N, \mathcal{S}, \mathcal{T})\Gamma^{-1}$ is equal to

$$\Big\{ \big((i, a, \lambda), (j, b, \mu)\big) \;\Big|\; (p_{\xi i} a p_{\lambda k})(p_{\xi j} b p_{\mu k})^{-1} \in N, (i, j) \in \mathcal{S}, (\lambda, \mu) \in \mathcal{T} \Big\} \cup \big\{(0, 0)\big\},$$

where $\xi \in \Lambda$ and $k \in I$ can be any elements such that $p_{\xi i}$ and $p_{\lambda k}$ are both non-zero, as shown in [How95, Lemma 3.5.6]. Note that ξ and k definitely exist, since P is a regular matrix, and so column i and row λ must each contain a non-zero entry.

Note that the definition of N_ρ does not depend on the choice of ξ and k. Independence from the choice of ξ is established by the following lemma, and independence from the choice of k follows by a similar argument.

Lemma 3.11. *Let $\xi_1, \xi_2 \in \Lambda$ and $k \in I$ such that $p_{\xi_1 k} \neq 0$ and $p_{\xi_2 k} \neq 0$. Then*

$$(k, a, \xi_1) \;\rho\; (k, 1_G, \xi_1) \quad \textit{if and only if} \quad (k, a, \xi_2) \;\rho\; (k, 1_G, \xi_2)$$

for all $a \in G$.

Proof. Assume that $(k, a, \xi_1) \;\rho\; (k, 1_G, \xi_1)$. We can right-multiply both sides by $(k, p_{\xi_1 k}^{-1}, \xi_2)$ to give

$$(k, a, \xi_1)(k, p_{\xi_1 k}^{-1}, \xi_2) \;\rho\; (k, 1_G, \xi_1)(k, p_{\xi_1 k}^{-1}, \xi_2),$$

which simplifies to

$$(k, a p_{\xi_1 k} p_{\xi_1 k}^{-1}, \xi_2) \;\rho\; (k, 1_G p_{\xi_1 k} p_{\xi_1 k}^{-1}, \xi_2),$$

and then to $(k, a, \xi_2) \;\rho\; (k, 1_G, \xi_2)$, as required. The converse argument is identical, swapping ξ_1 for ξ_2. $\square$

Our discussion so far has focused on 0-simple semigroups, but very similar structures exist for completely *simple* semigroups. They are isomorphic to **Rees matrix semigroups**, and linked triples can be defined on them in almost exactly the same way, except for the removal of complications related to the zero element. A Rees matrix semigroup follows Definition 3.6 but with the removal of the zero element, and linked triples follow Definition 3.8, where the restrictions related to placements of 0 in P are irrelevant. It should also be noted that even the universal congruence has a linked triple in this case – $(G, I \times I, \Lambda \times \Lambda)$ – so the domain of Γ is not only the non-universal congruences, but all congruences on S.

3.1.4 Inverse semigroups: kernel–trace pairs

An inverse semigroup also has a structure which can be used in place of its congruences: its *kernel–trace pairs* (sometimes confusingly known as “congruence pairs”). In [Tor14b, Chapter 5] the author focused on a computational use of kernel–trace pairs to solve problems about congruences. They can certainly be used effectively to carry out calculations, in a similar way to linked triples.

The basic theory about kernel–trace pairs is presented here, for reference. In all these definitions, S is an inverse semigroup, E is the set of idempotents in S, and and ρ is a congruence on S. Recall that E is an inverse subsemigroup of S. This is standard background theory, which is adapted from [How95, §5.3].

Definition 3.12. The **kernel** of ρ is $\bigcup_{e\in E}[e]_\rho$, the union of all the ρ-classes of S which contain idempotents. It is denoted by $\ker\rho$.

Definition 3.13. The **trace** of ρ is $\rho\cap(E\times E)$, the restriction of ρ to the idempotents of S. It is denoted by $\operatorname{tr}\rho$.

We will shortly see that a congruence on S is completely defined by its kernel and trace. First we will approach kernel–trace pairs from an abstract route which will help us to classify the congruences on S completely. We start with two different definitions of the word "normal", one for subsemigroups and one for congruences.

Definition 3.14. A subsemigroup K of S is called **normal** if it is *full* (contains all the idempotents of S) and *self-conjugate* ($a^{-1}xa\in K$ for all $x\in K, a\in S$).

Definition 3.15. A congruence τ on E is **normal** in S if

$$(a^{-1}ea, a^{-1}fa)\in\tau$$

for every pair $(e,f)\in\tau$ and every element $a\in S$.

Now we can define a *kernel–trace pair*, an abstract structure which relates very closely to a congruence.

Definition 3.16. A **kernel–trace pair** on S is a pair (K,τ) consisting of a normal subsemigroup K of S and a normal congruence τ on E, such that

(i) If $ae\in K$ and $(e,a^{-1}a)\in\tau$, then $a\in K$

(ii) If $a\in K$, then $(aa^{-1},a^{-1}a)\in\tau$

for all elements $a\in S$ and $e\in E$.

Now we state the result which identifies an abstract kernel–trace pair with the kernel and trace of a congruence, and allows us to calculate information about ρ by using $\ker\rho$ and $\operatorname{tr}\rho$ directly.

Theorem 3.17. *Let S be an inverse semigroup. There exists a bijection Ψ from the congruences on S to the kernel–trace pairs on S, defined by*

$$\Psi:\rho\mapsto(\ker\rho,\operatorname{tr}\rho),$$

and its inverse satisfies

$$\Psi^{-1}:(K,\tau)\mapsto\{(x,y)\in S\times S\mid xy^{-1}\in K,(x^{-1}x,y^{-1}y)\in\tau\}.$$

Proof. Theorem 5.3.3 in [How95]. □

This theorem tells us everything we need to know about kernel–trace pairs and their relationship to congruences on an inverse semigroup. Once we have the kernel–trace pair of a congruence, we can solve any problem we wish to using the kernel and trace alone, and computational problems such as determining whether a given pair (x, y) lies in the congruence are much faster than using generating pairs directly [Tor14b, §6.1.3]. However, we may find that if a congruence is specified initially using generating pairs, it may be costly to find its kernel–trace pair in the first place; Section 3.2.5 presents a relatively fast method for finding a kernel–trace pair.

3.1.5 Rees congruences

Recall that a *Rees congruence* is a congruence on a semigroup S with a distinguished congruence class I which is a two-sided ideal of S, and in which every other congruence class is a singleton. We may write this congruence as ρ_I, and we may write its quotient S/ρ_I as S/I. Hence, a pair (x, y) lies in ρ_I if and only if $x = y$ or x and y both lie in I.

Some or all of a semigroup's congruences may be Rees: in particular, since S is an ideal of S, the universal congruence $S \times S$ is a Rees congruence which could be written as ρ_S. If S has a zero 0, then $\{0\}$ is an ideal and so the trivial congruence Δ_S is a Rees congruence which could be written as $\rho_{\{0\}}$.

A good example of a semigroup with many Rees congruences is the monoid of all order-preserving transformations $\mathcal{O}_n$. All of its congruences are Rees, apart from one – the trivial congruence $\Delta_{\mathcal{O}_n}$, which is not Rees because $\mathcal{O}_n$ does not contain a zero [LS99]. Some examples of semigroups whose congruences are all Rees can be found in [Gar91, §5].

In order to use the theory of Rees congruences in a computational setting, we must consider how a Rees congruence can be stored on a computer. To store a Rees congruence ρ_I we do not have to store a large set of pairs, or even any information about the congruence's classes. We only need to store the ideal I itself, along with the overall semigroup S, and we have everything we need to know about the congruence. Furthermore, we do not even need to store all the elements of I, but just a generating set for it. This could be a generating set for I as a subsemigroup of S, or better yet, an ideal generating set (see Section 1.8). An ideal generating set can be even smaller than a subsemigroup generating set, since a single generator a, when considered an ideal generator, gives rise to elements xay for all $x, y \in S^1$ rather than just for all $x, y \in I$.

The Semigroups package for GAP takes the approach of storing an ideal to represent a Rees congruence, and will store any generating set that is available for the ideal, whether as a subsemigroup or as an ideal.

3.2 Converting between representations

In Section 3.1 we presented five different ways of representing a congruence. In this section, we present a survey of the different ways in which they can be converted to each other. Table 3.1 summarises the methods which exist, and the sections in which they are described.

3.2.1 Normal subgroups and linked triples

In this section we will consider how to convert between a normal subgroup (which represents a congruence on a group) and a linked triple (which represents a congruence on a simple semigroup). This conversion is rather trivial, but is presented as a good example of how different congruence representations can be closely related.

Any group is a completely simple semigroup. In fact, since any group G has precisely one $\mathscr{H}$-class, it is isomorphic to the Rees matrix semigroup $\mathcal{M}[G; I, \Lambda; P]$ where $|I| = |\Lambda| = 1$ and P is the 1×1 matrix (1_G). Let $\phi : G \to \mathcal{M}[G; I, \Lambda; P]$ be the isomorphism defined by $(g)\phi = (1, g, 1)$.

As described in Theorem 3.2, a congruence ρ on a group G is associated with a normal subgroup $N \trianglelefteq G$, according to the rule that $x \; \rho \; y$ if and only if $xy^{-1} \in N$. Similarly, as described in Definition 3.10, a congruence ρ' on $\mathcal{M}[G; I, \Lambda; P]$ is associated with a linked triple $(N', \mathcal{S}, \mathcal{T})$, according to the rule that $(i, a, \lambda) \; \rho \; (j, b, \mu)$ if and only if $i \; \mathcal{S} \; j$, $\lambda \; \mathcal{T} \; \mu$, and $(p_{1i} a p_{\lambda 1})(p_{1j} b p_{\mu 1})^{-1} \in N'$. Since $|I| = |\Lambda| = 1$ and $P = (1_G)$, this last condition simplifies to $ab^{-1} \in N'$.

Let ρ' be the congruence on $\mathcal{M}[G; I, \Lambda; P]$ such that $(x)\phi \; \rho' \; (y)\phi$ (i.e. $(1, x, 1) \; \rho' \; (1, y, 1)$) if and only if $x \; \rho \; y$. The condition defining ρ, that $xy^{-1} \in N$, is equivalent to the condition defined by the linked triple $(N, \Delta_I, \Delta_\Lambda)$, since I and Λ are both trivial. Hence any normal subgroup N corresponds to the linked triple $(N, \Delta_I, \Delta_\Lambda)$, making linked triples on groups very easy to deal with.

3.2.2 Generating pairs of a Rees congruence

A natural question, given an ideal I, is how to find a set of generating pairs for the Rees congruence ρ_I. In this section we will limit our discussion to finite semigroups.

Theorem 3.18. *Let S be a finite semigroup, and let I be an ideal of S. If X is an ideal generating set for I (see Section 1.8) and M is the minimal ideal of S (which may or may not be equal to I), then*

$$X \times M$$

is a set of generating pairs for the Rees congruence ρ_I.

Proof. Let ρ be the congruence generated by $X \times M$. First we show that $\rho \subseteq \rho_I$, and then that $\rho_I \subseteq \rho$.

Let $(i, m) \in X \times M$. We have $X \subseteq I$ since X is a generating set for I, and $M \subseteq I$ since M is contained in any ideal of S. Hence i and m both lie in I, so they are in the same class of the Rees congruence: $(i, m) \in \rho_I$. Hence $X \times M \subseteq \rho_I$, and so ρ (the least congruence containing $X \times M$) must also be contained in ρ_I. Hence $\rho \subseteq \rho_I$.

Now let $(a, b) \in \rho_I$; we wish to show that $(a, b) \in \rho$. If $a = b$ then we certainly have $(a, b) \in \rho$. Otherwise we must have $a, b \in I$. Since X *generates* I, we have $I = S^1 X S^1$. Therefore we can write

$$a = s_1 x_1 t_1, \quad b = s_2 x_2 t_2,$$

for some $x_1, x_2 \in X$ and $s_1, s_2, t_1, t_2 \in S^1$.

Now choose some $m \in M$. By definition $(x_1, m), (x_2, m) \in \rho$ since $X \times M \subseteq \rho$, and by the compatibility properties of a congruence,

$$(s_1 x_1 t_1, s_1 m t_1), (s_2 x_2 t_2, s_2 m t_2) \in \rho.$$

Since $m \in M$, we must have $s_1 m t_1, s_2 m t_2 \in M$. Let x_0 be an arbitrary element of X. We see $(x_0, s_1 m t_1), (x_0, s_2 m t_2) \in X \times M$, and so by transitivity $(s_1 m t_1, s_2 m t_2) \in \rho$. Hence

$$a \ = \ s_1 x_1 t_1 \ \rho \ s_1 m t_1 \ \rho \ s_2 m t_2 \ \rho \ s_2 x_2 t_2 = b,$$

and $(a, b) \in \rho$ as required. $\square$

3.2.3 Linked triple from generating pairs

In [Tor14a, §6.1] it is observed that calculating information about a congruence using its linked triple is much faster than using a set of generating pairs. However, it may well be that a congruence on a finite simple or finite 0-simple semigroup is specified by generating pairs, and we do not know its linked triple *a priori*. In this case, we will need to calculate the congruence's linked triple before we can use it to calculate any other information. We could do this by enumerating all the elements of all the classes of the congruence, and then simply looking up the relevant information to find the linked triple. However, this is very expensive, and once the classes are enumerated there is likely no need for the linked triple, since all information about the congruence has already been calculated.

In [Tor14b, §3.2], the author presents an algorithm to calculate a congruence's linked triple directly from a set of generating pairs, calculating as few extra pairs as possible. This algorithm performs quickly, representing a big improvement on using a more naïve algorithm to find the linked triple [Tor14b, §6.1.2]. The algorithm is justified by the following definition and theorem from [Tor14b].

Definition 3.19 ([Tor14b, Definition 3.10])**.** Let $S = \mathcal{M}^0[G; I, \Lambda; P]$ be a finite Rees 0-matrix semigroup over a group G with regular matrix P, and let $\mathbf{R} \subseteq S \times S$ be a relation on it. We define the relations $\mathbf{R}|_I$ and $\mathbf{R}|_\Lambda$ by

$$\mathbf{R}|_I = \big\{(i, j) \in I \times I \;\big|\; (i, a, \lambda) \ \mathbf{R} \ (j, b, \mu)\big\},$$

$$\mathbf{R}|_\Lambda = \big\{(\lambda, \mu) \in \Lambda \times \Lambda \;\big|\; (i, a, \lambda) \ \mathbf{R} \ (j, b, \mu)\big\}.$$

Theorem 3.20 ([Tor14b, Theorem 3.11])**.** *Let $S = \mathcal{M}^0[G; I, \Lambda; P]$ be a finite 0-simple semigroup over a group G with regular matrix P, with a relation $\mathbf{R} \subseteq S \times S$ that generates a non-universal congruence $\mathbf{R}^\sharp$. Let $\mathcal{S}_{\mathbf{R}^\sharp} = (\mathbf{R}|_I)^e$, let $\mathcal{T}_{\mathbf{R}^\sharp} = (\mathbf{R}|_\Lambda)^e$, and let $N_{\mathbf{R}^\sharp}$ be the least* *normal subgroup of G containing the set*

$$\begin{aligned}\Big\{(p_{\xi i}ap_{\lambda x})(p_{\xi j}bp_{\mu x})^{-1} \;\Big|\;& i,j,x \in I,\ \lambda,\mu,\xi \in \Lambda,\ a,b \in G \\ & \textit{such that } (i,a,\lambda)\ \mathbf{R}\ (j,b,\mu) \textit{ and } p_{\xi i}, p_{\lambda x} \neq 0\Big\} \\ \cup \Big\{q_{\lambda\mu ij} \;\Big|\;& (i,j) \in \mathbf{R}|_I,\ \lambda,\mu \in \Lambda \text{ such that } p_{\lambda i}, p_{\mu i} \neq 0\Big\} \\ \cup \Big\{q_{\lambda\mu ij} \;\Big|\;& (\lambda,\mu) \in \mathbf{R}|_\Lambda,\ i,j \in I \text{ such that } p_{\lambda i}, p_{\lambda j} \neq 0\Big\}.\end{aligned}$$

Then $(N_{\mathbf{R}^\sharp}, \mathcal{S}_{\mathbf{R}^\sharp}, \mathcal{T}_{\mathbf{R}^\sharp})$ is the linked triple corresponding to $\mathbf{R}^\sharp$.

This theorem is enough to show the correctness of our algorithm for converting a set of generating pairs to a linked triple – we present this algorithm here as Algorithm 3.21. In reading the algorithm, it will be helpful to refer to Definition 3.8 for the relations ε_I and ε_Λ and elements of the form $q_{\lambda\mu ij}$. The notation $\langle\langle N, x\rangle\rangle$ describes the least normal subgroup of G containing $N \cup \{x\}$ (see Definition 1.13); in particular, this is equal to N if $x \in N$. We will now give a brief description of how the algorithm operates, with reference to the pseudo-code in Algorithm 3.21. For a fuller description of the algorithm and how it is justified by Theorem 3.20, see [Tor14b, §3.2].

The LinkedTripleFromPairs algorithm starts with the minimal linked triple possible: $(N, \mathcal{S}, \mathcal{T}) = \big(\{1_G\}, \Delta_I, \Delta_\Lambda\big)$, where $\{1_G\}$ is the trivial subgroup of G, and Δ_I and Δ_Λ are the trivial equivalences on the sets of columns and rows in the matrix P. This is the linked triple that corresponds to the trivial congruence Δ_S, which is the result that should be returned if $\mathbf{R}$ is empty. The rest of the algorithm (lines 5–28) consist of going through each pair in $\mathbf{R}$, adding any necessary extra elements to N, $\mathcal{S}$ and $\mathcal{T}$ that are implied by that pair, and finally adding further elements to ensure that $(N, \mathcal{S}, \mathcal{T})$ remains a linked triple.

For each pair $(x, y) \in \mathbf{R}$, we first check whether $x = y$ (line 6) – if so, it is a pair in Δ_S and we do not need to do anything to the linked triple in order to account for it. If $x \neq y$, then we need to check whether one of x or y is equal to zero; if so, we have a non-zero element related to zero, which means that $\mathbf{R}^\sharp$ must be the universal congruence, which has no linked triple, and we quit the algorithm immediately returning this information (line 9).

If neither element is zero, then x and y must be non-zero elements, which we can rewrite as (i, a, λ) and (j, b, μ) (lines 10–11). Before we try modifying the linked triple, we check that columns i and j, and rows λ and μ, have zeroes in the same places in the matrix P (line 12); if not, they cannot be related by a linked triple, so we again have the universal congruence, and quit the algorithm immediately (line 13). Otherwise, we can proceed to add information to the triple, first by uniting the columns (i, j) in $\mathcal{S}$, and then by uniting the rows (λ, μ) in $\mathcal{T}$ (lines 15–16). This could perhaps be tracked using a union–find table for each of $\mathcal{S}$ and $\mathcal{T}$ (see Section 1.13).

Next we modify the normal subgroup N by adding an element based on the group elements a and b (lines 18–20), in line with the definition of Γ^{-1} in Definition 3.10. As we add this, we add any necessary elements to N to make it a normal subgroup (taking its *normal closure*). Finally, we add any necessary elements of the form $q_{\lambda\mu ij}$ to make N compatible with Definition 3.8 condition (iii) (lines 21–26). At the end of this, we have a triple $(N, \mathcal{S}, \mathcal{T})$ which is linked, and whose congruence contains every pair in $\mathbf{R}$. Since we added no elements but those required by $\mathbf{R}$ and the definition of a linked triple, we can be certain that $(N, \mathcal{S}, \mathcal{T})$ describes $\mathbf{R}^\sharp$, the least congruence containing all the pairs in $\mathbf{R}$. Again, see [Tor14b, §3.2] for a full justification of the algorithm.

Algorithm 3.21 The LinkedTripleFromPairs algorithm

```
Require: $\mathcal{M}^0[G; I, \Lambda; P]$ a finite Rees 0-matrix semigroup
Require: $G$ a group, $P$ a regular matrix
 1: procedure LinkedTripleFromPairs($\mathbf{R}$)
 2:     $N := \{1_G\}$
 3:     $\mathcal{S} := \Delta_I$
 4:     $\mathcal{T} := \Delta_\Lambda$
 5:     for $(x, y) \in \mathbf{R}$ do
 6:         if $x = y$ then
 7:             continue
 8:         else if $x = 0$ or $y = 0$ then
 9:             return Universal Congruence (no linked triple)
10:         Let $x = (i, a, \lambda)$
11:         Let $y = (j, b, \mu)$
12:         if $(i, j) \notin \varepsilon_I$ or $(\lambda, \mu) \notin \varepsilon_\Lambda$ then
13:             return Universal Congruence (no linked triple)
14:         ▷ Combine row and column classes
15:         $\mathcal{S} \leftarrow (\mathcal{S} \cup (i, j))^e$
16:         $\mathcal{T} \leftarrow (\mathcal{T} \cup (\lambda, \mu))^e$
17:         ▷ Add generators for normal subgroup
18:         Choose $\nu \in \Lambda$ such that $p_{\nu i} \neq 0$
19:         Choose $k \in I$ such that $p_{\lambda k} \neq 0$
20:         $N \leftarrow \langle\langle N, (p_{\nu i} a p_{\lambda k})(p_{\nu j} b p_{\mu k})^{-1} \rangle\rangle$
21:         for $\xi \in \Lambda \setminus \{\nu\}$ such that $p_{\xi i} \neq 0$ do
22:             $N \leftarrow \langle\langle N, q_{\nu\xi ij} \rangle\rangle$
23:             ▷ $N$ is a normal subgroup of $G$ containing every $q_{\nu\xi ij}$ considered so far
24:         for $z \in I \setminus \{k\}$ such that $p_{\lambda z} \neq 0$ do
25:             $N \leftarrow \langle\langle N, q_{\lambda\mu kz} \rangle\rangle$
26:             ▷ $N$ is a normal subgroup of $G$ containing every $q_{\lambda\mu kz}$ considered so far
27:         ▷ $(N, \mathcal{S}, \mathcal{T})$ is linked and its congruence contains every pair $(x, y)$ considered so far
28:     return $(N, \mathcal{S}, \mathcal{T})$
```

We can see the algorithm working in the following example.

Example 3.22. Let $S = \mathcal{M}^0[\mathcal{D}_4; \{1, 2, 3, 4\}, \{1, 2\}; P]$ be a Rees 0-matrix semigroup, where $\mathcal{D}_4$ is the permutation group $\langle (1\ 2\ 3\ 4), (2\ 4) \rangle$, isomorphic to the dihedral group on 4 points, and P is the 2×4 matrix

$$\begin{pmatrix} 0 & (1\ 2)(3\ 4) & 0 & (1\ 4\ 3\ 2) \\ (2\ 4) & (1\ 4)(2\ 3) & (2\ 4) & 0 \end{pmatrix}.$$

Let ρ be the congruence generated by the single pair $\big((1, (), 1), (3, (1\ 2\ 3\ 4), 1)\big)$. We can use LinkedTripleFromPairs to find the linked triple corresponding to ρ, or to determine that ρ is universal.

We set our triple to $\big(\{()\}, \Delta_4, \Delta_2\big)$ to start with, where Δ_n is the diagonal relation on $\{1, \dots, n\}$. Then we consider the single pair in the generating set: $x = (1, (), 1)$ and $y =$

$(3, (1\ 2\ 3\ 4), 1)$. We do not have $x = y$, $x = 0$ or $y = 0$, so we go on to consider the two elements componentwise. The pair $(i, j) = (1, 3)$ lies in ε_I since columns 1 and 3 contain zeroes in the same positions, and $(\lambda, \mu) = (1, 1)$ certainly lies in ε_Λ since it is a reflexive pair; hence we do not have to return the universal congruence. We modify $\mathcal{S}$ by joining the classes of 1 and 3 together; we do not have to modify $\mathcal{T}$, since $(1, 1)$ is already in $\mathcal{T}$. Finally we have to add generators to N: we can set both ν and k to 2, and then we add to N the element

$$\begin{aligned}(p_{\nu i} a p_{\lambda k})(p_{\nu j} b p_{\mu k})^{-1} &= \big(p_{21}()p_{12}\big)\big(p_{23}(1\ 2\ 3\ 4)p_{12}\big)^{-1} \\ &= \big((2\ 4)()(1\ 2)(3\ 4)\big)\big((2\ 4)(1\ 2\ 3\ 4)(1\ 2)(3\ 4)\big)^{-1} \\ &= (1\ 2\ 3\ 4),\end{aligned}$$

and take the normal closure. Finally we have to add any appropriate q values. There is no value of ξ which meets the stated requirements, but there is one appropriate value for x: $x = 4$. Hence we have to add the element $q_{\lambda\mu k x} = q_{1124} = p_{12} p_{12}^{-1} p_{14} p_{14}^{-1} = ()$. Since the identity already lies in N, we do not need to make any changes. There are no more pairs to process, so we return the linked triple $(N, \mathcal{S}, \mathcal{T}) = (\mathcal{C}_4, (1, 3)^e, \Delta_2)$, where

- $\mathcal{C}_4 = \langle (1\ 2\ 3\ 4) \rangle$ is the subgroup of $\mathcal{D}_4$ consisting of the four rotations, isomorphic to the cyclic group of order 4;
- $(1, 3)^e$ is the least equivalence on $\{1, 2, 3, 4\}$ containing the pair $(1, 3)$ (its classes are $\{1, 3\}$, $\{2\}$, and $\{4\}$);
- Δ_2 is the diagonal relation on $\{1, 2\}$ (its classes are $\{1\}$ and $\{2\}$).

3.2.4 Generating pairs from a linked triple

Let S be a completely simple or completely 0-simple semigroup, and let ρ be a non-universal congruence on S. In Section 3.2.3 we presented an algorithm to find the linked triple of ρ, given only a set of generating pairs for ρ. In this section, we will present the reverse: a method to find a set of generating pairs for ρ given only its linked triple $(N, \mathcal{S}, \mathcal{T})$.

Firstly we require a lemma describing the inclusion of congruences in each other, and how it mirrors an inclusion of linked triples.

Lemma 3.23 ([How95, Lemma 3.5.5], [Tor14b, Lemma 3.9])**.** *Let ρ and σ be non-universal congruences on S with linked triples $(N_\rho, \mathcal{S}_\rho, \mathcal{T}_\rho)$ and $(N_\sigma, \mathcal{S}_\sigma, \mathcal{T}_\sigma)$ respectively. We have $\rho \subseteq \sigma$ if and only if $N_\rho \leq N_\sigma$, $\mathcal{S}_\rho \subseteq \mathcal{S}_\sigma$, and $\mathcal{T}_\rho \subseteq \mathcal{T}_\sigma$.*

Now we can state the main theorem which will inform this algorithm. It also relies on ideas from Theorem 3.20.

Theorem 3.24. *Let $S = \mathcal{M}^0[G; I, \Lambda; P]$ be a finite 0-simple semigroup, and let ρ be a non-universal congruence with linked triple $(N_\rho, \mathcal{S}_\rho, \mathcal{T}_\rho)$. Let $N'_\rho \subseteq N_\rho$, $\mathcal{S}'_\rho \subseteq \mathcal{S}_\rho$ and $\mathcal{T}'_\rho \subseteq \mathcal{T}_\rho$ be any subsets with the following properties:*

- *N_ρ is the normal closure of N'_ρ in G,*
- *$\mathcal{S}_\rho = (\mathcal{S}'_\rho)^e$,*

- $\mathcal{T}_\rho = (\mathcal{T}'_\rho)^e$.

If $\mathbf{R}$ *is a subset of* ρ *such that*

(i) *for each pair* $(i,j) \in \mathcal{S}'_\rho$ *there exist* $\lambda, \mu \in \Lambda$ *and* $a, b \in G$ *such that* $(i,a,\lambda)\ \mathbf{R}\ (j,b,\mu)$*;*

(ii) *for each pair* $(\lambda,\mu) \in \mathcal{T}'_\rho$ *there exist* $i,j \in I$ *and* $a, b \in G$ *such that* $(i,a,\lambda)\ \mathbf{R}\ (j,b,\mu)$*;*

(iii) *for each element* $n \in N'_\rho$ *there exist* $i,j,x \in I$ *and* $\lambda,\mu,\xi \in \Lambda$ *such that* $p_{\xi i}$ *and* $p_{\lambda x}$ *are both non-zero and*
$$(i, p_{\xi i}^{-1} n p_{\lambda x}^{-1}, \lambda)\ \mathbf{R}\ (j, p_{\xi j}^{-1} p_{\mu x}^{-1}, \mu);$$

then $\mathbf{R}^\sharp = \rho$.

Proof. Assume $\mathbf{R}$ is as stated. Since ρ is a congruence and $\mathbf{R} \subseteq \rho$, we know that $\mathbf{R}^\sharp \subseteq \rho$. Hence we only need to show that $\rho \subseteq \mathbf{R}^\sharp$.

Let $(N_{\mathbf{R}^\sharp}, \mathcal{S}_{\mathbf{R}^\sharp}, \mathcal{T}_{\mathbf{R}^\sharp})$ denote the linked triple associated with $\mathbf{R}^\sharp$. We will show that $N_\rho \subseteq N_{\mathbf{R}^\sharp}$, $\mathcal{S}_\rho \subseteq \mathcal{S}_{\mathbf{R}^\sharp}$, and $\mathcal{T}_\rho \subseteq \mathcal{T}_{\mathbf{R}^\sharp}$, and therefore that $\rho \subseteq \mathbf{R}$ by Lemma 3.23.

Recall the relations $\mathbf{R}|_I$ and $\mathbf{R}|_\Lambda$ from Definition 3.19. By (i) we can see that $\mathcal{S}'_\rho \subseteq \mathbf{R}|_I$ and hence $(\mathcal{S}'_\rho)^e \subseteq (\mathbf{R}|_I)^e$. Meanwhile by Theorem 3.20 we have $(\mathbf{R}|_I)^e = \mathcal{S}_{\mathbf{R}^\sharp}$. In total this gives us $\mathcal{S}_\rho = (\mathcal{S}'_\rho)^e \subseteq (\mathbf{R}|_I)^e = \mathcal{S}_{\mathbf{R}^\sharp}$, so $\mathcal{S}_\rho \subseteq \mathcal{S}_{\mathbf{R}^\sharp}$. Similarly by (ii) we have $\mathcal{T}_\rho \subseteq \mathcal{T}_{\mathbf{R}^\sharp}$.

Now we turn our attention to N_ρ, and its generating set N'_ρ – we wish to show that $N_\rho \subseteq N_{\mathbf{R}^\sharp}$. Let $n \in N'_\rho$. By (iii), there exist $i,j,x \in I$ and $\lambda,\mu,\xi \in \Lambda$ such that $p_{\xi i}$ and $p_{\lambda x}$ are both non-zero and $(i,a,\lambda)\ \mathbf{R}\ (j,b,\mu)$, where
$$a = p_{\xi i}^{-1} n p_{\lambda x}^{-1} \qquad \text{and} \qquad b = p_{\xi j}^{-1} p_{\mu x}^{-1}.$$

Note that $p_{\xi j}$ and $p_{\mu x}$ must also be non-zero since $(i,j) \in \varepsilon_I$ and $(\lambda,\mu) \in \varepsilon_\Lambda$. To see that $n \in N_{\mathbf{R}^\sharp}$, observe that $p_{\xi i} a p_{\lambda x} = n$ and $p_{\xi j} b p_{\mu x} = 1_G$. Hence n satisfies the condition that
$$n = (p_{\xi i} a p_{\lambda x})(p_{\xi j} b p_{\mu x})^{-1}$$
for some $i,j,x \in I$, some $\lambda,\mu,\xi \in \Lambda$, and some $a,b \in G$ such that $(i,a,\lambda)\ \mathbf{R}\ (j,b,\mu)$ and $p_{\xi i}$ and $p_{\lambda x}$ are non-zero; this is precisely the requirement in Theorem 3.20 which means that $n \in N_{\mathbf{R}^\sharp}$. Hence $N'_\rho \subseteq N_{\mathbf{R}^\sharp}$. Since N_ρ is the normal closure of N'_ρ, and $N_{\mathbf{R}^\sharp}$ is a normal subgroup, we have $N_\rho \subseteq N_{\mathbf{R}^\sharp}$.

Since $N_\rho \subseteq N_{\mathbf{R}^\sharp}$, $\mathcal{S}_\rho \subseteq \mathcal{S}_{\mathbf{R}^\sharp}$ and $\mathcal{T}_\rho \subseteq \mathcal{T}_{\mathbf{R}^\sharp}$, Lemma 3.23 gives us $\rho \subseteq \mathbf{R}^\sharp$, as required. □

Theorem 3.24 is enough to justify the PAIRSFROMLINKEDTRIPLE algorithm, which is presented in this thesis as Algorithm 3.26. Given a linked triple $(N, \mathcal{S}, \mathcal{T})$, we only need to choose applicable subsets $N' \subseteq N$, $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{T}' \subseteq \mathcal{T}$ and we have a good idea of what pairs are necessary to generate a congruence. In the algorithm, we assume that a generating set N' is known for N (line 2) – this is certainly likely to be the case in a computational setting, for example in GAP [GAP18] where groups almost always have a known generating set. We should note that this set should act as a set of normal subgroup generators, meaning that it might be even smaller than a standard set of subgroup generators. For $\mathcal{S}'$ we use as few pairs as possible for each class of $\mathcal{S}$: for each class $C = \{i_1, \dots, i_n\}$ we include the pair (i_1, i_l) for all $l \in \{2, \dots, n\}$ (lines 3–8). These pairs relate all elements in the class to each other by transitivity (line 7), so we do not need to add any other elements. Hence each class requires a number of pairs in $\mathcal{S}'$ equal to one less than the size of the class; and so $|\mathcal{S}'| = |I| - k_{\mathcal{S}}$, where $k_{\mathcal{S}}$ is the number of classes in $\mathcal{S}$. We similarly choose as small a set as possible for $\mathcal{T}'$ (lines 9–14), so $|\mathcal{T}'| = |\Lambda| - k_{\mathcal{T}}$, where $k_{\mathcal{T}}$ is the number of classes in $\mathcal{T}$.

Once our three generating sets have been calculated, we collate them into the set of pairs $\mathbf{R}$ as efficiently as possible: each pair we add to $\mathbf{R}$ can satisfy the conditions in Theorem 3.24 for one pair $(i, j) \in \mathcal{S}'$, one pair $(\lambda, \mu) \in \mathcal{T}'$, and one element $n \in N'$. The while-loop on lines 19–30 steps through the three lists, on each iteration taking a new element a from N', a new pair of columns (i, j) from $\mathcal{S}'$, and a new pair of rows (λ, μ) from $\mathcal{T}'$. Then, after fixing appropriate ξ and k in lines 26–27 as in Theorem 3.24 condition (iii), we add the pair

$$\big((i, p_{\xi i}^{-1} a p_{\lambda k}^{-1}, \lambda), (j, p_{\xi j}^{-1} p_{\mu k}^{-1}, \mu)\big),$$

which can be seen by inspection to satisfy (i), (ii) and (iii) for the three generators in question.

On each iteration of the while-loop, we remove one item from each of N', $\mathcal{S}'$ and $\mathcal{T}'$ (lines 20–25), and we add one pair to $\mathbf{R}$. This is repeated until all three sets are exhausted, and so the total number of pairs returned by the algorithm is equal to the size of the largest of the three sets – that is,

$$\max(|N'|, |I| - k_{\mathcal{S}}, |\Lambda| - k_{\mathcal{T}}).$$

If the sets have different sizes, then on some of the later runs through the loop, one or two of the three sets will be empty. This does not present a problem: if a set is empty, it is not popped, and the last value from the loop is simply used again. The set of pairs still satisfies the necessary condition from the theorem. The only remaining case is that one of the sets may be empty to start with. To account for this case, we give each of the three variables a default value in lines 16–18, so that even if a set is never popped, there is a sensible default value that does not invalidate the condition: for a we can use the identity 1_G which must always be in N; for (i, j) or (λ, μ) we can use a reflexive pair, from Δ_I or Δ_Λ respectively.

It is natural to ask whether a set of generating pairs returned by PairsFromLinkedTriple is minimal – that is, to ask whether any smaller set of pairs could be found which generates the same congruence.

Theorem 3.25. *If $|N'| \leq |I| - k_{\mathcal{S}}$ or $|N'| \leq |\Lambda| - k_{\mathcal{T}}$, then* PairsFromLinkedTriple *returns a set of generating pairs which is minimal.*

Proof. The number of pairs returned by PairsFromLinkedTriple depends solely on the sizes of N', $\mathcal{S}'$ and $\mathcal{T}'$: it is simply the maximum of these three sizes. The generating set N' is assumed by the algorithm to have been known in advance, and hence is not guaranteed to be minimal in any way. However, $\mathcal{S}'$ and $\mathcal{T}'$ are created in the algorithm, each one consisting of a set of pairs in $I \times I$ or $\Lambda \times \Lambda$ which makes as few links as possible between elements in a class. In other words, $\mathcal{S}'$ and $\mathcal{T}'$ contain the smallest possible number of pairs such that $(\mathcal{S}')^e = \mathcal{S}$ and $(\mathcal{T}')^e = \mathcal{T}$. Note that $|\mathcal{S}'| = |I| - k_{\mathcal{S}}$ and $|\mathcal{T}'| = |\Lambda| - k_{\mathcal{T}}$.

Let $\mathbf{R}$ denote the output of the algorithm, and let $\mathbf{R}|_I$ be as in Definition 3.19. We can see from the definition of our algorithm that $\mathbf{R}|_I = \mathcal{S}'$, and Theorem 3.20 tells us that $\mathcal{S}_{\mathbf{R}^\sharp} = (\mathbf{R}|_I)^e$; that is, we require that $(\mathcal{S}')^e = \mathcal{S}$ for the algorithm for the output $\mathbf{R}$ to be valid. Since we have already seen that $\mathcal{S}'$ has as few pairs as possible such that $(\mathcal{S}')^e = \mathcal{S}$, we know that every pair in $\mathcal{S}'$ is necessary to produce a congruence with linked triple $(N, \mathcal{S}, \mathcal{T})$. So $|\mathcal{S}'|$ is a lower bound for the size of a set of generating pairs for the congruence. By similar reasoning, $|\mathcal{T}'|$ is also a lower bound.

Assume $|N'| \leq |I| - k_{\mathcal{S}}$ or $|N'| \leq |\Lambda| - k_{\mathcal{T}}$. The number of pairs returned by the algorithm will be either $|I| - k_{\mathcal{S}}$ or $|\Lambda| - k_{\mathcal{T}}$. Since we know that these are both lower bounds for the possible size of a generating set, we can conclude that the size of $\mathbf{R}$ equals the minimum possible size, so $\mathbf{R}$ is minimal. $\square$

In the case that N' is larger than both $|I| - k_{\mathcal{S}}$ and $|\Lambda| - k_{\mathcal{T}}$, a claim to minimality cannot be made so easily. Again referring to Theorem 3.20, we see that whereas $\mathcal{S}_{\mathbf{R}^\sharp}$ and $\mathcal{T}_{\mathbf{R}^\sharp}$ are determined entirely by the I and Λ parts of $\mathbf{R}$ respectively, the normal subgroup $N_{\mathbf{R}^\sharp}$ contains elements that may be implied by all three components of pairs in $\mathbf{R}$. Indeed, it may be that some elements in N' are in fact implied to be in N by some $q_{\lambda\mu ij}$, and so could be removed from N' without any loss. Identifying which elements are required in N' and which are not could be difficult computationally, but could be an interesting area of further research that would guarantee minimality in all cases. Note that, so long as N', $\mathcal{S}'$ and $\mathcal{T}'$ are all finite, the set of pairs is guaranteed to be finite, as is the number of steps in the algorithm. Hence we can be certain that the algorithm will terminate, since we can never enter an infinite loop.

We can see how the algorithm performs on the following example.

Example 3.27. Consider the semigroup $S = \mathcal{M}^0[\mathcal{S}_4; \{1,2,3\}, \{1,2,3,4\}; P]$ where $\mathcal{S}_4$ is the symmetric group of degree 4, and P is the 4×3 matrix

$$\begin{pmatrix} 0 & 0 & (1\ 2)(3\ 4) \\ (1\ 4) & () & 0 \\ (1\ 3\ 2\ 4) & (2\ 3\ 4) & 0 \\ 0 & (1\ 4\ 2) & 0 \end{pmatrix}.$$

This semigroup has 8 congruences: the universal congruence ∇_S, and 7 congruences defined by linked triples – this can be calculated slowly by hand, but much more quickly using the Semigroups package [M+19]. One such congruence is given by the linked triple $(\mathcal{A}_4, \Delta_3, (2,3)^e)$, where $\mathcal{A}_4$ is the alternating group of degree 4, Δ_3 is the diagonal relation on the column set $\{1,2,3\}$, and $(2,3)^e$ is the equivalence on the row set $\{1,2,3,4\}$ which only unites rows 2 and 3.

If we call $\textsc{PairsFromLinkedTriple}(\mathcal{A}_4, \Delta_3, (2,3)^e)$, the algorithm first produces the three generating components N', $\mathcal{S}'$ and $\mathcal{T}'$. The alternating group $\mathcal{A}_4$ can be generated by the set $\{(1\ 2\ 3), (2\ 3\ 4)\}$, so we may choose this to be our generating set N'; since Δ_3 is diagonal we produce $\mathcal{S}' = \varnothing$ and since only two rows are united by $(2,3)^e$ we produce $\mathcal{T}' = \{(2,3)\}$. Now we collate these three sets to make generating pairs for the congruence.

For the first pair, we have $a = (1\ 2\ 3)$, the default column values of $(i,j) = (1,1)$, and $(\lambda, \mu) = (2,3)$. We use the lowest possible values for ξ and k: $\xi = 2$ and $k = 1$. The pair we add is

$$\begin{aligned}\big((i, p_{\xi i}^{-1} a p_{\lambda k}^{-1}, \lambda), (j, p_{\xi j}^{-1} p_{\mu k}^{-1}, \mu)\big) &= \big((1, p_{21}^{-1}(1\ 2\ 3)p_{21}^{-1}, 2), (1, p_{21}^{-1} p_{31}^{-1}, 3)\big) \\ &= \big((1, (1\ 4)(1\ 2\ 3)(1\ 4), 2), (1, (1\ 4)(1\ 4\ 2\ 3), 3)\big) \\ &= \big((1, (2\ 3\ 4), 2), (1, (1\ 2\ 3), 3)\big).\end{aligned}$$

Algorithm 3.26 The PairsFromLinkedTriple algorithm

```
1: procedure PairsFromLinkedTriple($\mathcal{M}^0[G; I, \Lambda; P], (N, \mathcal{S}, \mathcal{T})$)
2:     $N'$ := normal subgroup generating set for $N$
3:     $\mathcal{S}' := \varnothing$
4:     for each non-singleton class $\{i_1, i_2, \ldots, i_n\}$ of $\mathcal{S}$ do
5:         for $l \in \{2, \ldots, n\}$ do
6:             Push $(i_1, i_l)$ onto $\mathcal{S}'$
7:             ▷ Columns $i_1, \ldots, i_l$ are all linked together through $i_1$
8:         ▷ $\{i_1, i_2, \ldots, i_n\}$ is a class of $(\mathcal{S}')^e$
9:     $\mathcal{T}' := \varnothing$
10:    for each non-singleton class $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ of $\mathcal{T}$ do
11:        for $l \in \{2, \ldots, n\}$ do
12:            Push $(\lambda_1, \lambda_l)$ onto $\mathcal{T}'$
13:            ▷ Rows $\lambda_1, \ldots, \lambda_l$ are all linked together through $\lambda_1$
14:        ▷ $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ is a class of $(\mathcal{T}')^e$
15:    $\mathbf{R} := \varnothing$
16:    $a := 1_G$
17:    $(i, j) := (1, 1)$
18:    $(\lambda, \mu) := (1, 1)$
19:    while $N' \neq \varnothing$ or $\mathcal{S}' \neq \varnothing$ or $\mathcal{T}' \neq \varnothing$ do
20:        if $N' \neq \varnothing$ then
21:            $a \leftarrow$ Pop($N'$)
22:        if $\mathcal{S}' \neq \varnothing$ then
23:            $(i, j) \leftarrow$ Pop($\mathcal{S}'$)
24:        if $\mathcal{T}' \neq \varnothing$ then
25:            $(\lambda, \mu) \leftarrow$ Pop($\mathcal{T}'$)
26:        Fix some $\xi \in \Lambda$ such that $p_{\xi i} \neq 0$
27:        Fix some $k \in I$ such that $p_{\lambda k} \neq 0$
28:        $\mathbf{R} \leftarrow \mathbf{R} \cup \left\{ \left( (i, p_{\xi i}^{-1} a p_{\lambda k}^{-1}, \lambda), (j, p_{\xi j}^{-1} p_{\mu k}^{-1}, \mu) \right) \right\}$
29:        ▷ $\mathbf{R}^\sharp$ is a subset of the congruence defined by $(N, \mathcal{S}, \mathcal{T})$
30:        ▷ If $(\bar{N}, \bar{\mathcal{S}}, \bar{\mathcal{T}})$ is the linked triple of $\mathbf{R}^\sharp$, then $a \in \bar{N}$, $(i, j) \in \bar{\mathcal{S}}$, and $(\lambda, \mu) \in \bar{\mathcal{T}}$
             for all $a, (i, j), (\lambda, \mu)$ popped so far
31:    return $\mathbf{R}$
```

For the second pair, we change a to the second generator $(2\ 3\ 4)$, and having exhausted both $\mathcal{S}'$ and $\mathcal{T}'$ we leave (i,j) and (λ,μ) unchanged. We can use the same values for ξ and k, so the next pair we add is

$$\begin{aligned}\big((1, p_{21}^{-1}(2\ 3\ 4)p_{21}^{-1}, 2), (1, p_{21}^{-1}p_{31}^{-1}, 3)\big) &= \big(\,(1, (1\ 4)(2\ 3\ 4)(1\ 4), 2)\,, (1, (1\ 4)(1\ 4\ 2\ 3), 3)\,\big)\\ &= \big(\,(1, (1\ 2\ 3), 2)\,, (1, (1\ 2\ 3), 3)\,\big).\end{aligned}$$

This exhausts N' as well, so have exhausted all three sets. We therefore return the set of two pairs,

$$\Big\{\big(\,(1, (2\ 3\ 4), 2)\,, (1, (1\ 2\ 3), 3)\,\big), \big(\,(1, (1\ 2\ 3), 2)\,, (1, (1\ 2\ 3), 3)\,\big)\Big\},$$

which is a valid generating set for the congruence.

3.2.5 Kernel and trace from generating pairs

Given a set of generating pairs $\mathbf{R}$ over a semigroup S, we may wish to consider the congruence $\rho = \mathbf{R}^\sharp$ and ask questions such as whether a pair lies in the congruence, or the number of congruence classes. This is certainly possible by various methods, for example the variety of algorithms mentioned in Chapter 2 – however, if S is an inverse semigroup then the congruence has an associated kernel–trace pair, as described in Section 3.1.4. If we know this kernel–trace pair, then we can use methods associated with it to carry out calculations, and benchmarking in [Tor14b, §6.1.3] indicates that these calculations are likely to be much faster than by using other methods. We therefore wish for an algorithm that determines the kernel and trace of ρ.

One way of calculating the kernel and trace would be simply to enumerate all the elements in all the classes of ρ, and to search for the idempotents to compute the kernel and trace. However, enumerating all the classes is very time-consuming, and the main reason to calculate the kernel–trace pair in the first place is probably to avoid this work. Hence, we want to find the kernel–trace pair directly from the generating pairs $\mathbf{R}$, enumerating as few pairs in $\mathbf{R}^\sharp$ as possible.

A new way of finding the kernel and trace directly from the generating pairs is presented in pseudo-code in Algorithm 3.28, which will require some explanation. It is based on a simple idea: firstly, populate K and τ with those elements that are implied directly by the pairs in $\mathbf{R}$; then, add further elements to K and τ to satisfy the conditions of a kernel–trace pair. This means we return the least kernel–trace pair (K,τ) that implies the pairs in $\mathbf{R}$ – that is, we return the kernel–trace pair that corresponds to $\mathbf{R}^\sharp$. This idea is explained more explicitly below.

To understand why the algorithm is correct, we make use of the following lemma, akin to Lemma 3.23 for linked triples.

Lemma 3.29. *Let ρ and σ be congruences on S with kernel–trace pairs (K_ρ, τ_ρ) and (K_σ, τ_σ) respectively. We have $\rho \subseteq \sigma$ if and only if $K_\rho \le K_\sigma$ and $\tau_\rho \subseteq \tau_\sigma$.*

Proof. Assume $K_\rho \le K_\sigma$ and $\tau_\rho \subseteq \tau_\sigma$, and let $(x,y) \in \rho$. By Theorem 3.17, we have $xy^{-1} \in K_\rho$ and $(x^{-1}x, y^{-1}y) \in \tau_\rho$. Hence $xy^{-1} \in K_\sigma$ and $(x^{-1}x, y^{-1}y) \in \tau_\sigma$, which together imply $(x,y) \in \sigma$. Hence $\rho \subseteq \sigma$.

Conversely, assume $\rho \subseteq \sigma$. If $k \in K_\rho$ then $k = xy^{-1}$ for some $(x,y) \in \rho$; this means $(x,y) \in \sigma$, so $k = xy^{-1} \in K_\sigma$. Similarly, if $(e,f) \in \tau_\rho$ then $(e,f) = (x^{-1}x, y^{-1}y)$ for some

Algorithm 3.28 The KerTraceFromPairs algorithm

```
Require: $S$ an inverse semigroup with idempotents $E$
Require: $\mathbf{R} \subseteq S \times S$
procedure KerTraceFromPairs($\mathbf{R}$)
    $K := E$
    $\tau := \Delta_E$
    Let $S'$ be a generating set for $S$
    Let $E'$ be a generating set for $E$
    $X \leftarrow \{ab^{-1} : (a,b) \in \mathbf{R}\}$
    $\mathbf{T} \leftarrow \{(a^{-1}a, b^{-1}b) : (a,b) \in \mathbf{R}\}$
    $\tau \leftarrow (\tau \cup \mathbf{T})^e$
    repeat
        $\delta \leftarrow$ false                                   ▷ Nothing has changed yet
        EnumerateKernel( )
        EnforceConditions( )
        EnumerateTrace( )
        ▷ $K \subseteq \ker \mathbf{R}^\sharp$ and $\tau \subseteq \operatorname{tr} \mathbf{R}^\sharp$
    until $\delta =$ false                                  ▷ Exit loop if nothing changed
    return $(K, \tau)$
procedure EnumerateKernel( )
    if $X \setminus K \neq \varnothing$ then
        $K \leftarrow \langle\langle K, X \rangle\rangle$
        $\delta \leftarrow$ true
    $X \leftarrow \varnothing$
procedure EnforceConditions( )
    for $a \in S$ do
        if $a \in K$ then
            if $(aa^{-1}, a^{-1}a) \notin \tau$ then
                $\mathbf{T} \leftarrow \mathbf{T} \cup \{(aa^{-1}, a^{-1}a)\}$
                $\tau \leftarrow \tau \cup \{(aa^{-1}, a^{-1}a)\}$
                $\delta \leftarrow$ true
        else
            for $e \in [a^{-1}a]_\tau$ do
                if $ae \in K$ then
                    $X \leftarrow X \cup \{a\}$
                    $\delta \leftarrow$ true
        ▷ (i) and (ii) from Definition 3.16 hold for each element $a \in S$ considered so far
procedure EnumerateTrace( )
    while $\mathbf{T} \neq \varnothing$ do
        Pick any $(x, y) \in \mathbf{T}$
        for $e \in E'$ do
            if $(xe, ye) \notin \tau$ then
                $\delta \leftarrow$ true
                $\mathbf{T} \leftarrow \mathbf{T} \cup \{(xe, ye)\}$
                $\tau \leftarrow (\tau \cup \{(xe, ye)\})^e$
                for $a \in S'$ do
                    if $(a^{-1}xea, a^{-1}yea) \notin \tau$ then
                        $\mathbf{T} \leftarrow \mathbf{T} \cup \{(a^{-1}xea, a^{-1}yea)\}$
                        $\tau \leftarrow (\tau \cup \{(a^{-1}xea, a^{-1}yea)\})^e$
                    ▷ $(a^{-1}xea, a^{-1}yea) \in \tau$
            ▷ $(xe, ye), (ex, ey) \in \tau$
        $\mathbf{T} \leftarrow \mathbf{T} \setminus \{(x, y)\}$
        ▷ $\tau$ satisfies Definition 3.15 for all $a \in S'$ and all $(x, y) \in \tau$ considered so far
    ▷ $\tau$ is a normal congruence on $E$ (Definition 3.15)
```

$(x, y) \in \rho$; this means $(x, y) \in \sigma$, so $(e, f) = (x^{-1}x, y^{-1}y) \in \tau_\sigma$. Hence $K_\rho \leq K_\sigma$ and $\tau_\rho \subseteq \tau_\sigma$, as required. □

The kernel K starts out containing just the idempotents E (line 2), and the trace τ starts out as the trivial congruence on E (line 3). Every kernel and trace must contain at least these elements – in fact, after line 3, (K, τ) corresponds to the trivial congruence Δ_S. We assume that we have generating sets S' for S and E' for E (lines 4–5). In the worst case, we can use S and E themselves, but the algorithm is likely to run faster with a smaller generating set. Certainly in computational settings such as the Semigroups package for GAP [M+19] semigroups such as S and E have a generating set stored, and a smaller generating set can sometimes be created by eliminating unnecessary elements.

Once these setup steps have been done, we add information from the known pairs of ρ – that is, from the pairs in $\mathbf{R}$. Theorem 3.17 tells us that a pair (a, b) lies in ρ if and only if $ab^{-1} \in K$ and $(a^{-1}a, b^{-1}b) \in \tau$. Now instead of using K and τ to determine whether a pair is in ρ, we are using a pair in ρ to impose conditions on K and τ. We have two sets, X and $\mathbf{T}$, which act as queues for elements that need to be processed in K and τ respectively. For each $(a, b) \in \mathbf{R}$ we put ab^{-1} into X (line 6) and $(a^{-1}a, b^{-1}b)$ into $\mathbf{T}$ (line 7); elements in X will be added to K next time we call EnumerateKernel, and we add $\mathbf{T}$ to τ straight away (line 8).

Once this has been done, the rule that (a, b) lies in ρ if and only if $ab^{-1} \in K$ and $(a^{-1}a, b^{-1}b) \in \tau$ is satisfied for all pairs $(a, b) \in \mathbf{R}$. All that is left to do is to add any elements to K and pairs to τ required to make (K, τ) a kernel–trace pair. The rest of the algorithm (lines 9–16 and the three sub-procedures) focuses on this task.

Recall from Definitions 3.14, 3.15 and 3.16 the conditions for a kernel–trace pair. We require (K, τ) to satisfy these conditions, and we must make any additions necessary until they are all fulfilled. For this purpose we have three sub-procedures – EnumerateKernel, EnumerateTrace, and EnforceConditions – that test the conditions for a kernel–trace pair and add any elements necessary. Any of these methods might add to K or τ, which might in turn imply that another method has more information to find. Hence, the three methods are run repeatedly until an entire run is completed in which no new information is found (δ remains false throughout the entire run). If no new information is found, (K, τ) is guaranteed to be a kernel–trace pair, and we can return. The three methods could be run in any order without the correctness of the algorithm being affected, but the order shown in Algorithm 3.28 seems to have the best time performance, based on informal experiments. All three methods are considered to have access to any of the variables in the overall algorithm.

The first method, EnumerateKernel, first checks whether there are any new elements in X that have not already been added to K (line 18). If there are, it adds all the elements from X to K, and then on line 19 it adds any necessary elements $a^{-1}xa$ to K, as in Definition 3.14, to ensure that K remains self-conjugate. This process of adding elements to ensure self-conjugacy is denoted with the notation $\langle\langle\cdot\rangle\rangle$, as for normal closure in a group. Since K contains E from the beginning, this is enough to guarantee that K is a normal subsemigroup, a fact we can be certain of at the end of EnumerateKernel. If a change was made, we set δ to true, and in any case we empty the set X to indicate that no new elements have been found since this sub-procedure was run.

The EnumerateTrace method ensures that τ is a normal congruence (see Definition 3.15).

It considers all the pairs that have been added to τ since the last call to EnumerateTrace – these are precisely the pairs in $\mathbf{T}$ – and makes sure that any pairs implied by them are added to τ and $\mathbf{T}$. For each $(x, y) \in \mathbf{T}$, the left and right multiples of (x, y) must be in τ (as required by the definition of a congruence). In fact, only the right-multiples (xe, ye) need to be added, since idempotents commute in an inverse semigroup, and the trace is only a relation on the idempotents. If any of these pairs are new, they are added to $\mathbf{T}$ so that further multiples can be found; this is why we only need to multiply by the generators from E', rather than all elements in E. So, in EnumerateTrace, we go through all the pairs (x, y) in $\mathbf{T}$ one at a time (lines 36–37) and apply each generator $e \in E'$ to its right-hand side to make the pair (xe, ye) (lines 38–39), which is equal to (ex, ey) by commutativity. For each one of these pairs that is not already in τ, we have to add it to τ to ensure that τ remains a congruence (line 42), and we have to add it to $\mathbf{T}$ (line 41) to ensure that we process all of its right-multiples in a future iteration of the while-loop. In order to ensure that τ is normal, we also need to conjugate the pair by each generator of the semigroup a, and add any of these to τ and $\mathbf{T}$ if they are not already present (lines 43–47). At the end of a call to EnumerateTrace, we can thus be sure that τ is a normal congruence (line 51). If any changes are made in this call to EnumerateTrace, then we must of course set δ to true (line 40).

Finally, EnforceConditions deals with conditions (i) and (ii) from Definition 3.16. It adds any necessary elements to X and any necessary pairs to $\mathbf{T}$ and τ, and when finished, (K, τ) is guaranteed to satisfy conditions (i) and (ii). To achieve this, we iterate through each element a in the semigroup (line 23). If a is in the kernel K (line 24), then we need to ensure that $(aa^{-1}, a^{-1}a)$ is in the trace τ to enforce condition (ii); we add it if necessary (lines 25–27). If $a \notin K$ but $ae \in K$ and $(e, a^{-1}a) \in \tau$ for some idempotent e, then we need to add a to the kernel in order to satisfy condition (i). Hence, if a is not in the kernel (line 29), we check any idempotents e that are τ-related to $a^{-1}a$ (line 30), and if ae is in the kernel (line 31) then we add a to the list X of elements to be added to the kernel in the next run of EnumerateKernel (line 32). If we make any changes to $\mathbf{T}$ or X in this procedure, we again set δ to true (lines 28 and 33). Lines 24–33 thus ensure that conditions (i) and (ii) hold for the particular value of a in question, hence the assertion on line 34.

If all three methods complete without any new information being found, they will have acted as a test ensuring that K is a normal subsemigroup of S, that τ is a normal congruence on E, and that the two conditions in Definition 3.16 are satisfied; in other words, that (K, τ) is a valid kernel–trace pair. This means that (K, τ) corresponds to a congruence $(K, \tau)\Psi^{-1}$, and we know that this congruence contains every pair in $\mathbf{R}$. Hence $\mathbf{R}^\sharp \subseteq (K, \tau)\Psi^{-1}$. Since we did not add any elements to K or τ except those implied by $\mathbf{R}$ or those required by the definition of a kernel–trace pair, we can also be sure that $(K, \tau)\Psi^{-1} \subseteq \mathbf{R}^\sharp$, by Lemma 3.29. Hence (K, τ) is the kernel–trace pair corresponding to the congruence $\mathbf{R}^\sharp$.

Note that, so long as S is finite, the KerTraceFromPairs algorithm is guaranteed to complete in finite time. Due to the way δ is set, the repeat-loop can only continue so long as the last iteration of the loop added something new to K or τ. Since K can only contain elements from S, and τ can only contain elements from $E \times E$, we therefore have an upper bound of $|K| + |E|^2$ times that the loop can be executed, and the likely number of times it will be executed is much lower. Similarly, there are no loops inside any of the sub-procedures that can run indefinitely. Hence the algorithm is certain to terminate and return an answer eventually.

3.2.6 Kernel and trace of a Rees congruence

Let S be an inverse semigroup with idempotents E. Each congruence on S is defined by its kernel and trace (see Section 3.1.4), and some congruences on S may be Rees (see Section 3.1.5). We may wish to find the kernel and trace of a Rees congruence given the ideal that defines it. Conversely, we may wish to determine whether a given kernel–trace pair on S describes a Rees congruence, and if so, what ideal it is associated with.

Let I be an ideal in S, and let ρ_I be $(I \times I) \cup \Delta_S$, the Rees congruence corresponding to I. To find the kernel and trace of ρ_I we must consider the positions of the idempotents in S, and how they interact with I.

Since I is an ideal, it must be a non-empty union of $\mathscr{J}$-classes, and since S is inverse, it has an idempotent in every $\mathscr{J}$-class; hence, there is at least one idempotent in I. The kernel of ρ_I is defined as the set of all elements that are related to an idempotent – that is, all elements in I and all idempotents outside I. Hence $\ker \rho_I = E \cup I$. The trace of ρ_I is defined as the restriction of ρ_I to the idempotents. Two distinct idempotents are ρ_I-related if and only if they both lie in I; hence $\operatorname{tr} \rho_I = ((E \cap I) \times (E \cap I)) \cup \Delta_E$.

Now we turn our attention to the other direction. Let ρ be a congruence defined by a kernel–trace pair (K, τ). How can we determine directly from K and τ whether ρ is a Rees congruence? We first prove a lemma, and then go on to answer this question.

Lemma 3.30. *Let x and y be elements of an inverse semigroup S. If xy^{-1} is idempotent and $x^{-1}x = y^{-1}y$, then $x = y$.*

Proof. We start by proving that $xy^{-1} = yy^{-1}$, and then go on to prove that $x = y$. By Proposition 1.5(i), we know that yy^{-1} is idempotent. We can left-multiply each of xy^{-1} and yy^{-1} by an element in S to give the other: $yx^{-1}(xy^{-1}) = y(x^{-1}x)y^{-1} = yy^{-1}yy^{-1} = yy^{-1}$ and $xy^{-1}(yy^{-1}) = xy^{-1}$. Hence $xy^{-1} \mathscr{L} yy^{-1}$. However, we know from Proposition 1.5(ii) that an inverse semigroup has only one idempotent in each $\mathscr{L}$-class. Since xy^{-1} and yy^{-1} are both idempotent, we must therefore conclude that $xy^{-1} = yy^{-1}$. Finally, we observe that

$$x = x(x^{-1}x) = x(y^{-1}y) = (xy^{-1})y = (yy^{-1})y = y,$$

and so we have $x = y$ as required. □

Theorem 3.31. *Let S be an inverse semigroup with set of idempotents E. If (K, τ) is a kernel–trace pair on S, then the congruence it defines is a Rees congruence if and only if the following hold:*

(i) *τ is a Rees congruence on E, with ideal denoted by I_τ;*

(ii) *$K = SI_\tau S \cup E$.*

Proof. Recall that $SXS = S^1XS^1$ for any set $X \subseteq S$, since $x(x^{-1}x) = (xx^{-1})x = x$ for any x in an inverse semigroup.

Let ρ be the congruence defined by (K, τ). We will first show that (i) and (ii) imply that ρ is Rees, and then we will show that ρ being Rees implies (i) and (ii).

First, assume that (i) and (ii) hold. Let $I = SI_\tau S$, so that $K = I \cup E$. This I is closed under left and right multiplication, so it is certainly an ideal of S. We will show that ρ is equal to the Rees congruence ρ_I, by showing $\rho_I \subseteq \rho$ and $\rho \subseteq \rho_I$.

For the first, let $(x, y) \in \rho_I$. If $x = y$ then $(x, y) \in \rho$ by reflexivity. Otherwise x and y are both in I, so $x = aeb$ and $y = cfd$ for some $e, f \in I_\tau$ and $a, b, c, d \in S$. Since I is an ideal and $x \in I$, we have $xy^{-1} \in I$ and therefore $xy^{-1} \in K$. Meanwhile we have $x^{-1}x = (aeb)^{-1}(aeb) = b^{-1}ea^{-1}aeb$: since $a^{-1}a$ and e are both idempotents, $ea^{-1}ae$ is an idempotent, and since $e \in I_\tau$ we also have $ea^{-1}ae \in I_\tau$; finally, since τ is a normal congruence (see Definition 3.15) we have $b^{-1}(ea^{-1}ae)b \in I_\tau$, and so $x^{-1}x \in I_\tau$. Similarly $y^{-1}y \in I_\tau$, so $(x^{-1}x, y^{-1}y) \in \tau$. Since $xy^{-1} \in K$ and $(x^{-1}x, y^{-1}y) \in \tau$, we have $(x, y) \in \rho$ by Theorem 3.17.

For the second, let $(x, y) \in \rho$. By Theorem 3.17 we have $xy^{-1} \in K = I \cup E$ and $(x^{-1}x, y^{-1}y) \in \tau$: either $x^{-1}x = y^{-1}y$, or $x^{-1}x, y^{-1}y \in I_\tau$. If $x^{-1}x = y^{-1}y$, then $x \mathscr{L} y$; we also have $(xy^{-1})y = xx^{-1}x = x$, so $x \mathscr{R} xy^{-1}$. Now, since $\mathscr{L} \subseteq \mathscr{J}$ and $\mathscr{R} \subseteq \mathscr{J}$, we know that x, y and xy^{-1} are all $\mathscr{J}$-related, so we have $x, y \in I$ in the case that $xy^{-1} \in I$. In the case that $xy^{-1} \in E$, we must have $x = y$ by Lemma 3.30. Either way, $(x, y) \in \rho_I$. Alternatively, if $x^{-1}x, y^{-1}y \in I_\tau$, then since $x \mathscr{J} x^{-1}x$ and $y \mathscr{J} y^{-1}y$, we have $x, y \in SI_\tau S$, that is $x, y \in I$ and so $(x, y) \in \rho_I$. So $(x, y) \in \rho$ implies $(x, y) \in \rho_I$, as required. Hence $\rho = \rho_I$, so ρ is Rees.

We now wish to show the converse, that ρ being Rees implies (i) and (ii). Assume ρ is a Rees congruence with ideal I, and let (K, τ) be its kernel–trace pair. The trace τ of ρ is the restriction of ρ to the idempotents E; this is easily seen to be a Rees congruence on E with ideal $I \cap E$. This gives us (i), where $I_\tau = I \cap E$. The kernel K of ρ is the set of elements ρ-congruent to an idempotent: this gives us $K = I \cup E$, since K consists of every element in the ideal I along with any other idempotents. Since any ideal is a union of $\mathscr{J}$-classes, and since any $\mathscr{J}$-class in an inverse semigroup contains an idempotent, we know that I is equal to $S(I \cap E)S$, which is equal to $SI_\tau S$. Hence $K = SI_\tau S \cup E$, and so we have (ii). $\square$

3.2.7 Trivial conversions

Some of the conversions between different representations are particularly trivial in nature, requiring almost no computational resources to calculate. However, it is worth mentioning them here for completeness.

Normal subgroups and kernel–trace pairs

All groups are inverse semigroups. Hence, if we have a congruence on a group, it can be represented by a normal subgroup or by a kernel–trace pair. Let ρ be such a congruence, on a group G: the classes of ρ are the cosets of some normal subgroup N. The kernel of ρ is defined as the set of elements which are ρ-related to an idempotent. Since there is only one idempotent – the identity 1_G – the kernel is all the elements in N. The trace of ρ is defined as the restriction of ρ to the idempotent; so $\operatorname{tr}\rho$ is just the trivial equivalence on the single element 1_G. Hence a congruence with normal subgroup N has kernel–trace pair $(N, \Delta_{\{1_G\}})$.

Normal subgroups and Rees congruences

A group G has precisely one Rees congruence: the universal congruence ρ_G. Its normal subgroup is the entire group G.

Linked triples and Rees congruences

A completely 0-simple semigroup $\mathcal{M}^0[G; I, \Lambda; P]$ (over a group G and regular matrix P) has two Rees congruences: the universal congruence and the trivial congruence. The universal congruence has no linked triple, while the trivial congruence corresponds to the linked triple $(\{1_G\}, \Delta_I, \Delta_\Lambda)$. A completely simple semigroup $\mathcal{M}[G; I, \Lambda; P]$ has only one Rees congruence: the universal congruence, which has linked triple $(G, \nabla_I, \nabla_\Lambda)$.

Linked triples and kernel–trace pairs

We may wish to convert between linked triples and kernel–trace pairs, in the case of an inverse semigroup which is completely simple or completely 0-simple. An inverse semigroup has exactly one idempotent in each $\mathscr{L}$-class and each $\mathscr{R}$-class, while a simple semigroup has just one $\mathscr{D}$-class and an idempotent in every $\mathscr{H}$-class. Hence a completely simple inverse semigroup has just one $\mathscr{H}$-class, and since it contains an idempotent it must be a group. Since it is a group, we can conclude that any congruence on a completely simple inverse semigroup has a linked triple of the form $(N, \Delta_I, \Delta_\Lambda)$ (see Section 3.2.1) which corresponds to the kernel–trace pair $(N, \Delta_{\{1_G\}})$ (see *Normal subgroups and kernel–trace pairs* in this section).

A completely 0-simple inverse semigroup is somewhat different, but also uncomplicated. Let S be such a semigroup, with idempotent set E. Since each $\mathscr{L}$-class and each $\mathscr{R}$-class has precisely one idempotent, the relations ε_I and ε_Λ are both trivial, so the non-universal congruences on S correspond to triples of the form $(N, \Delta_I, \Delta_\Lambda)$ for any normal subgroup $N \trianglelefteq G$. Now, the triviality of Δ_I and Δ_Λ implies that no two elements can be related by a non-universal congruence ρ unless they lie inside the same $\mathscr{H}$-class. Hence no two idempotents are related, so $\operatorname{tr}\rho = \Delta_E$. The kernel consists of all elements in S related to an idempotent. Idempotents are either 0 or have the form $(i, p_{\lambda i}^{-1}, \lambda)$ where $p_{\lambda i} \neq 0$, so non-zero elements in the kernel must have the form (i, a, λ) for $p_{\lambda i} \neq 0$. For $(i, a, \lambda)\ \rho\ (i, p_{\lambda i}^{-1}, \lambda)$ we just need $(p_{\xi i} a p_{\lambda x})(p_{\xi i} p_{\lambda i}^{-1} p_{\lambda x})^{-1} \in N$ for appropriate ξ and x as in Definition 3.10; but since each $\mathscr{L}$-class and $\mathscr{R}$-class contains just one idempotent, the only possible values are $x = i$ and $\xi = \lambda$. So the actual condition is $(p_{\lambda i} a p_{\lambda i})(p_{\lambda i} p_{\lambda i}^{-1} p_{\lambda i})^{-1} \in N$, which is the same as $p_{\lambda i} a p_{\lambda i} p_{\lambda i}^{-1} \in N$, or just $p_{\lambda i} a \in N$. Hence the kernel is given by

$$\ker\rho = \{(i, p_{\lambda i} n, \lambda) \in S \mid p_{\lambda i} \neq 0, n \in N\} \cup \{0\}.$$

Generating pairs and normal subgroups

If G is a group, then a congruence on G can be defined either by a set of generating pairs, or by a normal subgroup N. We may wish to convert from one of these representations to the other. We start with a proposition for converting a normal subgroup to a set of generating pairs.

Proposition 3.32. *Let G be a group, and N be a normal subgroup of G. If N' is a normal subgroup generating set for N, then $\{(1_G, n) : n \in N'\}^\sharp$ is the congruence on G defined by N.*

Proof. Let $\rho = \{(1_G, n) : n \in N'\}^\sharp$, and let ρ_N be the congruence whose classes are the cosets of N. Certainly ρ is a congruence on G, so by Theorem 3.2 we know that $[1_G]_\rho$, the ρ-class containing the identity, is a normal subgroup of G. This ρ-class contains all the elements in N', so we must have $N \subseteq [1_G]_\rho$; and since the pairs used to generate ρ were all from ρ_N, we must also have $[1_G]_\rho \subseteq N$. Hence the congruence classes of ρ are the cosets of N, as required. □

We also have a proposition for converting a set of generating pairs to a normal subgroup.

Proposition 3.33. *Let G be a group, and $\mathbf{R} \subseteq G \times G$. The normal subgroup generated by $\{xy^{-1} : (x, y) \in \mathbf{R}\}$ has cosets equal to the classes of the congruence $\mathbf{R}^\sharp$.*

Proof. Let $N' = \{xy^{-1} : (x, y) \in \mathbf{R}\}$, let $N = \langle\langle N' \rangle\rangle$, and let ρ_N be the congruence whose classes are the cosets of N. If $(x, y) \in \mathbf{R}$, then $xy^{-1} \in N$, which implies $Nx = Ny$, so that $(x, y) \in \rho_N$; hence $\mathbf{R} \subseteq \rho_N$. Since $\mathbf{R}^\sharp$ is the least possible congruence containing R, we must have $\mathbf{R}^\sharp \subseteq \rho_N$. Since N contains only elements that are required by $\mathbf{R}$ or by the definition of a normal subgroup, we must also have $\rho_N \subseteq \mathbf{R}^\sharp$. Hence $\rho_N = \mathbf{R}^\sharp$, as required. □

3.3 Further work

In this chapter we have given a survey of five different representations of congruences, and shown some algorithms to convert one to another without enumerating entire congruences. Table 3.1 shows these five representations, and gives references to various conversions between them. However, there are more areas of research which could be investigated, both in creating new representations, and in creating new algorithms to convert from one to another.

3.3.1 Generating pairs from a kernel–trace pair

Given an inverse semigroup S and a congruence ρ defined by a kernel–trace pair (K, τ), it is natural to wish for a set of generating pairs for ρ. There is not yet an algorithm to produce a set of generating pairs directly from a kernel–trace pair, but this would be an interesting area of future research.

A solution to this problem might follow the same structure as PairsFromLinkedTriple (Algorithm 3.26): break down the problem into a component for K and a component for τ, establishing a small set of elements which generate K as a normal subsemigroup of S, and a small set of pairs which generate τ as a normal congruence on E, and somehow combining these sets to find a set of generating pairs.

One could use the definitions of kernel and trace to produce a relatively straightforward algorithm. The kernel is the set of elements that are ρ-congruent to an idempotent. Hence, if K' is a generating set for K, then adding (k, e) for each $k \in K'$, where e is some idempotent such that $k \mathrel{\rho} e$, would ensure that the kernel contains K. The trace is the restriction of ρ to the idempotents; hence, if τ' is a relation such that $(\tau')^e = \tau$, adding all the pairs from τ' would ensure that the trace contains τ. This approach would result in a very large generating set, and could almost certainly be improved in some ways, particularly by exploiting (i) and (ii) in Definition 3.16 of a kernel–trace pair.

3.3.2 Rees congruences from generating pairs

Given a semigroup S and a set $\mathbf{R} \subseteq S \times S$, we may wish to know whether the generated congruence $\mathbf{R}^\sharp$ is a Rees congruence. A method exists for this in the Semigroups package [M+19].

It finds the congruence classes of $\mathbf{R}^\sharp$ and examines their sizes: if all classes are singletons, then $\mathbf{R}^\sharp$ is Rees if and only if a zero element exists, and if so, the ideal is $\{0\}$; otherwise, we check that there is only one non-trivial class, and if there is, then we check that it is an ideal. This method works, but of course involves enumerating the classes first.

It would be desirable to have an algorithm which can inspect the pairs in $\mathbf{R}$ and decide whether $\mathbf{R}^\sharp$ is Rees, while doing as little calculation of the congruence as possible. If it is somehow determined that $\mathbf{R}^\sharp$ is Rees, then it is the Rees congruence corresponding to the ideal

$$S^1\{x, y \in S : (x, y) \in \mathbf{R} \setminus \Delta_S\}S^1.$$

However, it may be that an answer cannot be determined without a large amount of work being done first. A more achievable aim would be to find some quick tests which could determine that $\mathbf{R}^\sharp$ is or is not Rees in limited cases. For example, if S is the Motzkin monoid $\mathcal{M}_n$, then $\mathbf{R}^\sharp$ is certainly Rees if any pair in $\mathbf{R} \setminus \Delta_S$ contains an element of rank greater than 1, by Theorem 5.23. Recognising many special cases like this would make it possible for a computer package to avoid enumerating certain congruences, which is desirable.

3.3.3 Regular semigroups

Recall that a regular semigroup is one in which every element x has an element x' such that $x = xx'x$. An inverse semigroup is a regular semigroup in which each element has a unique such element x^{-1}, with the additional requirement that $x^{-1}xx^{-1} = x^{-1}$. In Section 3.1.4 we discussed how a congruence on an inverse semigroup is uniquely determined by its kernel and trace, and gave both an abstract characterisation of a kernel–trace pair (Definition 3.16) and a concise description of how this pair describes its congruence (Ψ^{-1} in Theorem 3.17). It turns out that the congruences on a regular semigroup can be described in a similar way, which we will briefly examine here. First we will make a definition.

Definition 3.34 ([PP86, Result 1.5])**.** Let S be a regular semigroup, and let K be a subset of S. We define π_K as the relation on S containing all pairs $(a, b) \in S \times S$ such that

$$xay \in K \Leftrightarrow xby \in K$$

for all $x, y \in S^1$.

A congruence on a regular semigroup is uniquely determined by its kernel and trace [PP86, Corollary 2.11], and a kernel–trace pair can be characterised in the following way, analogous to Definition 3.16. Recall the definition of $\mathbf{E}^\flat$ (Definition 1.40).

Definition 3.35 ([PP86, Definition 2.12])**.** A **kernel–trace pair** on a regular semigroup S is a pair (K, τ) such that the following hold:

(i) $K \subseteq S$ and $K = \ker \pi_K$;

(ii) τ is an equivalence on E such that $\tau = \operatorname{tr}(\tau^\sharp)$;

(iii) $K \subseteq \ker(\mathscr{L}\tau\mathscr{L}\tau\mathscr{L} \cap \mathscr{R}\tau\mathscr{R}\tau\mathscr{R})^\flat$;

(iv) $\tau \subseteq \operatorname{tr} \pi_K$.

We even have a description of how a kernel–trace pair describes its congruence, analogue to Theorem 3.17.

Theorem 3.36 ([PP86, Theorem 2.13]). *Let S be a regular semigroup. There exists a bijection Ψ from the congruences on S to the kernel–trace pairs on S, defined by*

$$\Psi : \rho \mapsto (\ker \rho, \operatorname{tr} \rho),$$

and its inverse satisfies

$$\Psi^{-1} : (K, \tau) \mapsto \pi_K \cap (\mathscr{L}\tau\mathscr{L}\tau\mathscr{L} \cap \mathscr{R}\tau\mathscr{R}\tau\mathscr{R})^\flat.$$

This characterisation of congruences on regular semigroups falls short of its inverse semigroup counterpart. Firstly, this broader definition of a kernel–trace pair is a lot more complicated and harder to compute with: for example, calculating π_K for a subset K could be computationally difficult, as could verifying (iii) and (iv) in Definition 3.35. It is certainly difficult to contemplate any analogue of KerTraceFromPairs (Algorithm 3.28) which could find the least kernel–trace pair from a set of generating pairs in anything like as quick a time or as simple a procedure as in the inverse semigroup case. Secondly, the result in Theorem 3.36 is not as convenient as the inverse semigroup version (Theorem 3.17): if the kernel–trace pair of an inverse semigroup congruence is known, checking the presence of a given pair (x, y) is as simple as looking up one easily computed element in the kernel, and looking up another easily computed pair in the trace. In the regular semigroup case, checking whether a pair lies in $\pi_K \cap (\mathscr{L}\tau\mathscr{L}\tau\mathscr{L} \cap \mathscr{R}\tau\mathscr{R}\tau\mathscr{R})^\flat$ does not appear to be anything like as easy or quick.

For these reasons, using the kernel–trace approach for regular semigroups is not nearly as attractive as using it for inverse semigroups. However, it is possible that using the representation in a computational way would be feasible, and it is possible that in some cases it would be preferable to the naïve use of generating pairs. An algorithm to check the presence of a pair in $(K, \tau)\Psi^{-1}$ given a kernel–trace pair (K, τ) would be the first requirement; then a version of KerTraceFromPairs would be highly desirable, since it would allow us to use this approach even when the kernel and trace of a congruence are not known in advance, without enumerating the entire congruence first.

Chapter 4

Calculating congruence lattices

We can learn a lot about a semigroup's structure by examining its congruences: they describe a semigroup's homomorphic images, and quotient semigroups, as explained in Section 1.5. For this reason, it is of great interest to be able to produce a complete list of congruences on a given semigroup S. In this chapter, we present an algorithm to do this.

It is natural, when considering a problem in semigroup theory, to consider the approach we would take in the group case, and to see whether we can apply any of the ideas in this approach to semigroups generally. Hence, we will start by considering how to compute the congruence lattice of a group G.

In group theory, we study normal subgroups instead of studying congruences directly. As described in Section 3.1.2, a normal subgroup N of a group G has cosets equal to the classes of a congruence ρ_N, and we know that all congruences on G arise in this way. Furthermore, containment of normal subgroups corresponds to containment of congruences (i.e. $\rho_M \subseteq \rho_N \iff M \leq N$) so computing a group's congruence lattice is equivalent to computing the lattice of its normal subgroups.

Several algorithms exist for computing a group's normal subgroups. We will first describe a fairly naïve way to compute the normal subgroups, and then go on to outline the approach used in GAP. First, recall that a subgroup $H \leq G$ is normal if and only if $g^{-1}hg \in H$ for all $h \in H$ and $g \in G$.

A naïve way to compute the normal subgroups of a group G is by using its *conjugacy classes* – that is, the sets $C_x = \{g^{-1}xg : g \in G\}$ for all $x \in G$. We can see, from the definition of a normal subgroup given above, that a subgroup of G is normal if and only if it is a union of conjugacy classes. Hence, we can compute the conjugacy classes of G, and then take normal closures of their unions. All normal subgroups can be found in this way. This approach is guaranteed to complete for a finite group, but it is not particularly efficient: firstly, the conjugacy classes of G have to be computed, and then the taking of unions and normal closures are both likely to require a lot of work. Just computing the conjugacy classes may take up as much run-time as the rest of the algorithm, as shown in [Hul98, Table 1].

Next we mention a more sophisticated alternative, as used in GAP. The process is rather technical, and is not the main focus of this thesis, so we will only give an outline of the method here, referring the reader to [Hul98] for a fuller explanation. To compute the normal subgroups of a group G, we first compute a *chief series* for G – that is, a series of k normal subgroups of

G,

$$1 = N_k \subset N_{k-1} \subset \cdots \subset N_1 \subset N_0 = G,$$

such that there exists no normal subgroup $A \trianglelefteq G$ with $N_i \subset A \subset N_{i-1}$ for any $i \in \{1, \ldots, k\}$. Once such a chief series has been computed, the normal subgroups of G/N_i are computed inductively along the series: G/N_0 is trivial, and at each subsequent step we compute the normal subgroups of G/N_i using the normal subgroups of G/N_{i-1}, until on the last step we have the normal subgroups of $G/N_k = G$. This algorithm is a definite improvement on the naïve approach described above; indeed, tests summarised in [Hul98, Table 1] show that it is generally quicker to run this whole algorithm than to compute even just the conjugacy classes, the first step of the naïve method. This quick run-time includes the time taken to find a chief series of G, methods for which can be found in [CH97].

In examining these group algorithms, we hope to find ideas that can be extended to apply to semigroups generally. However, inspecting the two approaches described reveals nothing obvious which we can use. Firstly, we consider the naïve algorithm: the whole method is based on a normal subgroup being a union of conjugacy classes. Since a semigroup does not generally have inverses, the definition of conjugacy given above is not well-defined on a generic semigroup, meaning that a similar statement cannot be made that links the notion of conjugacy to the classes of a congruence. Several attempts have been made to extend the idea of conjugacy to semigroups in general [AKM14] but none of these has an obvious link to congruences. Hence, the first algorithm described cannot easily be extended to semigroup theory. Considering the second algorithm, there is also no concept of a chief series in semigroup theory. A related idea would be a chain of k congruences on S,

$$\Delta_S = \rho_k \subset \rho_{k-1} \subset \cdots \subset \rho_1 \subset \rho_0 = \nabla_S,$$

such that there exists no congruence ρ on G with $\rho_i \subset \rho \subset \rho_{i-1}$ for any $i \in \{1, \ldots, k\}$. However, it is not clear how such a series could be computed without doing as much work as it would take to compute all the congruences on S anyway. Furthermore, it is not clear how the congruences on S/ρ_i could be computed from the congruences on S/ρ_{i-1}, the obvious analogue of the inductive step described above; the bulk of [Hul98] describes how this step can be achieved in various cases, applying such concepts as group centre, conjugacy, and composition factors, all concepts which are not directly transferable to semigroup theory. Hence the second algorithm also cannot easily be converted.

In this chapter, we present a method for calculating all the congruences of a finite semigroup. This algorithm takes advantage of the fact that congruences lie in a lattice with respect to containment ($\subseteq$), intersection ($\cap$) and join ($\vee$). It computes the lattice structure while it computes the congruences themselves, and so the lattice structure is returned as an output of the algorithm, along with the set of congruences. This algorithm was used as a starting point for the work described in Chapter 5.

In Section 4.1 we give the algorithm in pseudo-code, and explain how it works. In Section 4.2 we outline some practical concerns for implementing the algorithm, with particular reference to how it is implemented in the Semigroups package [M$^+$19] for GAP [GAP18]. Finally, in Section 4.3, we present some examples of lattices which have been computed using this algorithm.

4.1 The algorithm

For the purposes of this section, we will make the following definition.

Definition 4.1. A **congruence poset** on a semigroup S is a pair $(\Gamma, \mathbf{O})$ where:

- Γ is a set of congruences on S; and
- $\mathbf{O}$ is $\subseteq$, the partial order of containment on Γ.

Recall that a partial order is defined as a relation that is reflexive ($x \leq x$), anti-symmetric ($x \leq y$ and $y \leq x$ if and only if $x = y$), and transitive ($x \leq y$ and $y \leq z$ implies $x \leq z$). Hence $\mathbf{O}$ will be a set of pairs of the form (ρ, σ), where ρ and σ are both congruences on S, and $\rho \subseteq \sigma$. If Γ is the set of all congruences on S, then $(\Gamma, \mathbf{O})$ will be a lattice by Proposition 1.29, and two congruences ρ and σ will have an intersection $\rho \cap \sigma$ and a join $\rho \vee \sigma$ in Γ. But note that in general, a congruence poset need not be closed under such operations.

4.1.1 Principal congruences

We first present an algorithm to calculate the principal congruences of a semigroup, along with their partial ordering $\subseteq$. This is a congruence poset, but since it may not contain all the congruences on the given semigroup, it may not be a lattice. We call this algorithm PrincCongPoset. Pseudo-code for is given for it in Algorithm 4.2, and it is discussed in more detail below.

Algorithm 4.2 The PrincCongPoset algorithm

```
Require: $S$ a finite semigroup
 1: procedure PrincCongPoset($S$)
 2:     $\Gamma := \varnothing$                                              ▷ Set of congruences
 3:     $\mathbf{O} := \varnothing$                                          ▷ Partial order ($\subseteq$) on congruences
 4:     for $(x, y) \in S \times S$ do
 5:         $P := \{((x,y)^\sharp, (x,y)^\sharp)\}$                          ▷ $(x,y)^\sharp \subseteq (x,y)^\sharp$
 6:         for $(a,b)^\sharp \in \Gamma$ do
 7:             if $(x,y) \in (a,b)^\sharp$ then
 8:                 if $(a,b) \in (x,y)^\sharp$ then
 9:                     goto line 4 and next pair $(x,y)$                   ▷ $(a,b)^\sharp = (x,y)^\sharp$
10:                 else
11:                     $P \leftarrow P \cup \{((x,y)^\sharp, (a,b)^\sharp)\}$   ▷ $(x,y)^\sharp \subseteq (a,b)^\sharp$
12:             else if $(a,b) \in (x,y)^\sharp$ then
13:                 $P \leftarrow P \cup \{((a,b)^\sharp, (x,y)^\sharp)\}$       ▷ $(a,b)^\sharp \subseteq (x,y)^\sharp$
14:         ▷ $(x,y)^\sharp \neq (a,b)^\sharp$ for each $(a,b)^\sharp$ considered so far
15:         $\Gamma \leftarrow \Gamma \cup \{(x,y)^\sharp\}$
16:         $\mathbf{O} \leftarrow \mathbf{O} \cup P$
17:         ▷ $\mathbf{O}$ is equal to the containment relation $\subseteq$ on $\Gamma$
18:     return $(\Gamma, \mathbf{O})$
```

The PrincCongPoset algorithm is not a very sophisticated algorithm, being based on a concept with a lot of brute-force work checking the presence of pairs in a congruence. However, when paired with the fast code in libsemigroups [MT$^+$18] for testing the presence of a pair in a single congruence (as described in Chapter 2) it can usually give results about small semigroups (say, up to size 400) in a reasonable amount of time (see Section 4.3).

The algorithm creates a set Γ of congruences on S, and a partial order $\mathbf{O}$ on Γ. By the end of the algorithm, Γ should contain every principal congruence on S, and $\mathbf{O}$ should be the partial order of containment $\subseteq$ on Γ. To find congruences, we go through each pair (x, y) in $S \times S$ (line 4), and consider the congruence $(x, y)^\sharp$ generated by that pair. We create a set P which will contain pairs that will be added to the partial order $\mathbf{O}$ if $(x, y)^\sharp$ is added to Γ; it initially contains $\big((x, y)^\sharp, (x, y)^\sharp\big)$, since any congruence contains itself with respect to $\subseteq$. In this way, since we go through every possible pair in $S \times S$, we certainly encounter every possible principal congruence at some point.

Starting on line 6, we compare the new congruence $(x, y)^\sharp$ to each of the congruences $(a, b)^\sharp$ that we have found and added to Γ so far. If the new congruence is equal to the old one, then we discard it (lines 8–9) and go on looking for more congruences. Note that this "goto" statement is necessary: lines 8–9 do not just avoid the else statement in lines 10–11, but actually discard the entire new congruence $(x, y)^\sharp$ and begin the next iteration of the outer for-loop on line 4. If the new congruence is strictly contained in the old congruence (line 10) or if the old is strictly contained in the new (line 12) we add a pair to P to show the containment. Once we have gone through all the old congruences in Γ, if we have not found one that is equal to $(x, y)^\sharp$, then we add $(x, y)^\sharp$ to Γ as a new congruence (line 15), and add the set of pairs P to $\mathbf{O}$ to describe how it contains or is contained in the other congruences (line 16). Since each new congruence is compared to every previously found congruence, every possible appropriate pair is added to $\mathbf{O}$, and we are therefore guaranteed that $\mathbf{O}$ will be equal to the containment relation $\subseteq$ on the set of congruences found so far (line 17). So long as S is finite, since both the loops in the algorithm are for-loops based on strictly finite sets, we are guaranteed that this algorithm will complete in a finite number of steps.

One positive outcome of using generating pairs in this way is that we can use the result

$$(a, b)^\sharp \subseteq (x, y)^\sharp \iff (a, b) \in (x, y)^\sharp$$

for any two pairs $(a, b), (x, y) \in S \times S$. Hence, in order to compare the two congruences comprehensively, we only need to test the presence of one pair in each congruence: $(a, b) \in (x, y)^\sharp$ and $(x, y) \in (a, b)^\sharp$. Testing the presence of a given pair in a congruence is likely to be faster than, for example, exhaustively computing its congruence classes. A general algorithm for testing whether a given pair lies in a congruence specified by generating pairs is described in Chapter 2; in some cases this can be improved by first converting the congruence to another representation, as described in Chapter 3.

Since this algorithm is based on iterating over all the pairs in $S \times S$, the time taken to compute the principal congruences increases rapidly as $|S|$ grows. This makes the algorithm ineffective for large semigroups. However, useful results can be obtained for small semigroups; see Section 4.3 for some examples.

Algorithm 4.2 shows a theoretical description of the PrincCongPoset algorithm, described in a fairly simple way to aid the understanding of the reader. However, it can be modified in a few simple ways to improve its performance. Firstly, we should consider the source of generating pairs: we iterate through all pairs $(x, y) \in S \times S$. There are ways in which this process is guaranteed to encounter a given congruence twice, and therefore waste time. For example, if we consider a pair (x, y), there is no need later to consider (y, x), since it will generate the same congruence. Similarly there is no need to consider every reflexive pair (x, x), since each one is guaranteed to generate the trivial congruence Δ_S; we can instead exclude reflexive pairs from the algorithm, and simply add Δ_S at the end, with an empty set of generating pairs. Thus, if S has n elements, we need only consider $\frac{1}{2}n(n-1)$ pairs, rather than all n^2 pairs from $S \times S$. In the best cases, this may reduce the runtime of the algorithm by more than 50%; however, note that the asymptotic complexity of the algorithm is not improved.

Note that we could also replace S here with some subset $X \subset S$, if we wish to see what congruences can be generated only with pairs from $X \times X$. For instance, we might be interested in congruences generated by pairs from some ideal of S, and how they affect elements outside the ideal. These questions can be answered with minimal changes to the algorithm.

Another possible improvement would be to use pairs already in $\mathbf{O}$, along with the axiom of transitivity, to skip certain comparisons. For example, if our new congruence $(x, y)^\sharp$ is found to be a subset of $(a, b)^\sharp$, but $(a, b)^\sharp$ is itself already known to be a subset of some congruence $(c, d)^\sharp$, then we can immediately add the pair $\big((x, y)^\sharp, (c, d)^\sharp\big)$ to P and we can skip the comparison of $(x, y)^\sharp$ to $(c, d)^\sharp$ later in the algorithm. Since most of the computational work in this algorithm tends to be in comparing congruences to each other, this ability to skip comparisons is important.

4.1.2 Adding joins

Our second algorithm is called JoinClosure. This algorithm takes a congruence poset $(\Gamma, \mathbf{O})$ as its argument, and returns the congruence poset containing all the congruences in Γ along with all their joins. That is, for any collection of k congruences $(\rho_i)_{1 \leq i \leq k}$ from Γ, the output of JoinClosure will contain the congruence

$$\bigvee_{1 \leq i \leq k} \rho_i \quad = \quad \rho_1 \vee \rho_2 \vee \ldots \vee \rho_k.$$

In order to calculate this, we can take advantage of one important property of all lattices: a lattice can be viewed as a semigroup in its own right. In particular, the set of congruences of a semigroup forms a semigroup under the operation $\vee$ of taking joins. Hence, finding the join-closure of a congruence poset $(\Gamma, \mathbf{O})$ is equivalent to finding the elements of the semigroup generated by Γ under $\vee$, along with information about how they multiply together.

There exist several algorithms which compute all the elements of a semigroup S using its generators X. An overview of such algorithms can be found in [EENMP18, §1]. A relatively naïve algorithm would simply create a list of elements, starting with the generators X, and multiply each element in the list with each generator, on the right and left, adding any new elements to the list and multiplying them in turn, until no new elements can be found. However, this algorithm would entail many unnecessary multiplications that could be avoided by using a more sophisticated algorithm. A better candidate is the Froidure–Pin algorithm which was mentioned in Section 2.5. This algorithm, first described in [FP97], takes a set of generators X for a semigroup S and returns, among other things, left and right Cayley graphs for S, and a list of words $w \in X^+$ representing one possible factorisation of each $s \in S$ in terms of the generators. Naturally, this algorithm only works when the multiplication of elements is well-defined and understood without knowledge of the semigroup as a whole; fortunately, this is the case for the join operation $\vee$ on congruences.

The outputs of the Froidure–Pin algorithm are sufficient to build up the entire congruence lattice of a semigroup, given the principal congruences. For each congruence ρ we have a word $w \in \Gamma^+$ representing how it is factorised in terms of the generators: this factorisation is precisely a list of principal congruences which need to be joined together to give ρ, which gives us a list of pairs from $S \times S$ which generates the congruence. The right Cayley graph returned by the algorithm describes the lattice structure in terms of joins ($\vee$), from which we can easily deduce the structure in terms of containment ($\subseteq$). Note that the left Cayley graph will be identical to the right Cayley graph, since the lattice is commutative. Again, a full description of the Froidure–Pin algorithm is outside the scope of this thesis, but it is described more completely in [FP97], and the version implemented in libsemigroups is explained in [JMP18].

It is sometimes preferable to use other methods when enumerating a semigroup. For example, the Semigroups package uses the method described in [EENMP18] to enumerate semigroups of transformations, partial permutations, matrices, and various other important classes, taking advantage of their Green's relations in order to avoid certain calculations. However, a lattice is known to be $\mathscr{D}$-trivial, meaning that the advantages of [EENMP18] do not apply to it. For this reason, the Froidure–Pin algorithm is likely to be a better choice.

The Froidure–Pin algorithm requires a method of deciding whether two congruences are equal. In JoinClosure, unlike in PrincCongPoset, we may encounter congruences with more than one generating pair. Hence, for two congruences ρ and σ, we cannot find out whether $\rho = \sigma$ in quite the same way as we did in PrincCongPoset. However, we have one useful result: if $\mathbf{R}$ and $\mathbf{S}$ are sets of generating pairs, then

$$\mathbf{R}^\sharp = \mathbf{S}^\sharp \quad \iff \quad \mathbf{R} \subseteq \mathbf{S}^\sharp \text{ and } \mathbf{S} \subseteq \mathbf{R}^\sharp,$$

so we only have to check containment of generating pairs in order to check equality of congruences. However, a congruence may have many generating pairs, so in some cases this check may take a long time. For this reason, if there is an alternative way of representing the congruences (for example, another representation from Chapter 3) then it may be quicker to use a containment method specific to that representation. For example, if S is a 0-simple semigroup, then our two congruences will have linked triples $(N_1, \mathcal{S}_1, \mathcal{T}_1)$ and $(N_2, \mathcal{S}_2, \mathcal{T}_2)$ respectively; instead of checking containment of generating pairs, we can check whether $N_1 = N_2$, $\mathcal{S}_1 = \mathcal{S}_2$ and $\mathcal{T}_1 = \mathcal{T}_2$.

Now that we have described the two algorithms, it is easy to see how we can use them to find the whole congruence lattice of a finite semigroup S. PrincCongPoset finds all the principal congruences of S, and JoinClosure finds all the joins of a poset of congruences. Since, in a finite semigroup, any congruence is the join of a finite number of principal congruences, we can produce the congruence lattice of S by simply calling

$$\textsc{JoinClosure}\big(\textsc{PrincCongPoset}(S)\big).$$

This is the basis of the function `LatticeOfCongruences` in the Semigroups package for GAP [M+19].

4.2 Implementation

So far, we have given a theoretical description of the PrincCongPoset and JoinClosure algorithms. As mentioned above, these correspond to functions implemented in the Semigroups package [M$^+$19] for GAP [GAP18]. PrincCongPoset is implemented approximately as described above, while JoinClosure currently uses a rather more rudimentary method than the Froidure–Pin method, something closer to the naïve method described earlier. In implementing these algorithms, we have to take into account various technical details which we might see as unimportant from a theoretical point of view. Some of these details are described below.

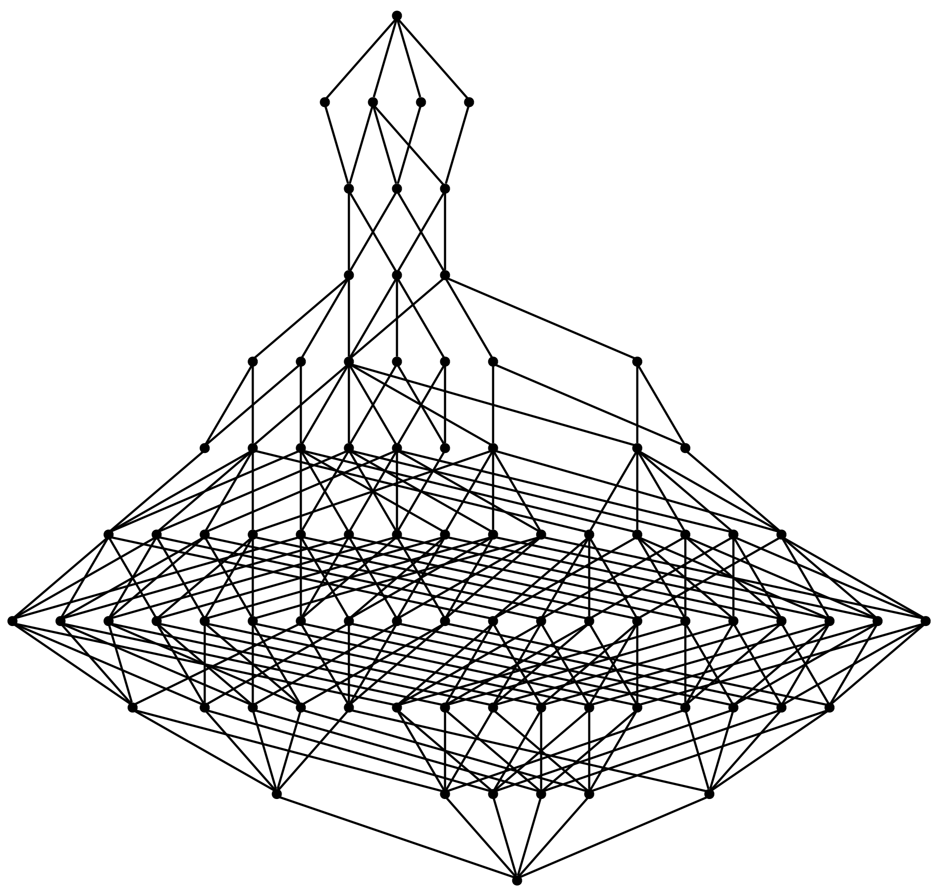
Firstly, let us consider how the partial order $\mathbf{O}$ is stored on a computer. This problem has certainly been considered before, and the solution we give here is not a new one, but is included for the interest and aid of anyone attempting to implement the algorithms described. A naïve approach would be simply to store all the pairs that are found in an array. This approach has the advantage of simplicity, and the advantage that the computational object is as close as possible to the mathematical object it describes. However, it has certain disadvantages that render it unattractive from a computational point of view – namely, that it is difficult to search for a given pair, and that it is difficult to find all the super-relations and sub-relations of a given congruence. Consider looking up whether a given pair (ρ, σ) is in an array of pairs: if the array is unsorted, this has complexity $O(n)$; even if the array is sorted, it has complexity $O(\log n)$. This complexity is similar to the problem, for a given ρ, of retrieving a list of all elements σ such that (ρ, σ) is in the array.

A better representation than a list of pairs is that of *adjacency lists* [BB08]. This method requires Γ to be stored with some order (which may be arbitrary). Instead of an array of pairs for $\mathbf{O}$, we have two lists of lists, `parents` and `children`, which store, respectively, a list of indices for all the congruences above each congruence, and a list of indices for all the congruences below each congruence, in the partial order $\mathbf{O}$. As an example, suppose we have a congruence ρ, and we want to know all the congruences which lie below ρ in the partial order $\mathbf{O}$. We look up the index i of ρ in the list Γ, and then the ith list in `children` contains all the indices of the congruences we want. If ρ_i and ρ_j are the congruences in Γ with indices i and j, we can find out whether $\rho_i \subseteq \rho_j$ by checking if $i \in$ `children`$[j]$ or $j \in$ `parents`$[i]$.

We mentioned above that the containment method ($\subseteq$) based on checking generating pairs can sometimes be improved by adopting a different congruence representation, for example using linked triples or kernel–trace pairs (see Chapter 3). In the Semigroups package, these different representations may be used automatically via GAP's method selection feature. When a congruence is created from a generating pair (x, y), the semigroup and the generating pair are supplied as arguments to a function `SemigroupCongruence`, which examines the properties of the semigroup, and determines what representation to use. For example, if the semigroup is known to be simple or 0-simple, `SemigroupCongruence` will compute the congruence's linked triple using the LinkedTripleFromPairs method (Algorithm 3.21) and use it instead of generating pairs wherever possible; similarly, if the semigroup is known to be inverse, then a kernel–trace pair will be computed using KerTraceFromPairs (Algorithm 3.28) and the congruence will be stored in that way. Since a congruence in PrincCongPoset is always generated by a single pair, we check containment as shown, by testing whether the given pair is in the congruence; but in JoinClosure, where the number of generating pairs could be much higher, GAP's method selection is used to choose a method for containment ($\subseteq$), generally preferring a method specific to the congruence representation in question.

4.3 Examples

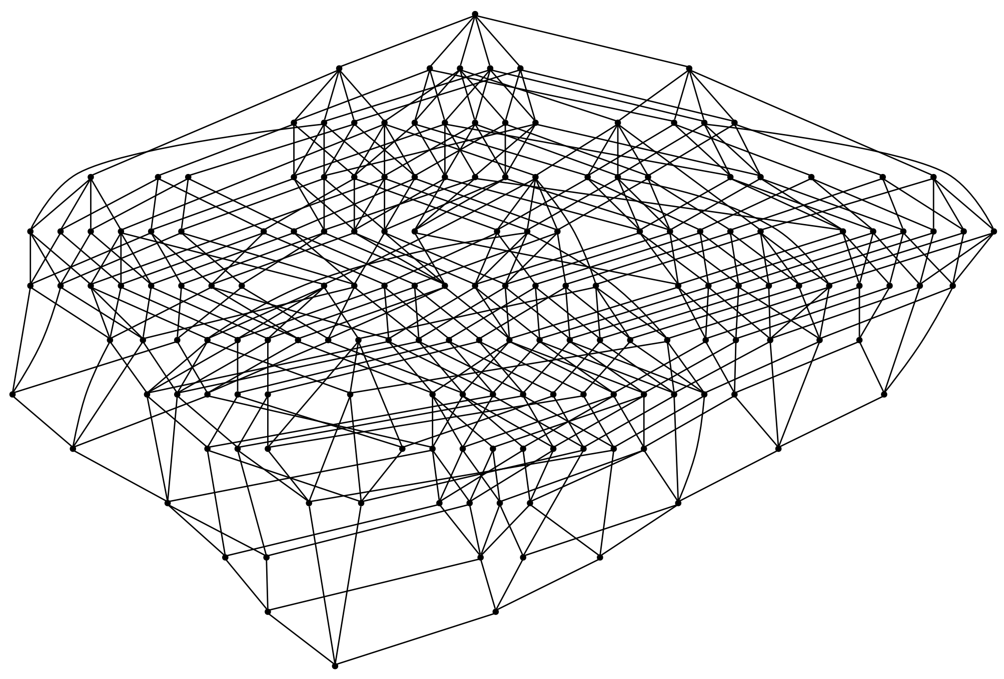
In this section we will show a few examples of congruence lattices that were computed in the Semigroups package [M+19] using the above algorithms. The output of the algorithm is shown in Figures 4.3, 4.4 and 4.5, and the code used to produce each lattice is shown underneath it.



```
gap> Splash(DotString(LatticeOfCongruences(GossipMonoid(3))));
```

Figure 4.3: Congruence lattice of the Gossip monoid G_3 as described in [FJK18, §2]. The semigroup contains 11 elements, and the lattice contains 84 congruences.

There are two main factors which determine how long `LatticeOfCongruences` takes to compute the lattice: the size of the semigroup S, and the number of congruences in the lattice Γ itself. Informal analysis shows that these two values do not necessarily go hand in hand. For instance, the monoids considered later in Section 5.4.2 show a variety of numbers of congruences which do not always correlate with the sizes of the semigroups. Even Figures 4.4 and 4.5



```
gap> Splash(DotString(LatticeOfCongruences(FullPBRMonoid(1))));
```

Figure 4.4: Congruence lattice of the full PBR monoid $\mathcal{PB}_1$ as described in [EENFM15, §2.1]. The semigroup contains 16 elements, and the lattice contains 167 congruences.

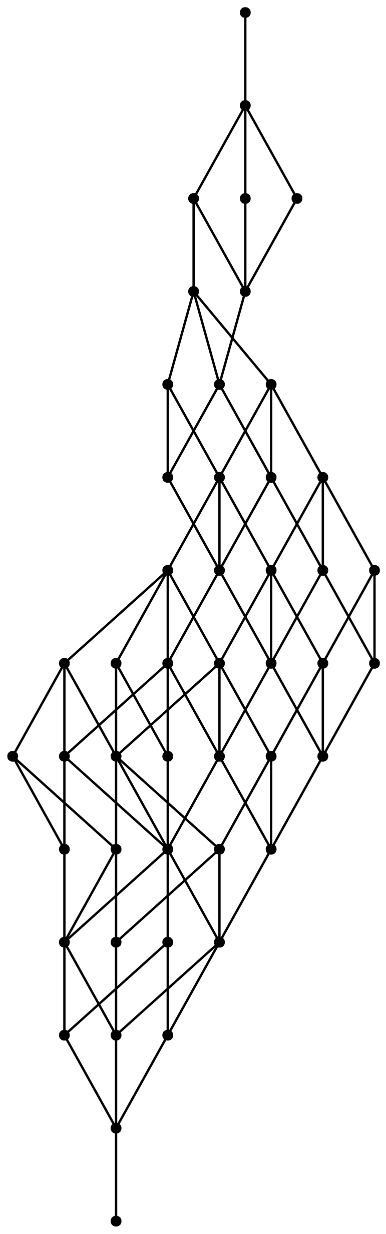
demonstrate between them that an increase in semigroup size need not indicate an increase in number of congruences.

In one test on an Intel Core i7-4770S CPU running at 3.10GHz with 16GB of memory, calculating the lattice of congruences of the wreath product $C_2 \wr \mathcal{T}_3$ (Figure 4.5) took 3140 ms, of which almost all the time (3019 ms) was consumed by PrincCongPoset. This is because the semigroup is relatively large (216 elements), and therefore iterating through all relevant pairs in $S \times S$ takes a long time; whereas the number of congruences is relatively small (only 47) meaning that the taking of joins does not take long. A contrasting example is the full PBR monoid $\mathcal{PB}_1$ (Figure 4.4): this took 5445 ms in total, of which almost all (5422 ms) was spent in JoinClosure. This is because the semigroup is relatively small (only 16 elements), so iterating through $S \times S$ is quick; but it has many congruences (167) meaning that it takes a long time to compute all the joins.

Since it is unknown in advance how many congruences a semigroup has, it is difficult to predict the feasibility of computing the lattice of a given semigroup, even if its size is known. Certainly all 853, 303 semigroups of size up to 7 have had their congruence lattices computed (see Section 6.3) with the aid of the smallsemi library [DM17], and tests on randomly generated transformation semigroups can usually calculate the lattice of a semigroup of size up to 400 in less than a minute (on the previously mentioned computer). However, we can choose very small examples in which JoinClosure runs for an unreasonable amount of time. Take, for example, the zero semigroup $\mathcal{Z}_{10}$, with only 10 elements. Calculating all its congruences using the method above does not complete within an hour, though computing the principal congruences takes only 16 milliseconds. This is because $\mathscr{Z}_{10}$ has a large number of congruences, given by the Bell number $B_{10} = 115975$, as will be shown in Theorem 6.16. An alternative method would work better here, since that theorem shows us that any equivalence on $\mathscr{Z}_{10}$ is a congruence.

In both parts of the algorithm, most of the work consists of comparing congruences to each other. These comparisons can be done relatively quickly by the efficient C++ code in libsemigroups for generating pairs (see Chapter 2), but minimising the number of comparisons that need to be made is nevertheless helpful for the algorithm's overall runtime. Hence it would be desirable, as future work, to improve the PrincCongPoset algorithm somehow to avoid unnecessary comparisons, as well as to implement the Froidure–Pin algorithm for JoinClosure in the Semigroups package.

Since the algorithm described above was implemented in the Semigroups package [M$^+$19], it has been possible to compute the congruence lattice of many semigroups. Part II of this thesis examines the congruence lattices of a variety of semigroups, and attempts to explain their structure. Many of these lattices were originally computed using PrincCongPoset and JoinClosure. After examining these lattices, it was possible in some cases to classify the congruences of entire infinite families of semigroups, with proofs that were independent of any computer code (see, for example, Theorems 5.23 and 6.4). In others it was possible at least to produce conjectures about families of semigroups, and to prove them for small cases (see Conjectures 6.17 and 6.18).



```
gap> C2 := Group((1, 2));;

gap> T3 := FullTransformationMonoid(3);;

gap> W := WreathProduct(C2, T3);;

gap> Splash(DotString(LatticeOfCongruences(W)));
```

Figure 4.5: Congruence lattice of the Wreath product $C_2 \wr \mathcal{T}_3$ as described in [Mel95, §10.1]. The semigroup contains 216 elements, and the lattice contains 47 congruences.

Part II

Theoretical applications

Chapter 5

Congruences of the Motzkin monoid

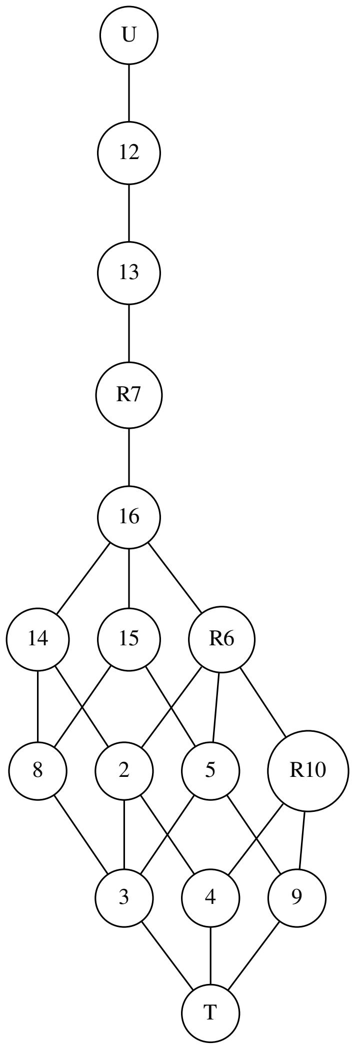
In Chapter 4 we explained a relatively quick way of computing all of a semigroup's congruences, along with information about how they fit into their lattice structure. This was implemented in the Semigroups package [M^+19], greatly increasing the size and complexity of semigroups whose congruence lattices can be found using a computer.

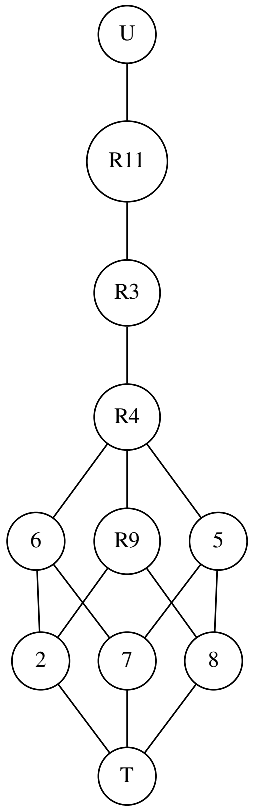
One of the first semigroups towards which this new methodology was directed was the bipartition monoid $\mathcal{P}_n$, whose congruence lattice was not previously known. Computing this lattice for the first few values of n showed a lattice with a relatively simple structure (see Figure 5.1) which did not appear to increase much in complexity as n grew higher than 3. The congruence lattices of various submonoids of $\mathcal{P}_n$ were also computed, and appeared to have a similar structure (again, see Figure 5.1).

With the rapidly increasing size of $\mathcal{P}_n$ (see Table 1.84) it proved impractical to naïvely calculate the congruence lattices beyond $n = 4$, but careful study of the lattices for small values of n, along with those lattices computed for various submonoids of $\mathcal{P}_n$, yielded a general classification of the congruence lattice of $\mathcal{P}_n$ for arbitrary n (see Figure 5.40), along with a classification of the congruence lattices of various important submonoids. This classification is explained and proven in [EMRT18], the paper upon which this chapter is based. In this chapter, we will examine the structure of these congruence lattices.

As an author of [EMRT18], my particular focus was the Motzkin monoid $\mathcal{M}_n$, which will be defined below. The other authors on the paper used my code for computing congruence lattices, as presented in Chapter 4, to study the congruences of $\mathcal{P}_n$, and they produced a classification of its congruences. I then modified and extended this work to classify the congruences of the Motzkin monoid, and helped with general tasks towards completing the paper. As such, this chapter focuses on the Motzkin monoid, only presenting the results for $\mathcal{P}_n$ and other monoids at the end. Many of the results we describe here are contained in some form in [EMRT18], and are included in this thesis with the permission of my co-authors.

We will start with the definition of the Motzkin monoid $\mathcal{M}_n$, then describe some preliminary theory, then describe the lattice of congruences of $\mathcal{M}_n$ (Theorem 5.23), and finally give a brief description of how these ideas can be extended to $\mathcal{P}_n$ and its other submonoids (Section 5.4).





```
gap> Splash(DotString(LatticeOfCongruences(PartitionMonoid(3)),
>                     rec(info:=true)));
gap> Splash(DotString(LatticeOfCongruences(MotzkinMonoid(4)),
>                     rec(info:=true)));
```

Figure 5.1: Congruence lattices of $\mathcal{P}_3$ (left) and $\mathcal{M}_4$ (right), as produced and displayed by the Semigroups package for GAP. Here 'T' represents the trivial congruence, 'U' the universal congruence, and 'R' a Rees congruence. Figures 5.40 and 5.24 illustrate these lattices with more meaningful labels.

5.1 The Motzkin monoid $\mathcal{M}_n$

In order to define the Motzkin monoid, we must first define a *planar* bipartition.

Definition 5.2. A bipartition of degree n is called **planar** if, when represented in diagram form (see Section 1.11.4), with the points $\{1, \dots, n\}$ in left-to-right order forming the top of a rectangle, and the points $\{1', \dots, n'\}$ in left-to-right order forming the bottom of the rectangle, with all edges contained inside the rectangle, it can be drawn without any edges crossing.

Example 5.3. Let $\alpha = \left[\begin{array}{c|c|c|c} 1,2 & 3 & 4 & 5 \\ 2,5 & 1 & 3,4 & \end{array}\right]$ and $\beta = \left[\begin{array}{c|c|c|c} 2 & 5 & 1,3 & 4 \\ 1 & 3,4 & 2 & 5 \end{array}\right]$. As can be seen in Figure 5.4, α is planar. However, β cannot be drawn inside the rectangle without the upper block $\{1, 3\}$ crossing lines with the transversal $\{2, 1'\}$ – hence, β is not planar.

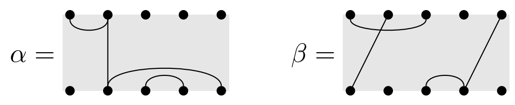


Figure 5.4: A planar and a non-planar bipartition.

We can now define the Motzkin monoid.

Definition 5.5. The **Motzkin monoid** $\mathcal{M}_n$ is the submonoid of $\mathcal{P}_n$ consisting of all planar bipartitions of degree n in which every block has size 1 or 2.

To see that this is indeed a monoid, we should observe that it is closed. It is easy to see that the product of two planar bipartitions is also planar, since a double diagram as in Figure 1.76 would contain no crossing lines, and therefore would resolve to a product with no crossing lines. It is also easy to see that if two bipartitions have no block larger than 2, their product also has no block larger than 2: any transversal can only contain one point in $\mathbf{n}$ and one point in $\mathbf{n}'$, so any transversal in the product can only contain two points. The upper and lower blocks of the product are inherited from the original bipartitions, so they will not break the condition either.

The Motzkin monoid $\mathcal{M}_n$ grows much slower than its parent $\mathcal{P}_n$, having only $\sum_{k=0}^{n} \binom{2n}{2k} C_k$ elements [OEIS, A026945], where C_k is the kth Catalan number. Its size in comparison with $\mathcal{P}_n$ is shown in Table 5.6.

n	$\lvert\mathcal{M}_n\rvert$	$\lvert\mathcal{P}_n\rvert$
1	2	2
2	9	15
3	51	203
4	323	4 140
5	2 188	115 975
6	15 511	4 213 597
7	113 634	190 899 322
8	853 467	10 480 142 147
9	6 536 382	682 076 806 159
10	50 852 019	51 724 158 235 372

Table 5.6: Sizes of $\mathcal{M}_n$ and $\mathcal{P}_n$ for small values of n.

The Motzkin monoid $\mathcal{M}_n$ shares a number of features with $\mathcal{P}_n$ – indeed, we will see later that its congruence lattice is very similar. Like $\mathcal{P}_n$, $\mathcal{M}_n$ is regular with a possible inverse given by the * function. Another important similarity is in its Green's relations: consider the following proposition, akin to Proposition 1.83.

Proposition 5.7. *Let α and β be bipartitions in $\mathcal{M}_n$. The following hold:*

(i) *$\alpha \mathrel{\mathscr{R}} \beta$ if and only if $\operatorname{dom}\alpha = \operatorname{dom}\beta$ and $\ker\alpha = \ker\beta$;*

(ii) *$\alpha \mathrel{\mathscr{L}} \beta$ if and only if $\operatorname{codom}\alpha = \operatorname{codom}\beta$ and $\operatorname{coker}\alpha = \operatorname{coker}\beta$;*

(iii) *$\alpha \mathrel{\mathscr{J}} \beta$ if and only if $\operatorname{rank}\alpha = \operatorname{rank}\beta$;*

(iv) *$J_\alpha \leq J_\beta$ if and only if $\operatorname{rank}\alpha \leq \operatorname{rank}\beta$;*

(v) *the ideals of $\mathcal{M}_n$ are precisely the sets $I_k = \{\alpha \in \mathcal{M}_n : \operatorname{rank}\alpha \leq k\}$ for $k \in \{0, \ldots, n\}$.*

Proof. For (i) to (iii), see [DEG17, Theorem 2.4]. For (iv) and (v), see [DEG17, Proposition 2.6]. □

This description of the Motzkin monoid's Green's relations, and its containment of $\mathscr{J}$-classes and ideals, will help us greatly later on. However, one consequence of (i) and (ii) gives $\mathcal{M}_n$ a feature which $\mathcal{P}_n$ does not share, namely the following corollary.

Corollary 5.8. *The Motzkin monoid $\mathcal{M}_n$ is $\mathscr{H}$-trivial.*

Proof. Let $\alpha, \beta \in \mathcal{M}_n$ such that $\alpha \mathrel{\mathscr{H}} \beta$. This tells us that $\alpha \mathrel{\mathscr{L}} \beta$ and $\alpha \mathrel{\mathscr{R}} \beta$, so by Proposition 5.7 parts (i) and (ii), we know that α and β share the same domain, kernel, codomain and cokernel. The upper blocks and lower blocks of α and β must certainly be the same, since they are just the blocks of the kernel and cokernel that do not lie in the domain or codomain. The only choice is in the transversals: which blocks in the domain connect to which blocks in the codomain. In $\mathcal{P}_n$ there are $(\operatorname{rank}\alpha)!$ ways of choosing this match-up; but in $\mathcal{M}_n$ there is only one way possible, since we cannot allow any lines in the diagram to cross. Hence $\alpha = \beta$. □

Finally, we will state one other feature of $\mathcal{M}_n$ that distinguishes it from $\mathcal{P}_n$: an interesting property of its minimal ideal I_0.

Lemma 5.9. *Let α and β be bipartitions in $\mathcal{M}_n$, with α in the minimal ideal I_0. Then $\alpha\beta\alpha = \alpha$.*

Proof. Since α has no transversals, $\alpha\beta\alpha$ also has no transversals. The upper blocks of a product are equal to those of its first factor, the lower blocks to those of its last factor – so $\alpha\beta\alpha$ has the upper and lower blocks of α. Hence it equals α. □

The property described in the previous lemma implies that I_0 is a *rectangular band* (see Definition 1.59).

5.2 Lifted congruences

We will now define some concepts which allow us to find certain congruences in any semigroup: *retractable ideals* (Definition 5.10) and *liftable congruences* (Definition 5.11). These constructions are new, first appearing in [EMRT18] to help describe some of the congruences on $\mathcal{P}_n$ and its submonoids. It will turn out that all non-Rees congruences of $\mathcal{M}_n$ can be found using these two building blocks.

Definition 5.10. Let S be a finite semigroup, with minimal ideal M. An ideal I of S is called **retractable** if there exists some homomorphism $\phi : I \to M$ such that $(m)\phi = m$ for all $m \in M$; we call ϕ a **retraction**.

Definition 5.11. Let S be a finite semigroup, with minimal ideal M. A congruence σ on M is a **liftable congruence** of S if either, and therefore both, of the following equivalent conditions are satisfied:

(i) $\sigma \cup \Delta_S$ is a congruence on S;

(ii) $(ax, bx), (xa, xb) \in \sigma$ for all pairs $(a, b) \in \sigma$ and elements $x \in S$.

To see that the two conditions in the last definition are equivalent, assume we have S, M and σ such that (i) is satisfied. Now let $(a, b) \in \sigma$ and $x \in S$ be arbitrary. Since $\sigma \cup \Delta_S$ is a congruence and $(a, b) \in \sigma \cup \Delta_S$, we must have $(ax, bx) \in \sigma \cup \Delta_S$. If $(ax, bx) \in \Delta_S$ then $ax = bx$, and since M is an ideal we must have both ax and bx in M; hence (ax, bx) is a reflexive pair and lies in the congruence σ. If $(ax, bx) \notin \Delta_S$ then $(ax, bx) \in \sigma$. Hence, either way, (ax, bx) is in σ, and by a similar argument, so is (xa, xb), so we have (ii).

Conversely, assume that (ii) holds, let $(a, b) \in \sigma \cup \Delta_S$, and let $x \in S$. If $(a, b) \in \Delta_S$ then $a = b$, and so $ax = bx$ and $xa = xb$. Otherwise, $(a, b) \in \sigma$ and by (ii) we have $(ax, bx), (xa, xb) \in \sigma$. In either case, we have $(ax, bx), (xa, xb) \in \sigma \cup \Delta_S$, and so we have (i).

In order to use these building blocks to produce new congruences, we first need to establish some results about them. Note that, since $\mathcal{M}_n$ is finite, it must have a minimal ideal. More specifically, the minimal ideal of $\mathcal{M}_n$ is given by $I_0 = \{\alpha \in \mathcal{M}_n : \operatorname{rank} \alpha = 0\}$ (see Proposition 5.7). The following lemma will be used at various times throughout this chapter.

Lemma 5.12. *Let S be a finite semigroup with minimal ideal M, and let I be an ideal of S. If I is retractable and ϕ is a retraction from I to M, then $(sxt)\phi = s \cdot (x)\phi \cdot t$ for all elements $x \in I$ and all $s, t \in S^1$.*

Proof. Since S is a finite semigroup, we know that its minimal ideal M is regular, by [How95, Proposition 3.1.4]. Hence any element $m \in M$ has an element $m' \in M$ such that $mm'm = m$. Since $(mm')m = m$ we have a left identity for m; and since $m(m'm) = m$, we also have a right identity. Let e be a right identity for $(x)\phi$, so that $(x)\phi \cdot e = (x)\phi$. Since ϕ is a retraction and $e, xe \in M$, we have

$$(x)\phi = (x)\phi \cdot e = (x)\phi \cdot (e)\phi = (xe)\phi = xe,$$

so $(x)\phi = xe$. Now let f be a left identity for $(sx)\phi$; we also have

$$\begin{aligned}(sx)\phi \cdot e &= f \cdot (sx)\phi \cdot e\\ &= (f)\phi \cdot (sx)\phi \cdot e\\ &= (fsx)\phi \cdot e\\ &= (fs)\phi \cdot (x)\phi \cdot e\\ &= (fs)\phi \cdot (x)\phi\\ &= (fsx)\phi\\ &= (f)\phi \cdot (sx)\phi\\ &= f \cdot (sx)\phi\\ &= (sx)\phi,\end{aligned}$$

which shows that e is a right identity for $(sx)\phi$ as well as for $(x)\phi$. Hence we have

$$s \cdot (x)\phi = s \cdot xe = (sxe)\phi = (sx)\phi \cdot (e)\phi = (sx)\phi \cdot e = (sx)\phi,$$

i.e. ϕ respects left multiplication; a symmetric argument gives $(xt)\phi = (x)\phi \cdot t$, i.e. ϕ respects right multiplication too. Finally we can combine these to give $(sxt)\phi = (sx)\phi \cdot t = s \cdot (x)\phi \cdot t$, as required. □

The previous lemma gives rise to an important corollary which we can use later when we combine retractable ideals with liftable congruences.

Corollary 5.13. *Let S be a finite semigroup, with minimal ideal M. If I is a retractable ideal of S, then the retraction $\phi : I \to M$ is unique.*

Proof. Let ϕ and ψ be retractions from I to M. Let $x \in I$, let e_l be a left identity for $(x)\phi$, and let e_r be a right identity for $(x)\psi$. By Lemma 5.12, we have

$$(x)\phi = e_l \cdot (x)\phi = (e_l x)\phi = e_l x = (e_l x)\psi = e_l \cdot (x)\psi,$$

so $(x)\phi = e_l \cdot (x)\psi$. Similarly,

$$(x)\psi = (x)\psi \cdot e_r = (xe_r)\psi = xe_r = (xe_r)\phi = (x)\phi \cdot e_r,$$

so $(x)\psi = (x)\phi \cdot e_r$. But then

$$(x)\phi = e_l \cdot (x)\psi = e_l \cdot (x)\phi \cdot e_r = (x)\phi \cdot e_r = (x)\psi,$$

so $\phi = \psi$. □

The effect of Corollary 5.13 is that, for a finite semigroup with a regular minimal ideal, we can talk about *the* retraction of a retractable ideal without any loss of generality. We can now use our two building blocks to produce a new congruence: a *lifted congruence.*

Definition 5.14. Let S be a semigroup with minimal ideal M, let I be a retractable ideal of S, and let σ be a liftable congruence of S. We associate to the pair (I, σ) the relation

$$\zeta_{I,\sigma} = \Big\{(x, y) \in I \times I : \big((x)\phi, (y)\phi\big) \in \sigma\Big\} \cup \Delta_S,$$

where ϕ is the unique retraction from I to M. We call $\zeta_{I,\sigma}$ the **lifted congruence** of (I, σ).

In order to justify the name *lifted congruence*, we require the following theorem.

Theorem 5.15. *The relation $\zeta_{I,\sigma}$ in Definition 5.14 is a congruence on S.*

Proof. For conciseness, let us refer to $\zeta_{I,\sigma}$ as ζ. Let (x, y) be a pair in ζ and let $s \in S$. To show ζ is a congruence, we must show that (sx, sy) and (xs, ys) both lie in ζ. If $(x, y) \in \Delta_S$, this is certainly true. Otherwise, we have $x, y \in I$ and $\big((x)\phi, (y)\phi\big) \in \sigma$. Since I is an ideal, we certainly have $sx, sy \in I$. Now by Definition 5.11(ii), and by Lemma 5.12, we have

$$\big(s \cdot (x)\phi, s \cdot (y)\phi\big) = \big((sx)\phi, (sy)\phi\big) \in \sigma,$$

so $(sx, sy) \in \zeta$. A symmetric argument gives us $(xs, ys) \in \zeta$. $\square$

This construction now gives us a usable source of congruences. All that is required is to find some liftable congruences and retractable ideals of a semigroup, and a number of new congruences can be described. It turns out that this is an excellent source of congruences for $\mathcal{M}_n$, yielding every non-Rees congruence on the semigroup, as we will see later.

5.3 Congruence lattice of $\mathcal{M}_n$

We can now apply the general theory of Section 5.2 to the Motzkin monoid, in order to find its congruences. First, let us mention the easiest congruences to describe – the Rees congruences (Definition 1.51).

Proposition 5.16. *The Rees congruences of $\mathcal{M}_n$ are the relations*

$$R_k = \{(x, y) \in \mathcal{M}_n \times \mathcal{M}_n : \operatorname{rank} x, \operatorname{rank} y \leq k\} \cup \Delta_{\mathcal{M}_n},$$

for $k \in \{0, \dots, n\}$.

Proof. This follows immediately from the description of the ideals of $\mathcal{M}_n$ in Proposition 5.7 part (v). $\square$

We will refer to these congruences by the name R_k for the rest of this chapter. We will soon see that R_0 and R_1 are in fact lifted congruences. The higher Rees congruences are not lifted congruences, as we will see in Corollary 5.34. Next, we will describe some other lifted congruences, by identifying some liftable congruences and retractions in $\mathcal{M}_n$ to use as building blocks.

First, recall that $I_0 = \{\alpha \in \mathcal{M}_n : \operatorname{rank} \alpha = 0\}$ is the minimal ideal of $\mathcal{M}_n$. Let us denote by $\mathscr{L}^{I_0}$ and $\mathscr{R}^{I_0}$ the $\mathscr{L}$- and $\mathscr{R}$-relations of $\mathcal{M}_n$ restricted to I_0, and let Δ_{I_0} and ∇_{I_0} be the trivial and universal congruences respectively on I_0.

Proposition 5.17. *The relations Δ_{I_0}, $\mathscr{L}^{I_0}$, $\mathscr{R}^{I_0}$ and ∇_{I_0} are all liftable congruences of $\mathcal{M}_n$.*

Proof. Since I_0 is a semigroup, Δ_{I_0} and ∇_{I_0} are certainly congruences of I_0; and both satisfy Definition 5.11(i), since their unions with $\Delta_{\mathcal{M}_n}$ are the congruences $\Delta_{\mathcal{M}_n}$ and R_0 respectively.

To see that $\mathscr{L}^{I_0}$ is a liftable congruence, consider Definition 5.11(ii); let $(a, b) \in \mathscr{L}^{I_0}$ and $x \in \mathcal{M}_n$. Since I_0 is the minimal ideal, we certainly have $xa, xb, ax, bx \in I_0$; and since

$\mathscr{L}$ is a right congruence on $\mathcal{M}_n$ (see Proposition 1.54) we have $(ax, bx) \in \mathscr{L}$ and therefore $(ax, bx) \in \mathscr{L}^{I_0}$. By Lemma 5.9, since $a \in I_0$, we also have $a(xa) = a$, so $xa \mathrel{\mathscr{L}} a$ and similarly $xb \mathrel{\mathscr{L}} b$. This means that $xa \mathrel{\mathscr{L}} a \mathrel{\mathscr{L}} b \mathrel{\mathscr{L}} xb$, so $(xa, xb) \in \mathscr{L}^{I_0}$. Hence $\mathscr{L}^{I_0}$ is a liftable congruence of $\mathcal{M}_n$, and by a similar argument, so is $\mathscr{R}^{I_0}$. $\square$

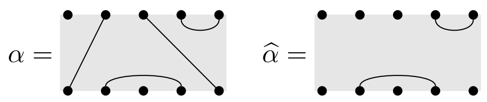
Now that we have some liftable congruences, we also want some retractable ideals in order to form lifted congruences. The following construction establishes one such ideal.

Definition 5.18. If α is a bipartition, then $\widehat{\alpha}$ is the unique bipartition of rank 0 with the same kernel and cokernel as α.

The element $\widehat{\alpha}$ can be computed easily from α: each transversal is split into an upper block (the points in $\mathbf{n}$) and a lower block (the points in $\mathbf{n}'$) and nothing else is changed. If we have a diagram for α, drawn in the standard way described after Example 1.75, then we simply remove any lines crossing the diagram. If we are using two-row notation, we can simply draw a horizontal line between the two rows. See Figure 5.19 for an example. Note that $\widehat{\alpha} = \alpha$ for all $\alpha \in I_0$.

$$\alpha = \big\{\{1\}, \{2, 1'\}, \{3, 5'\}, \{4, 5\}, \{2', 4'\}, \{3'\}\big\}$$

$$\widehat{\alpha} = \big\{\{1\}, \{2\}, \{3\}, \{4, 5\}, \{1'\}, \{2', 4'\}, \{3'\}, \{5'\}\big\}$$



$$\alpha = \left[\begin{array}{c|c|c|c} 2 & 3 & 1 & 4,5 \\ \cline{3-4} 1 & 5 & 2,4 & 3 \end{array}\right] \qquad \widehat{\alpha} = \left[\begin{array}{c|c|c|c} 2 & 3 & 1 & 4,5 \\ \hline 1 & 5 & 2,4 & 3 \end{array}\right]$$

Figure 5.19: Computing $\widehat{\alpha}$ from α.

Proposition 5.20. *The map $\phi : I_1 \to I_0$ defined by $\alpha \mapsto \widehat{\alpha}$ is a retraction. Hence, I_1 is a retractable ideal.*

Proof. Since $\widehat{\alpha} = \alpha$ for $\alpha \in I_0$, we can see that ϕ satisfies the condition $(m)\phi = m$ from Definition 5.10. Hence we only need to show that ϕ is a homomorphism. Let $\alpha, \beta \in I_1$, and we will try to prove that $\widehat{\alpha\beta} = \widehat{\alpha}\widehat{\beta}$. If both α and β have rank 0 then $\widehat{\alpha\beta} = \alpha\beta = \widehat{\alpha}\widehat{\beta}$. On the other hand, if at least one of α and β has rank 1 (without loss of generality, α) then we may write $\alpha = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \dots & A_r \\ \cline{2-4} B_0 & B_1 & \dots & B_s \end{array}\right]$ and $\beta = \left[\begin{array}{c|c|c|c} C_0 & C_1 & \dots & C_t \\ \cline{2-4} D_0 & D_1 & \dots & D_u \end{array}\right]$ or $\beta = \left[\begin{array}{c|c|c|c} C_0 & C_1 & \dots & C_t \\ \hline D_0 & D_1 & \dots & D_u \end{array}\right]$. This gives us $\alpha\beta = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \dots & A_r \\ \cline{2-4} D_0 & D_1 & \dots & D_u \end{array}\right]$ if β has the first form and $B_0 \cap C_0 \neq \varnothing$, or $\alpha\beta = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \dots & A_r \\ \hline D_0 & D_1 & \dots & D_u \end{array}\right]$ otherwise. Applying ϕ gives us $\widehat{\alpha} = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \dots & A_r \\ \hline B_0 & B_1 & \dots & B_s \end{array}\right]$, $\widehat{\beta} = \left[\begin{array}{c|c|c|c} C_0 & C_1 & \dots & C_t \\ \hline D_0 & D_1 & \dots & D_u \end{array}\right]$, and finally, in either case, $\widehat{\alpha\beta} = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \dots & A_r \\ \hline D_0 & D_1 & \dots & D_u \end{array}\right] = \widehat{\alpha}\widehat{\beta}$, so ϕ is a homomorphism. $\square$

This gives us a retractable ideal I_1, with a retraction $\alpha \mapsto \widehat{\alpha}$. It is also trivial to see that I_0 itself is retractable, with retraction $\alpha \mapsto \alpha$, the identity map. Corollary 5.13 shows that these retractions are unique.

We now have four liftable congruences $\big\{\Delta_{I_0}, \mathscr{L}^{I_0}, \mathscr{R}^{I_0}, \nabla_{I_0}\big\}$ and two retractable ideals $\{I_0, I_1\}$, giving rise to $4 \times 2 = 8$ lifted congruences by Definition 5.14 and Theorem 5.15. The congruences lifted from ∇_{I_0} are observed to be equal to the Rees congruences R_0 and R_1, while those lifted from Δ_{I_0}, $\mathscr{L}^{I_0}$ and $\mathscr{R}^{I_0}$ are named with appropriate Greek symbols, as follows:

$$\begin{aligned}
\delta_0 = \zeta_{I_0,\Delta_{I_0}} &= \{(\alpha,\beta) \in I_0 \times I_0 : (\alpha,\beta) \in \Delta_{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
\delta_1 = \zeta_{I_1,\Delta_{I_0}} &= \{(\alpha,\beta) \in I_1 \times I_1 : (\widehat{\alpha},\widehat{\beta}) \in \Delta_{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
\lambda_0 = \zeta_{I_0,\mathscr{L}^{I_0}} &= \{(\alpha,\beta) \in I_0 \times I_0 : (\alpha,\beta) \in \mathscr{L}^{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
\lambda_1 = \zeta_{I_1,\mathscr{L}^{I_0}} &= \{(\alpha,\beta) \in I_1 \times I_1 : (\widehat{\alpha},\widehat{\beta}) \in \mathscr{L}^{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
\rho_0 = \zeta_{I_0,\mathscr{R}^{I_0}} &= \{(\alpha,\beta) \in I_0 \times I_0 : (\alpha,\beta) \in \mathscr{R}^{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
\rho_1 = \zeta_{I_1,\mathscr{R}^{I_0}} &= \{(\alpha,\beta) \in I_1 \times I_1 : (\widehat{\alpha},\widehat{\beta}) \in \mathscr{R}^{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
R_0 = \zeta_{I_0,\nabla_{I_0}} &= \{(\alpha,\beta) \in I_0 \times I_0 : (\alpha,\beta) \in \nabla_{I_0}\} \cup \Delta_{\mathcal{M}_n},\\
R_1 = \zeta_{I_1,\nabla_{I_0}} &= \{(\alpha,\beta) \in I_1 \times I_1 : (\widehat{\alpha},\widehat{\beta}) \in \nabla_{I_0}\} \cup \Delta_{\mathcal{M}_n}.
\end{aligned}$$

This naming convention is summarised in Table 5.21.

	I_0	I_1
Δ_{I_0}	δ_0	δ_1
$\mathscr{L}^{I_0}$	λ_0	λ_1
$\mathscr{R}^{I_0}$	ρ_0	ρ_1
∇_{I_0}	R_0	R_1

Table 5.21: Lifted congruences of $\mathcal{M}_n$.

Interpreting these statements along with the use of Proposition 5.7 gives the following characterisation of the lifted congruences in terms of a bipartition's rank, kernel and cokernel.

Proposition 5.22. *The lifted congruences described above can be characterised in the following way, where* $(\alpha,\beta) \in \mathcal{M}_n \times \mathcal{M}_n$*:*

$$\begin{aligned}
\delta_0 &= \Delta_{\mathcal{M}_n},\\
\delta_1 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq 1, \ker\alpha = \ker\beta, \operatorname{coker}\alpha = \operatorname{coker}\beta\} \cup \Delta_{\mathcal{M}_n},\\
\lambda_0 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta = 0, \operatorname{coker}\alpha = \operatorname{coker}\beta\} \cup \Delta_{\mathcal{M}_n},\\
\lambda_1 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq 1, \operatorname{coker}\alpha = \operatorname{coker}\beta\} \cup \Delta_{\mathcal{M}_n},\\
\rho_0 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta = 0, \ker\alpha = \ker\beta\} \cup \Delta_{\mathcal{M}_n},\\
\rho_1 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq 1, \ker\alpha = \ker\beta\} \cup \Delta_{\mathcal{M}_n},\\
R_0 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta = 0\} \cup \Delta_{\mathcal{M}_n},\\
R_1 &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq 1\} \cup \Delta_{\mathcal{M}_n}.
\end{aligned}$$

Proof. Apart from reflexive pairs, the congruences $\delta_0, \lambda_0, \rho_0$ and R_0 contain only pairs from $I_0 \times I_0$. In δ_0 these pairs are all from Δ_{I_0}, and are therefore are in $\Delta_{\mathcal{M}_n}$ anyway, so it is equal to $\Delta_{\mathcal{M}_n}$. In λ_0, the non-reflexive pairs are precisely those in $\mathscr{L}^{I_0}$: we know that the elements in I_0 are those that have rank 0, and therefore all have empty codomains; and we know from Proposition 5.7(ii) that elements are $\mathscr{L}$-related if and only if they share a codomain and cokernel, so we have the statement for λ_0. Similarly using Proposition 5.7(i) we have the statement for ρ_0. In R_0 the non-reflexive pairs are those from ∇_{I_0}: this relation simply relates all elements of rank 0 to each other, so we have the statement.

Moving onto δ_1, λ_1, ρ_1 and R_1, we can see that all the non-reflexive pairs are from $I_1 \times I_1$, and so they all consist of elements of rank less than or equal to 1. For δ_1, consider a pair $(\alpha, \beta) \in I_1 \times I_1$ such that $(\widehat{\alpha}, \widehat{\beta}) \in \Delta_{I_1}$, i.e. $\widehat{\alpha} = \widehat{\beta}$: by Definition 5.18 these are precisely the pairs with the same kernel and cokernel, so we have the statement for δ_1. For λ_1 we require $(\alpha, \beta) \in I_1 \times I_1$ such that $(\widehat{\alpha}, \widehat{\beta}) \in \mathscr{L}^{I_0}$. All elements of I_0 have empty codomain, so a pair satisfies this if and only if $\widehat{\alpha}$ and $\widehat{\beta}$ have the same cokernel. Since α and $\widehat{\alpha}$ share a cokernel, and β and $\widehat{\beta}$ share a cokernel, this is therefore satisfied if and only if α and β share a cokernel, and so we have the statement for λ_1. The statement for ρ_1 follows by a similar argument. Finally, observe that R_1 contains all pairs $(\alpha, \beta) \in I_1 \times I_1$ such that $(\widehat{\alpha}, \widehat{\beta}) \in \nabla_{I_0}$ – this applies to all pairs in $I_1 \times I_1$, so we have that R_1 unites all elements of rank less than or equal to 1, giving the statement as shown. □

These characterisations will help us later when we consider generating pairs for the congruences. We will discover later that these are the only lifted congruences on $\mathcal{M}_n$, but we have not yet shown this.

We are now ready to state the main theorem of this chapter, giving a full description of the congruence lattice of $\mathcal{M}_n$. Much of the work to prove this has already been done, and the rest of this section will be devoted to completing the proof. Note that our main theorem requires $n \geq 2$; if $n = 1$ then $\mathcal{M}_n$ has only 2 elements, and its only congruences are $\Delta_{\mathcal{M}_n}$ and $\nabla_{\mathcal{M}_n}$.

Theorem 5.23. *Let $\mathcal{M}_n$ be the Motzkin monoid, with $n \geq 2$. The following hold:*

(i) *The congruences of $\mathcal{M}_n$ are precisely $\{\delta_0, \delta_1, \lambda_0, \lambda_1, \rho_0, \rho_1, R_0, R_1, \ldots, R_n\}$;*

(ii) *The congruence lattice of $\mathcal{M}_n$ is as shown in Figure 5.24;*

(iii) *Every congruence of $\mathcal{M}_n$ is principal.*

The remainder of this section serves to prove Theorem 5.23, as follows. Let Γ be the set of relations stated in (i). That the relations in Γ are congruences has already been established. Next we consider the joins of these congruences in Lemmas 5.25 and 5.26. These show that the congruences join together as in Figure 5.24, and therefore that Γ is closed under taking joins. Then, in order to see that these congruences are all distinct, we analyse the possible generating pairs of each congruence in Lemmas 5.28, 5.29 and 5.30. These results, which are summarised in Table 5.27, exhaust all pairs in $\mathcal{M}_n \times \mathcal{M}_n$ and all congruences in Γ, proving that all the congruences in Γ are principal, and that there are no other principal congruences. Since any congruence is a join of principal congruences, this proves that $\mathcal{M}_n$ has no congruences other than those in Γ. This completes the proof of (i), (ii) and (iii).

We will now state the lemmas required to complete the proof of Theorem 5.23. Firstly, we will focus on meets and joins of congruences, and show that there are no other congruences that can be generated by taking meets and joins of congruences in Γ.

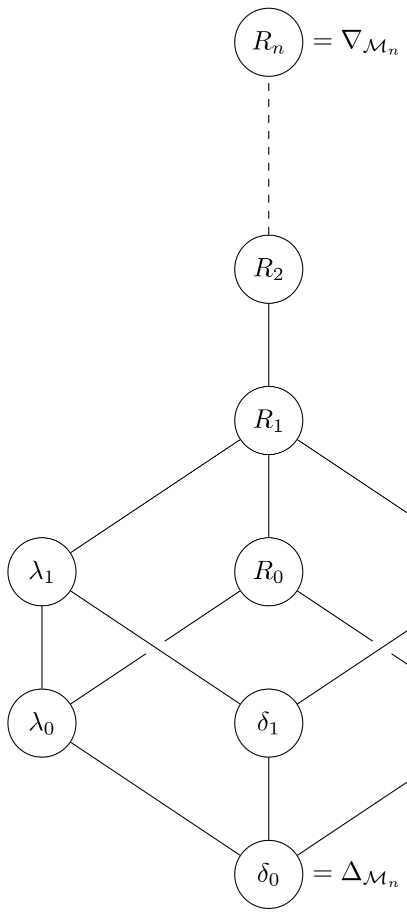


Figure 5.24: Congruence lattice of $\mathcal{M}_n$ for $n \geq 2$ (Hasse diagram).

Lemma 5.25. *Let $i \in \{0,1\}$ and $n \geq 2$. In $\mathcal{M}_n$, we have $\lambda_i \cap \rho_i = \delta_i$ and $\lambda_i \vee \rho_i = R_i$.*

Proof. Using Proposition 5.22, we have the following characterisations of δ_i, λ_i, ρ_i, and R_i:

$$\begin{aligned}
\delta_i &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq i, \ker\alpha = \ker\beta, \operatorname{coker}\alpha = \operatorname{coker}\beta\} \cup \Delta_{\mathcal{M}_n};\\
\lambda_i &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq i, \operatorname{coker}\alpha = \operatorname{coker}\beta\} \cup \Delta_{\mathcal{M}_n};\\
\rho_i &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq i, \ker\alpha = \ker\beta\} \cup \Delta_{\mathcal{M}_n};\\
R_i &= \{(\alpha,\beta) : \operatorname{rank}\alpha, \operatorname{rank}\beta \leq i\} \cup \Delta_{\mathcal{M}_n}.
\end{aligned}$$

The first statement, $\lambda_i \cap \rho_i = \delta_i$, follows directly from these characterisations. For the second, observe that since $\lambda_i \subseteq R_i$ and $\rho_i \subseteq R_i$, we must have $\lambda_i \vee \rho_i \subseteq R_i$. For $R_i \subseteq \lambda_i \vee \rho_i$, let $(\mu,\nu) \in R_i$. Observe that $\widehat{\mu\nu}$ has rank 0, the kernel of μ and the cokernel of ν. Hence $\mu \; \rho_i \; \widehat{\mu\nu} \; \lambda_i \; \nu$, so $(\mu,\nu) \in \lambda_i \vee \rho_i$, as required. □

Lemma 5.26. *The following hold in $\mathcal{M}_n$, with $n \geq 2$:*

(i) $\lambda_0 \subseteq \lambda_1$, $\rho_0 \subseteq \rho_1$, *and* $\delta_0 \subseteq \delta_1$;

(ii) $\lambda_0 \cap \rho_1 = \rho_0 \cap \lambda_1 = \delta_0$;

(iii) $\lambda_0 \vee \rho_1 = \rho_0 \vee \lambda_1 = R_1$;

(iv) $\lambda_0 \cap \delta_1 = \rho_0 \cap \delta_1 = \delta_0$;

(v) $\lambda_0 \vee \delta_1 = \lambda_1$ *and* $\rho_0 \vee \delta_1 = \rho_1$;

(vi) $R_0 \cap \delta_1 = \delta_0$ *and* $R_0 \vee \delta_1 = R_1$.

Proof. We can see (i) and (ii) immediately from the descriptions of the congruences in Proposition 5.22. The same proposition, and the fact that bipartitions of rank 0 are equal if they have the same kernel and cokernel, also give us (iv) and the first part of (vi).

For (iii), we will prove that $\lambda_0 \vee \rho_1 = R_1$, and observe that the rest follows by a similar argument. Since $\lambda_0 \subseteq R_1$ and $\rho_1 \subseteq R_1$, we must have $\lambda_0 \vee \rho_1 \subseteq R_1$. To prove $R_1 \subseteq \lambda_0 \vee \rho_1$, let $(\mu, \nu) \in R_1$. We certainly have $\mu \ \rho_1 \ \widehat{\mu}$ and $\nu \ \rho_1 \ \widehat{\nu}$. Observe that the product $\widehat{\mu}\widehat{\nu}$ has rank 0, the kernel of $\widehat{\mu}$, and the cokernel of $\widehat{\nu}$. Hence $\mu \ \rho_1 \ \widehat{\mu} \ \rho_1 \ \widehat{\mu}\widehat{\nu} \ \lambda_0 \ \widehat{\nu} \ \rho_1 \ \nu$, so $(\mu, \nu) \in \lambda_0 \vee \rho_1$, as required.

For (v), we will prove that $\lambda_0 \vee \delta_1 = \lambda_1$, and observe that the other part follows by a similar argument. Since $\lambda_0 \subseteq \lambda_1$ and $\delta_1 \subseteq \lambda_1$, we must have $\lambda_0 \vee \delta_1 \subseteq \lambda_1$. To prove $\lambda_1 \subseteq \lambda_0 \vee \delta_1$, let $(\mu, \nu) \in \lambda_1$. By the characterisation of λ_1 in Proposition 5.22, we have $\operatorname{coker} \mu = \operatorname{coker} \nu$, and therefore $\operatorname{coker} \widehat{\mu} = \operatorname{coker} \widehat{\nu}$. Hence $\mu \ \delta_1 \ \widehat{\mu} \ \lambda_0 \ \widehat{\nu} \ \delta_1 \ \nu$, so $(\mu, \nu) \in \lambda_0 \vee \delta_1$, as required.

Finally we prove the second part of (vi), $R_0 \vee \delta_1 = R_1$. Since $R_0 \subseteq R_1$ and $\delta_1 \subseteq R_1$, we must have $R_0 \vee \delta_1 \subseteq R_1$. To prove $R_1 \subseteq R_0 \vee \delta_1$, let $(\mu, \nu) \in R_1$. We certainly have $\mu \ \delta_1 \ \widehat{\mu}$ and $\nu \ \delta_1 \ \widehat{\nu}$, and since R_0 relates any pair of bipartitions of rank 0, we have $\mu \ \delta_1 \ \widehat{\mu} \ R_0 \ \widehat{\nu} \ \delta_1 \ \nu$, so $(\mu, \nu) \in R_0 \vee \delta_1$, as required. $\square$

The last two lemmas together prove that the congruences in Γ form the lattice shown in Figure 5.24 with respect to containment. Now we only need to show that there are no other principal congruences on $\mathcal{M}_n$, and the proof of Theorem 5.23 will be complete. We will do this by considering the generating pairs of all the congruences in Γ, and showing that any pair in $\mathcal{M}_n \times \mathcal{M}_n$ generates one of them. The results are summarised in Table 5.27.

$(\alpha,\beta)^\sharp$	$(\alpha,\beta) \in$	Reference
δ_0	δ_0	Trivial
δ_1	$\delta_1 \setminus \delta_0$	Lemma 5.28(iii)
λ_0	$\lambda_0 \setminus \delta_0$	Lemma 5.28(i)
λ_1	$\lambda_1 \setminus (\lambda_0 \cup \delta_1)$	Lemma 5.29(i)
ρ_0	$\rho_0 \setminus \delta_0$	Lemma 5.28(ii)
ρ_1	$\rho_1 \setminus (\rho_0 \cup \delta_1)$	Lemma 5.29(ii)
R_0	$R_0 \setminus (\lambda_0 \cup \rho_0)$	Lemma 5.29(iii)
R_1	$R_1 \setminus (\lambda_1 \cup \rho_1 \cup R_0)$	Lemma 5.29(iv)
$R_{k\geq 2}$	$R_k \setminus R_{k-1}$	Lemma 5.30

Table 5.27: Generating pairs for each congruence on $\mathcal{M}_n$.

For the remainder of this section, where there is no ambiguity we may write the trivial congruence $\Delta_{\mathcal{M}_n}$ as simply Δ, for brevity and readability.

Lemma 5.28. *Let $\alpha, \beta \in \mathcal{M}_n$, where $n \geq 2$. The following hold:*

(i) $\lambda_0 = (\alpha,\beta)^\sharp$ *if and only if* $(\alpha,\beta) \in \lambda_0 \setminus \Delta$*;*

(ii) $\rho_0 = (\alpha,\beta)^\sharp$ *if and only if* $(\alpha,\beta) \in \rho_0 \setminus \Delta$*;*

(iii) $\delta_1 = (\alpha,\beta)^\sharp$ *if and only if* $(\alpha,\beta) \in \delta_1 \setminus \Delta$*.*

Proof. In each statement, the "only if" part is obvious. We will prove (i) and observe that (ii) follows from a symmetric argument. Then we will prove (iii) separately.

For (i), let $(\alpha,\beta) \in \lambda_0 \setminus \Delta$, and let $\sigma = (\alpha,\beta)^\sharp$. Since λ_0 is a congruence, we clearly have $\sigma \subseteq \lambda_0$; hence we have only to prove that $\lambda_0 \subseteq \sigma$. First we require a special construction: if $\gamma \in I_0$, let γ' be the unique bipartition in I_0 with trivial kernel and $\operatorname{coker}\gamma' = \operatorname{coker}\gamma$. We claim that $(\gamma,\gamma') \in \sigma$ for any such γ. This claim is proven by induction on r, the number of kernel-classes of γ: if $r = n$ (trivial kernel) then $\gamma = \gamma'$ and we are done. Otherwise we have $r \leq n-1$, and we write $\gamma = \left[\begin{array}{c|c|c} A_1 & \dots & A_r \\ \hline B_1 & \dots & B_s \end{array}\right]$. Since $(\alpha,\beta) \in \lambda_0 \setminus \Delta$, Proposition 5.22 gives us $\operatorname{rank}\alpha = \operatorname{rank}\beta = 0$ and $\operatorname{coker}\alpha = \operatorname{coker}\beta$, but since $\alpha \neq \beta$ we must have $\ker\alpha \neq \ker\beta$. Swapping α and β if necessary, let us assume there exists some $(i,j) \in \ker\alpha \setminus \ker\beta$, and without loss of generality, assume $i < j$. We will write $\mathbf{n} \setminus \{i,j\}$ as $\{k_1,\dots,k_{n-2}\}$. Since $r \leq n-1$ there exists some kernel block of γ with 2 elements; let m be the lowest point in $\mathbf{n}$ in a non-trivial kernel block, and without loss of generality, let us assume m lies in A_1. We can now define the bipartition $\tau = \left[\begin{array}{c|c|c|c|c} m & p & A_2 & \dots & A_r \\ \cline{3-5} i & j & k_1 & \dots & k_{n-2} \end{array}\right]$, where $A_1 = \{m,p\}$. We observe that $\tau\alpha\gamma = \gamma = \left[\begin{array}{cc|c|c|c} \multicolumn{2}{c|}{m,p} & A_2 & \dots & A_r \\ \hline B_1 & \multicolumn{1}{|c|}{B_2} & B_3 & \dots & B_s \end{array}\right]$ and $\tau\beta\gamma = \left[\begin{array}{c|c|c|c|c} m & p & A_2 & \dots & A_r \\ \hline B_1 & B_2 & B_3 & \dots & B_s \end{array}\right]$. Since σ is left- and right-compatible, we deduce that $\gamma = \tau\alpha\gamma$ is σ-related to $\tau\beta\gamma$. Hence γ is σ-related to $\tau\beta\gamma$, a bipartition with rank 0, the same cokernel as γ, and $r+1$ kernel classes. Applying the same process inductively, with $\tau\beta\gamma$ in place of γ, implies a chain of σ-relations which relate γ to a bipartition with rank 0, the same cokernel as γ, and n kernel classes – that is, γ'. This proves the claim that $(\gamma,\gamma') \in \sigma$.

To return to the proof that $\lambda_0 \subseteq \sigma$, let $(\mu,\nu) \in \lambda_0$ be arbitrary. If $\mu = \nu$ then certainly $(\mu,\nu) \in \sigma$, so let us assume $\mu \neq \nu$. By Proposition 5.22 we must have $\operatorname{rank}\mu = \operatorname{rank}\nu = 0$ and $\operatorname{coker}\mu = \operatorname{coker}\nu$, so we have $\mu' = \nu'$. Hence, by the above claim, we have $\mu \;\sigma\; \mu' = \nu' \;\sigma\; \nu$, so $(\mu,\nu) \in \sigma$, and (i) is complete. Observe that (ii) follows by a similar argument.

To prove (iii), let $(\alpha,\beta) \in \delta_1 \setminus \Delta$ as stated, and let $\sigma = (\alpha,\beta)^\sharp$. Clearly $\sigma \subseteq \delta_1$; it remains to prove that $\delta_1 \subseteq \sigma$. Since $(\alpha,\beta) \in \delta_1 \setminus \Delta$, α and β must each have rank 0 or 1, and have the same kernel and cokernel, but be distinct. Since there is only one bipartition of rank 0 with a given kernel and cokernel, they cannot both have rank 0. Hence, swapping α and β if necessary, we may assume that $\operatorname{rank}(\alpha) = 1$, with transversal $\{i,j'\}$, and we can write $\alpha = \left[\begin{array}{c|c|c|c} i & A_1 & \cdots & A_r \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$. Then β has one of the following four forms, where without loss of generality, additional labelled elements are assumed to be from A_1 or B_1:

(a) $\beta = \left[\begin{array}{c|c|c|c|c} i & A_1 & A_2 & \cdots & A_r \\ \cline{2-5} j & B_1 & B_2 & \cdots & B_s \end{array}\right]$, so that $\beta = \widehat{\alpha}$;

(b) $\beta = \left[\begin{array}{c|c|c|c|c} k & i & A_2 & \cdots & A_r \\ \cline{2-5} j & B_1 & B_2 & \cdots & B_s \end{array}\right]$, so that β is the same as α but with k in the transversal instead of i;

(c) $\beta = \left[\begin{array}{c|c|c|c|c} i & A_1 & A_2 & \cdots & A_r \\ \cline{2-5} l & j & B_2 & \cdots & B_s \end{array}\right]$, so that β is the same as α but with l' in the transversal instead of j';

(d) $\beta = \left[\begin{array}{c|c|c|c|c} k & i & A_2 & \cdots & A_r \\ \cline{2-5} l & j & B_2 & \cdots & B_s \end{array}\right]$, so that β is the same as α but with transversal $\{k,l'\}$ instead of $\{i,j'\}$.

Now, for any $a,b \in \mathbf{n}$, let us denote by τ_{ab} the bipartition $(a,b')^e \in \mathcal{M}_n$ – this has just one non-trivial block, $\{a,b'\}$. Let us use $\tau_\varnothing$ to denote the bipartition in $\mathcal{M}_n$ consisting entirely of singletons. We will use the bipartitions τ_{ii} and τ_{jj}. In all four cases above, we have $\tau_{ii}\alpha\tau_{jj} = \tau_{ij}$ and $\tau_{ii}\beta\tau_{jj} = \tau_\varnothing$. Since $\alpha \;\sigma\; \beta$, we also have $\tau_{ii}\alpha\tau_{jj} \;\sigma\; \tau_{ii}\beta\tau_{jj}$, so $\tau_{ij} \;\sigma\; \tau_\varnothing$. Next, let γ be an arbitrary bipartition in $\mathcal{M}_n$ with rank 1. We can write $\gamma = \left[\begin{array}{c|c|c|c} c & C_1 & \cdots & C_t \\ \cline{2-4} d & D_1 & \cdots & D_u \end{array}\right]$, where $\{c,d'\}$ is the one transversal. Let us write $\mathbf{n} \setminus \{i\} = \{i_1,\dots,i_{n-1}\}$ and $\mathbf{n} \setminus \{j\} = \{j_1,\dots,j_{n-1}\}$. Then we can define two new bipartitions: $\overline{\gamma} = \left[\begin{array}{c|c|c|c} c & C_1 & \cdots & C_t \\ \cline{2-4} i & i_1 & \cdots & i_{n-1} \end{array}\right]$ and $\underline{\gamma} = \left[\begin{array}{c|c|c|c} j & j_1 & \cdots & j_{n-1} \\ \cline{2-4} d & D_1 & \cdots & D_u \end{array}\right]$. We can see that

$\gamma = \overline{\gamma}\tau_{ij}\underline{\gamma}$ and $\widehat{\gamma} = \overline{\gamma}\tau_{\varnothing}\underline{\gamma}$, so we have $\gamma \mathrel{\sigma} \widehat{\gamma}$ for any γ of rank 1. The same statement is also true for γ of rank 0, since $\gamma = \widehat{\gamma}$.

To prove that $\delta_1 \subseteq \sigma$, let $(\mu, \nu) \in \delta_1$ be arbitrary. Each of μ and ν must have rank 0 or 1, and they must have the same kernel and cokernel. Hence we have $\mu \mathrel{\sigma} \widehat{\mu} = \widehat{\nu} \mathrel{\sigma} \nu$, so $(\mu, \nu) \in \sigma$, completing the proof of (iii). □

Lemma 5.29. *Let $\alpha, \beta \in \mathcal{M}_n$, where $n \geq 2$. The following hold:*

(i) $\lambda_1 = (\alpha, \beta)^\sharp$ *if and only if* $(\alpha, \beta) \in \lambda_1 \setminus (\lambda_0 \cup \delta_1)$*;*

(ii) $\rho_1 = (\alpha, \beta)^\sharp$ *if and only if* $(\alpha, \beta) \in \rho_1 \setminus (\rho_0 \cup \delta_1)$*;*

(iii) $R_0 = (\alpha, \beta)^\sharp$ *if and only if* $(\alpha, \beta) \in R_0 \setminus (\lambda_0 \cup \rho_0)$*;*

(iv) $R_1 = (\alpha, \beta)^\sharp$ *if and only if* $(\alpha, \beta) \in R_1 \setminus (\lambda_1 \cup \rho_1 \cup R_0)$.

Proof. In each statement, as in the previous lemma, the "only if" part is obvious; we just need to consider the right-to-left implications. First we will prove (i) and observe that (ii) follows from a symmetric argument. Then we will prove (iii) and (iv) separately.

For (i), start by supposing $(\alpha, \beta) \in \lambda_1 \setminus (\lambda_0 \cup \delta_1)$, as in the premise. Clearly $(\alpha, \beta)^\sharp \subseteq \lambda_1$. To be outside λ_0, either α or β must have rank 1 – without loss of generality, assume $\operatorname{rank}(\alpha) = 1$. We may therefore write $\alpha = \left[\begin{array}{c|c|c|c} i & A_1 & \cdots & A_r \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$. Now, since $\operatorname{rank}\beta \leq 1$ and $\operatorname{coker}\alpha = \operatorname{coker}\beta$, we may write β in one of the following ways:

(a) $\beta = \left[\begin{array}{c|c|c|c|c} C_0 & C_1 & C_2 & \cdots & C_t \\ \hline j & B_1 & B_2 & \cdots & B_s \end{array}\right]$;

(b) $\beta = \left[\begin{array}{c|c|c|c|c} i & C_1 & C_2 & \cdots & C_t \\ \cline{2-5} j & B_1 & B_2 & \cdots & B_s \end{array}\right]$;

(c) $\beta = \left[\begin{array}{c|c|c|c|c} k & C_1 & C_2 & \cdots & C_t \\ \cline{2-5} j & B_1 & B_2 & \cdots & B_s \end{array}\right]$, for some $k \neq i$;

(d) $\beta = \left[\begin{array}{c|c|c|c|c} i & C_1 & C_2 & \cdots & C_t \\ \cline{2-5} l & j & B_2 & \cdots & B_s \end{array}\right]$, for some $l \neq j$;

(e) $\beta = \left[\begin{array}{c|c|c|c|c} k & C_1 & C_2 & \cdots & C_t \\ \cline{2-5} l & j & B_2 & \cdots & B_s \end{array}\right]$, for some $k \neq i$ and $l \neq j$.

Since $(\alpha, \beta) \notin \delta_1$, we have $\ker\alpha \neq \ker\beta$. Let $\gamma = \left[\begin{array}{c|c|c|c} j & B_1 & \cdots & B_s \\ \hline j & B_1 & \cdots & B_s \end{array}\right]$, so that $\operatorname{coker}(\alpha\gamma) = \operatorname{coker}(\beta\gamma)$, and hence $(\alpha\gamma, \beta\gamma) \in \lambda_0$. In particular, since $\ker\alpha \neq \ker\beta$, we find $\alpha\gamma \neq \beta\gamma$, and therefore $(\alpha\gamma, \beta\gamma) \in \lambda_0 \setminus \Delta$, so Lemma 5.28(i) gives us $\lambda_0 = (\alpha\gamma, \beta\gamma)^\sharp \subseteq (\alpha, \beta)^\sharp$. Since $\lambda_1 = \lambda_0 \vee \delta_1$ by Lemma 5.26, we need only show that $\delta_1 \subseteq (\alpha, \beta)^\sharp$, and (i) is complete. To do this, we will consider the cases (a)–(e) separately.

Firstly, assume (a) holds. We know that $\alpha\alpha^\star\alpha = \alpha$, and we can see that $\alpha\alpha^\star\beta = \widehat{\alpha}$. Hence Lemma 5.28(iii) gives $\delta_1 = (\alpha, \widehat{\alpha})^\sharp = (\alpha\alpha^\star\alpha, \alpha\alpha^\star\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$, so $\delta_1 \subseteq (\alpha, \beta)^\sharp$.

Next, suppose (b) holds. Since $\alpha \neq \beta$, the blocks A_1 to A_r cannot be the same as the blocks C_1 to C_t. Hence, swapping α and β if necessary, let $(a_1, a_2) \in \ker\alpha \setminus \ker\beta$. Now let $\mathbf{n} \setminus \{i, a_1, a_2\} = \{i_1, \ldots, i_{n-3}\}$, let $\tau = \left[\begin{array}{c|c|c|c|c} a_1 & i, a_2 & i_1 & \ldots & i_{n-3} \\ a_1 & i, a_2 & i_1 & \ldots & i_{n-3} \end{array}\right]$, and note that $\tau\alpha = \left[\begin{array}{c|c|c|c|c} a_1 & i, a_2 & i_1 & \ldots & i_{n-3} \\ \cline{2-5} j & B_1 & B_2 & \ldots & B_s \end{array}\right]$ but $\tau\beta = \left[\begin{array}{c|c|c|c|c} a_1 & i, a_2 & i_1 & \ldots & i_{n-3} \\ \hline j & B_1 & B_2 & \ldots & B_s \end{array}\right]$. Hence we have $(\tau\alpha, \tau\beta) \in \delta_1 \setminus \Delta$, so Lemma 5.28(iii) gives us $\delta_1 = (\tau\alpha, \tau\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$.

Next, suppose (c) holds. Let $\tau = (i, i')^e$, the bipartition containing just one non-trivial block $\{i, i'\}$, let $\mathbf{n} \setminus \{i\} = \{i_1, \ldots, i_{n-1}\}$, and note that $\tau\alpha = \left[\begin{array}{c|c|c|c} i & i_1 & \cdots & i_{n-1} \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$ but $\tau\beta = \left[\begin{array}{c|c|c|c} i & i_1 & \cdots & i_{n-1} \\ \hline j & B_1 & \cdots & B_s \end{array}\right]$. Again we have $(\tau\alpha, \tau\beta) \in \delta_1 \setminus \Delta$, so by Lemma 5.28(iii) we have $\delta_1 = (\tau\alpha, \tau\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$.

Next, suppose (d) holds. Again let $\mathbf{n}\setminus\{i\} = \{i_1, \ldots, i_{n-1}\}$, let $\tau = \left[\begin{array}{c|c|c|c} i & A_1 & \cdots & A_r \\ \cline{2-4} i & i_1 & \cdots & i_{n-1} \end{array}\right]$, and note that $\tau\alpha = \alpha$ and $\tau\beta = \left[\begin{array}{c|c|c|c|c} i & A_1 & A_2 & \cdots & A_r \\ \cline{2-5} l & j & B_2 & \cdots & B_s \end{array}\right]$, so that $\tau\alpha$ and $\tau\beta$ have the same kernel and cokernel but a different transversal. We have $(\tau\alpha, \tau\beta) \in \delta_1 \setminus \Delta$, so by Lemma 5.28(iii) we again have $\delta_1 = (\tau\alpha, \tau\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$.

Finally, suppose (e) holds. Let $\mathbf{n}\setminus\{i, k\} = \{k_1, \ldots, k_{n-2}\}$, let τ be the bipartition with non-trivial blocks $\{i, i'\}$ and $\{k, k'\}$, and note that $\tau\alpha = \left[\begin{array}{c|c|c|c|c} i & k & k_1 & \ldots & k_{n-2} \\ \cline{3-5} j & l & B_2 & \ldots & B_s \end{array}\right]$ and $\tau\beta = \left[\begin{array}{c|c|c|c|c} k & i & k_1 & \ldots & k_{n-2} \\ \cline{3-5} l & j & B_2 & \ldots & B_s \end{array}\right]$ We have $(\tau\alpha, \tau\beta) \in \delta_1 \setminus \Delta$, so again by Lemma 5.28(iii) we have $\delta_1 = (\tau\alpha, \tau\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$.

We have now considered all 5 cases, and shown that we always have $\delta_1 \subseteq (\alpha, \beta)^\sharp$. Hence $\lambda_1 \subseteq (\alpha, \beta)^\sharp$, and the proof of (i) is complete. Note that (ii) follows by a symmetric argument.

For (iii), suppose $(\alpha, \beta) \in R_0 \setminus (\rho_0 \cup \lambda_0)$. Since $\operatorname{rank}\alpha = \operatorname{rank}\beta = 0$, we may write $\alpha = \left[\begin{array}{c|c|c} A_1 & \cdots & A_r \\ \hline B_1 & \cdots & B_s \end{array}\right]$ and $\beta = \left[\begin{array}{c|c|c} C_1 & \cdots & C_t \\ \hline D_1 & \cdots & D_u \end{array}\right]$, noting that, since $(\alpha, \beta) \notin \rho_0$, we must have $\{A_1, \ldots, A_r\} \neq \{C_1, \ldots, C_t\}$. By Lemma 5.25, we have $R_0 = \lambda_0 \vee \rho_0$, so we will prove $(\alpha, \beta)^\sharp$ contains R_0 by showing that it contains both λ_0 and ρ_0. Let $\gamma = \alpha\beta = \left[\begin{array}{c|c|c} A_1 & \cdots & A_r \\ \hline D_1 & \cdots & D_u \end{array}\right]$. This gives us $(\gamma, \beta) \in \lambda_0 \setminus \Delta$, so by Lemma 5.28(i) we have $\lambda_0 = (\gamma, \beta)^\sharp = (\alpha\beta, \beta\beta)^\sharp \subseteq (\alpha, \beta)^\sharp$. By a similar argument we have $\rho_0 \subseteq (\alpha, \beta)^\sharp$, and hence $R_0 \subseteq (\alpha, \beta)^\sharp$. It is obvious that $(\alpha, \beta)^\sharp \subseteq R_0$, so (iii) is complete.

Finally, for (iv), suppose $(\alpha, \beta) \in R_1 \setminus (\lambda_1 \cup \rho_1 \cup R_0)$, as in the premise. Since the pair is in R_1, the elements' ranks must both be at most 1; but since it is not in R_0, at least one must be of rank 1 (without loss of generality, assume α). Since the pair is in neither λ_1 nor ρ_1, we also know that $\ker\alpha \neq \ker\beta$ and $\operatorname{coker}\alpha \neq \operatorname{coker}\beta$. Hence we can write $\alpha = \left[\begin{array}{c|c|c|c} i & A_1 & \cdots & A_r \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$, and $\beta = \left[\begin{array}{c|c|c|c} k & C_1 & \cdots & C_t \\ \cline{2-4} l & D_1 & \cdots & D_u \end{array}\right]$ or $\beta = \left[\begin{array}{c|c|c|c} k & C_1 & \cdots & C_t \\ \hline l & D_1 & \cdots & D_u \end{array}\right]$, with $\{\{i\}, A_1, \ldots, A_r\} \neq \{\{k\}, C_1, \ldots, C_t\}$. Now, as in (iii), since $R_1 = \lambda_1 \vee \rho_1$, by Lemma 5.25, we prove that $R_1 \subseteq (\alpha, \beta)^\sharp$ by showing that $(\alpha, \beta)^\sharp$ contains λ_1 and ρ_1. We will prove the statement for λ_1, and observe that ρ_1 follows by a similar argument. Let us proceed by solving three cases separately. One of the following three statements about β must hold:

(f) $\operatorname{rank}(\beta) = 0$;

(g) $\operatorname{rank}(\beta) = 1$ and $j = l$;

(h) $\operatorname{rank}(\beta) = 1$ and $j \neq l$.

First, suppose (f) or (g) holds. Let $\gamma = \left[\begin{array}{c|c|c|c} j & B_1 & \cdots & B_s \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$. Certainly we have $\alpha\gamma = \alpha$. To find $\beta\gamma$ we separate into two cases: in case (f) we have $\beta\gamma = \left[\begin{array}{c|c|c|c} k & C_1 & \cdots & C_t \\ \hline j & B_1 & \cdots & B_s \end{array}\right]$, and in case (g) we have $\beta\gamma = \left[\begin{array}{c|c|c|c} k & C_1 & \cdots & C_t \\ \cline{2-4} j & B_1 & \cdots & B_s \end{array}\right]$. In either case, we have $(\alpha\gamma, \beta\gamma) \in \lambda_1 \setminus (\lambda_0 \cup \delta_1)$, so by (i), we have $\lambda_1 = (\alpha\gamma, \beta\gamma)^\sharp \subseteq (\alpha, \beta)^\sharp$, completing this case.

Next, assume (h) holds. Write $\mathbf{n} \setminus \{j\} = \{j_1, \ldots, j_{n-1}\}$, and let $\tau = (j, j')^e$, the bipartition whose only non-trivial block is $\{j, j'\}$. Then we have $\alpha\tau = \left[\begin{array}{c|c|c|c} i & A_1 & \cdots & A_r \\ \cline{2-4} j & j_1 & \cdots & j_{n-1} \end{array}\right]$ and $\beta\tau = \left[\begin{array}{c|c|c|c} k & C_1 & \cdots & C_t \\ \cline{2-4} j & j_1 & \cdots & j_{n-1} \end{array}\right]$. We have $(\alpha\tau, \beta\tau) \in \lambda_1 \setminus (\lambda_0 \cup \delta_1)$, so again by (i), we have $\lambda_1 = (\alpha\tau, \beta\tau)^\sharp \subseteq (\alpha, \beta)^\sharp$, completing this case. This completes the proof that $\lambda_1 \subseteq (\alpha, \beta)^\sharp$, and the proof that $\rho_1 \subseteq (\alpha, \beta)^\sharp$ is similar. Hence $\lambda_1 \vee \rho_1 = R_1 \subseteq (\alpha, \beta)^\sharp$. It is obvious that $(\alpha, \beta)^\sharp \subseteq R_1$, and so the proof of (iv) is complete. $\square$

Lemma 5.30. *Let $\alpha, \beta \in \mathcal{M}_n$, where $n \geq 2$, and let $k \in \{2, \ldots, n\}$. We have $R_k = (\alpha, \beta)^\sharp$ if and only if $(\alpha, \beta) \in R_k \setminus R_{k-1}$.*

Proof. Since R_k and R_{k-1} are congruences, the "only if" part of the statement is obvious. For the right-to-left implication, let $k \in \{2, \ldots, n\}$, and let $(\alpha, \beta) \in R_k \setminus R_{k-1}$. For brevity, let $\sigma = (\alpha, \beta)^\sharp$. It is clear that $\sigma \subseteq R_k$ since R_k is a congruence. We now only need to prove that $R_k \subseteq \sigma$.

For (α, β) to lie in $R_k \setminus R_{k-1}$, at least one of α and β must have rank k. Without loss of generality, assume $\operatorname{rank} \alpha = k$ and $\operatorname{rank} \beta \leq k$. There must be exactly k transversals in α, and since $\alpha \in \mathcal{M}_n$ they must all have size 2. Let $\operatorname{dom} \alpha = \{i_1, \ldots, i_k\}$ and $\operatorname{codom} \alpha = \{j_1, \ldots, j_k\}$, with $i_1 < \ldots < i_k$ and $j_1 < \ldots < j_k$; since α is planar, its transversals must be $\{i_1, j_1'\}, \ldots, \{i_k, j_k'\}$.

Our proof now splits into two cases. Since $\alpha \neq \beta$ and $\operatorname{rank} \beta \leq \operatorname{rank} \alpha$, one of the following holds:

(a) α and β have precisely the same transversals $\{i_1, j_1'\} \ldots \{i_k, j_k'\}$, but their other blocks differ – without loss of generality, assume there exists some $(a_1, a_2) \in \ker \alpha \setminus \ker \beta$ with $a_1 < a_2$;

(b) there exists a transversal $\{i_x, j_x'\}$ of α that is not a block of β.

We will now prove two facts: firstly that $(\gamma, \widehat{\gamma}) \in \sigma$ for all $\gamma \in I_k$; and secondly that $R_0 \subseteq \sigma$.

For the first claim, let $\gamma \in I_k$, and let $r = \operatorname{rank} \gamma \leq k$. If $r = 0$, then $\gamma = \widehat{\gamma}$ and the claim is satisfied; hence let us assume $r \geq 1$. We may write $\gamma = \left[\begin{array}{c|c|c|c|c|c} c_1 & \cdots & c_r & C_1 & \cdots & C_t \\ \cline{4-6} d_1 & \cdots & d_r & D_1 & \cdots & D_u \end{array}\right]$. For ease of notation, we will also write $\mathbf{n} \backslash \{i_1, \ldots, i_r\} = \{a_1, \ldots, a_{n-r}\}$ and $\mathbf{n} \backslash \{j_1, \ldots, j_r\} = \{b_1, \ldots, b_{n-r}\}$. First, assume (a) holds. Let x be the maximal number such that $1 \leq x \leq r$ and $i_x < a_1$ (if this is not possible, we instead let x be the minimal number such that $a_2 < i_x$, producing a case the same as what we describe, but flipped horizontally). Note that since α is planar we must have $i_x < a_1 < a_2 < i_{x+1}$ if i_{x+1} exists. Now define

$$\tau = \left[\begin{array}{c|c|c|c|c|c|c|c|c|c|c} c_1 & \cdots & c_{x-1} & c_x & c_{x+1} & \cdots & c_r & C_1 & \multicolumn{2}{c|}{\cdots} & C_t \\ \cline{8-11} i_1 & \cdots & i_{x-1} & a_2 & i_{x+1} & \cdots & i_r & i_x, a_1 & a_3 & \cdots & a_{n-r} \end{array}\right]$$

and

$$\kappa = \left[\begin{array}{c|c|c|c|c|c} j_1 & \cdots & j_r & b_1 & \cdots & b_{n-r} \\ \cline{4-6} d_1 & \cdots & d_r & D_1 & \cdots & D_u \end{array}\right].$$

We have $\tau\alpha\kappa = \gamma$, but

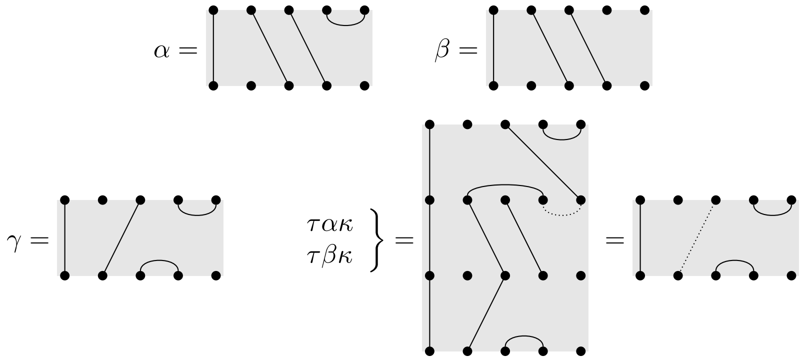
$$\tau\beta\kappa = \left[\begin{array}{c|c|c|c|c|c|c|c|c|c} c_1 & \cdots & c_{x-1} & c_{x+1} & \cdots & c_r & c_x & C_1 & \cdots & C_t \\ \cline{7-10} d_1 & \cdots & d_{x-1} & d_{x+1} & \cdots & d_r & d_x & D_1 & \cdots & D_u \end{array}\right],$$

a copy of γ but with the transversal $\{c_x, d_x'\}$ broken into singletons. See Figure 5.31 for an illustrative example. Hence γ is σ-related to a bipartition with the same kernel and cokernel but lower rank. Applying this procedure repeatedly and using transitivity relates γ to a bipartition with the same kernel and cokernel but rank 0, that is, $\widehat{\gamma}$. Hence $(\gamma, \widehat{\gamma}) \in \sigma$ in case (a).

For case (b), instead let

$$\tau = \left[\begin{array}{c|c|c|c|c|c} c_1 & \cdots & c_r & C_1 & \cdots & C_t \\ \cline{4-6} i_1 & \cdots & i_r & a_1 & \cdots & a_{n-r} \end{array}\right] \quad \text{and} \quad \kappa = \left[\begin{array}{c|c|c|c|c|c} j_1 & \cdots & j_r & b_1 & \cdots & b_{n-r} \\ \cline{4-6} d_1 & \cdots & d_r & D_1 & \cdots & D_u \end{array}\right].$$

This gives us $\tau\alpha\kappa = \gamma$, but $\tau\beta\kappa$ equal to a bipartition γ' which shares a kernel and cokernel with γ but which has a lower rank. Hence we have $(\gamma, \gamma') \in \sigma$, and repeating this process eventually gives $(\gamma, \widehat{\gamma}) \in \sigma$.

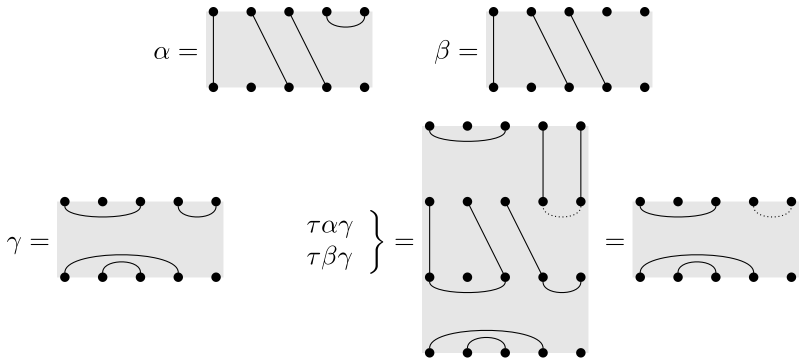


$\alpha =$
$\beta =$
$\gamma =$
$\left.\begin{matrix}\tau\alpha\kappa\\ \tau\beta\kappa\end{matrix}\right\} =$
$=$

Figure 5.31: Splitting a transversal of γ in Lemma 5.30 case (a).

Next we will show that $R_0 \subseteq \sigma$ – that is, that σ relates any pair of elements of rank 0. We do this by showing that any element γ of rank 0 is σ-related to $\varnothing^e$, the bipartition consisting of just singletons; our claim follows by transitivity. Let $\gamma = \left[\begin{smallmatrix} C_1 & \dots & C_t \\ D_1 & \dots & D_u \end{smallmatrix}\right]$.

First, assume (a) holds. We will show that any upper or lower block in γ can be split into singletons to obtain a bipartition which is σ-related to γ. First, splitting an upper block: choose an upper block of γ of size 2 (assume without loss of generality that it is C_1), label its elements $C_1 = \{c_1, c_2\}$ with $c_1 < c_2$. Recall that $(a_1, a_2) \in \ker\alpha \setminus \ker\beta$, and write $\mathbf{n} \setminus \{a_1, a_2\} = \{a_3, \dots, a_n\}$. We define τ to be the bipartition with transversals $\{c_1, a_1'\}$ and $\{c_2, a_2'\}$, upper (non-transversal) blocks $C_2, \dots, C_t$, and the rest singletons. Observe that $\tau\alpha\gamma = \gamma$ but $\tau\beta\gamma$ is equal to a copy of γ with the block $\{c_1, c_2\}$ split into two blocks $\{c_1\}$ and $\{c_2\}$. This process is illustrated in Figure 5.32.



$\alpha =$
$\beta =$
$\gamma =$
$\left.\begin{matrix}\tau\alpha\gamma\\ \tau\beta\gamma\end{matrix}\right\} =$
$=$

Figure 5.32: Splitting an upper block of γ in Lemma 5.30 case (a).

Next, splitting a lower block: choose a lower block of γ of size 2 (assume without loss of generality that it is D_1'), and label its elements $D_1' = \{d_1', d_2'\}$ with $d_1 < d_2$. Now choose two indices $p, q \in \{1, \dots, k\}$ such that $i_p < i_q < a_1 < a_2$ (or, if this is impossible, such that $a_1 < a_2 < i_p < i_q$ or $i_q < a_1 < a_2 < i_p$). Let τ be the bipartition of rank 0 with the upper blocks of γ and the lower blocks $\{i_p', a_2'\}$, $\{i_q', a_1'\}$, and the rest singletons. Let κ be the bipartition with 2 transversals $\{j_p, d_1'\}$ and $\{j_q, d_2'\}$, lower blocks $D_2', \dots, D_u'$, and the rest of its blocks singletons. Then $\tau\alpha\kappa = \gamma$, but $\tau\beta\kappa$ is equal to a copy of γ with the block $\{d_1', d_2'\}$ split into two singletons. This process is illustrated in Figure 5.33.

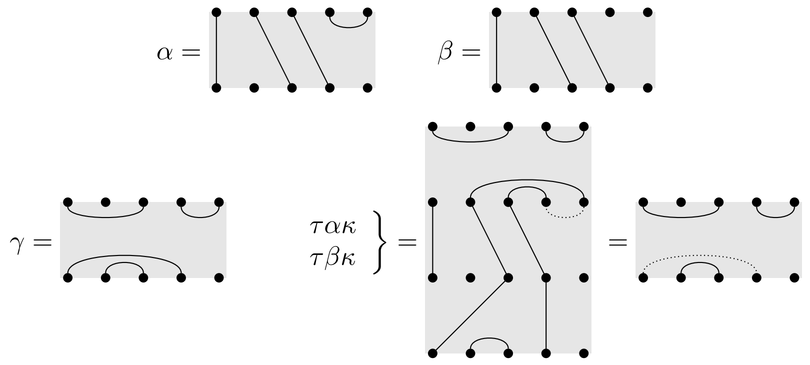


Figure 5.33: Splitting a lower block of γ in Lemma 5.30 case (a).

In either of the block-splitting procedures just mentioned, it should be noted that we cannot split a block enveloped by another block: that is, we cannot split a block $\{x_2, x_3\}$ if there exists another block $\{x_1, x_4\}$ with $x_1 < x_2 < x_3 < x_4$. However, we can first split the block $\{x_1, x_4\}$ leaving the block $\{x_2, x_3\}$ to be split later.

We have now shown that we can split any block of γ to produce a bipartition σ-related to γ. Hence, we can split every block and we reach $\varnothing^e$, showing that $(\gamma, \varnothing^e) \in \sigma$, and therefore that any two bipartitions of rank 0 are σ-related, in case (a).

Now assume (b), and recall that $\gamma = \left[\begin{array}{c|c|c} C_1 & \dots & C_t \\ \hline D_1 & \dots & D_u \end{array}\right]$. Once again, assume without loss of generality that C_1 has size 2, so $C_1 = \{c_1, c_2\}$. By (b) we have some transversal $\{i_x, j'_x\}$ in α but not in β; let $\{i_y, j'_y\}$ be another transversal from α, which may or may not be in β. Let τ be the bipartition with transversals $\{c_1, i'_x\}$ and $\{c_2, i'_y\}$, upper blocks $C_2, \dots, C_t$, and lower blocks all singletons. Let κ be the bipartition of rank 0 with lower blocks $D'_1, \dots, D'_u$, an upper block $\{j_x, j_y\}$, and the rest singletons. We have $\tau\alpha\kappa = \gamma$, but $\tau\beta\kappa$ equal to a copy of γ with the upper block $\{c_1, c_2\}$ split into singletons. This can be performed repeatedly to split all upper blocks, and a similar process can be applied to split all lower blocks. Note, again, that a block enveloped by another block cannot be split immediately. We can now see that, as in (a), any two bipartitions of rank 0 are σ-related.

We have now proven the two facts in both cases, so we can proceed to show that $R_k \subseteq \sigma$. Let $(\mu, \nu) \in R_k$. If $\mu = \nu$ then certainly $(\mu, \nu) \in \sigma$; otherwise, μ and ν must both lie in I_k. Hence by the first fact, we have $(\mu, \widehat{\mu}), (\nu, \widehat{\nu}) \in \sigma$. By the second fact, since $\operatorname{rank} \widehat{\mu} = \operatorname{rank} \widehat{\nu} = 0$, we have $(\widehat{\mu}, \widehat{\nu}) \in \sigma$. Hence we have $\mu \; \sigma \; \widehat{\mu} \; \sigma \; \widehat{\nu} \; \sigma \; \nu$, and by transitivity we can see $(\mu, \nu) \in \sigma$, as required. $\square$

We have now classified the generating pairs of all the congruences in Γ, and we find that this exhausts all the pairs in $\mathcal{M}_n \times \mathcal{M}_n$, as summarised in Table 5.27. This proves that Γ includes all the principal congruences on $\mathcal{M}_n$, and since we know from Lemmas 5.25 and 5.26 that these are closed under taking joins, we can conclude that there are no other congruences on $\mathcal{M}_n$. Hence the congruence lattice in Figure 5.24 is complete, and we have completed the proof of Theorem 5.23.

We will state one corollary regarding the retractable ideals of $\mathcal{M}_n$.

Corollary 5.34. *Let $n \geq 2$ and $k \geq 2$. The ideal I_k of $\mathcal{M}_n$ is not retractable, and hence the congruence R_k is not a lifted congruence.*

Proof. Assume that $k \geq 2$, and that I_k exists (i.e. $n \geq 2$) and is retractable. Then we can choose a liftable congruence and use it with I_k to create a lifted congruence. We choose $\mathscr{L}^{I_0}$, which gives us the lifted congruence

$$\lambda_k = \zeta_{I_k, \mathscr{L}^{I_0}} = \{(\alpha, \beta) : \operatorname{rank} \alpha, \operatorname{rank} \beta \leq k, \operatorname{coker} \alpha = \operatorname{coker} \beta\} \cup \Delta_{\mathcal{M}_n}.$$

Observe that λ_k is not equal to any of the congruences described in Theorem 5.23, and so it is not a congruence on $\mathcal{M}_n$, a contradiction. □

5.4 Other monoids

In Section 5.3 we described the congruences of the Motzkin monoid. These constructions and results were originally described in [EMRT18], culminating in the classification of the Motzkin monoid's congruence lattice as shown in Figure 5.24. However, that paper also considers other diagram semigroups: the bipartition monoid $\mathcal{P}_n$ and several of its other submonoids. Though these monoids are not the subject of this chapter, their congruence lattices are closely related to that of $\mathcal{M}_n$, and they use much of the preliminary theory described in Section 5.2. We will therefore briefly describe these monoids and outline their congruence lattices, for completeness, referring the reader to [EMRT18] for a full explanation.

5.4.1 IN-pairs

Before we can meaningfully describe the congruences of the other monoids in this section, we must make a definition which extends that of a lifted congruence (Definition 5.14) by further relating elements outside the ideal I, using a subgroup N that lies above I in the order of $\mathscr{J}$-classes.

Definition 5.35 ([EMRT18, Definitions 3.16 & 3.17])**.** Let S be a finite semigroup. An **IN-pair** on S is a pair (I, N) consisting of:

- an ideal I of S; and
- a normal subgroup N of some maximal subgroup of S that lies in a regular $\mathscr{J}$-class of S that is minimal among the $\mathscr{J}$-classes in $S \setminus I$.

The IN-pair (I, N) is called **retractable** if the following hold:

(i) I is retractable (Definition 5.10);

(ii) $|Nx| = |xN| = 1$ for each x in the minimal ideal of S.

In the same way we associated to a retractable ideal I and liftable congruence σ the lifted congruence $\zeta_{I,\sigma}$ (Definition 5.14), we can associate to a rectractable IN-pair (I, N) and liftable congruence σ a relation $\zeta_{I,N,\sigma}$ defined by

$$\zeta_{I,N,\sigma} = \zeta_{I,\sigma} \cup \{(sxt, syt) \in J \times J : x, y \in N \text{ and } s, t \in S^1\},$$

where J is the $\mathscr{J}$-class containing N.

We also define a relation which can always be built from an IN-pair, whether it is retractable or not. If (I, N) is an IN-pair, then let the relation $R_{I,N}$ be defined by

$$R_{I,N} = R_I \cup \{(sxt, syt) \in J \times J : x, y \in N \text{ and } s, t \in S^1\},$$

where J is the $\mathscr{J}$-class containing N. This can be viewed as an extension of the definition of a Rees congruence R_I. It turns out that both the relations $\zeta_{I,N,\sigma}$ and $R_{I,N}$ are congruences on S [EMRT18, Proposition 3.22]. We will use these to describe congruences on the other monoids considered in the following section.

Note that two IN-pairs (I, N_1) and (I, N_2) may contain normal subgroups N_1 and N_2 from two different maximal subgroups G_1 and G_2 of J (which must be isomorphic to each other). It turns out that all possible congruences $\zeta_{I,N,\sigma}$ and $R_{I,N}$ can be found using the normal subgroups of just one maximal subgroup of J, and that it does not matter which subgroup is chosen. We will therefore simply consider the isomorphism class of a maximal subgroup in J, without needing to specify which precise subgroup we are working with.

5.4.2 Results

We will now consider a number of submonoids in turn, giving the classification of their congruences. The total number of congruences of each monoid is shown in Table 5.36.

Monoid	Size	Number of congruences
$\mathcal{S}_n$	$n!$	3
$\mathcal{I}_n$	$\sum_{k=0}^{n} \binom{n}{k} \frac{n!}{(n-k)!}$	$3n - 1$
$\mathcal{POI}_n$	$\binom{2n}{n}$	$n + 1$
$\mathcal{M}_n$	$\sum_{k=0}^{n} \binom{2n}{2k} C_k$	$n + 7$
$\mathscr{P}\mathcal{P}_n$	C_{2n}	$n + 7$
$\mathcal{B}_n$	$(2n - 1)!!$	$\frac{3}{2}n + \frac{5}{2}$ or $\frac{3}{2}n + 13$
$\mathcal{J}_n$	C_n	$\frac{1}{2}n + \frac{7}{2}$ or $\frac{1}{2}n + 7$
$\mathcal{PB}_n$	$\sum_{k=0}^{n} \binom{2n}{2k} (2k - 1)!!$	$3n + 7$
$\mathcal{P}_n$	B_{2n}	$3n + 7$

Table 5.36: The number of congruences on various diagram monoids. Numbers shown are correct for $n \geq 5$.

We saw in Example 1.81 a way in which partial transformations lie in the partition monoid $\mathcal{P}_n$. We will therefore start with three monoids of partial transformations which embed into $\mathcal{P}_n$ as submonoids: $\mathcal{S}_n$, $\mathcal{I}_n$, and $\mathcal{POI}_n$. For all the monoids below, we assume $n \geq 3$, since any lower n has very few elements and is rather trivial to solve.

Symmetric group $\mathcal{S}_n$

The symmetric group $\mathcal{S}_n$ is isomorphic to the subgroup of $\mathcal{P}_n$ consisting of all bipartitions of rank n with blocks of size precisely 2. Since $\mathcal{S}_n$ is a group, its congruences are described by its normal subgroups (see Section 3.1.2). These normal subgroups are well known: the trivial group $\{\mathrm{id}\}$, the alternating group $\mathcal{A}_n$, the whole symmetric group $\mathcal{S}_n$ itself, and uniquely in the case that $n = 4$, the *Klein 4-group* $K_4 = \langle(1\ 2)(3\ 4), (1\ 3)(2\ 4)\rangle$. For $n \geq 3$ these normal subgroups (and hence these congruences) are all distinct.

Symmetric inverse monoid $\mathcal{I}_n$

Recall that the symmetric inverse monoid $\mathcal{I}_n$ consists of all the partial permutations of rank up to n under composition. This embeds into $\mathcal{P}_n$ as in Example 1.81 as the submonoid consisting of all bipartitions with trivial kernel and cokernel.

The ideals of $\mathcal{I}_n$ form a chain with respect to containment, and are precisely the sets

$$I_k = \{\alpha \in \mathcal{M}_n : \operatorname{rank} \alpha \leq k\},$$

for $k \in \{0, \dots, n\}$, as is the case for $\mathcal{P}_n$ and $\mathcal{M}_n$.

The congruences of $\mathcal{I}_n$ were classified in [Lib53], and are reformulated in the context of IN-pairs as follows.

Theorem 5.37 ([EMRT18, Theorem 4.1])**.** *Let $\mathcal{I}_n$ be the inverse symmetric monoid of degree n, for $n \geq 0$. The congruences of $\mathcal{I}_n$ form a chain, and are as follows:*

- *the Rees congruences R_k corresponding to the ideals I_k, for $k \in \{0, \dots, n\}$;*
- *the congruences $R_{I,N}$ corresponding to the IN-pairs (I_{k-1}, N) for $k \in \{2, \dots, n\}$ and $N \in \{K_4, \mathcal{A}_k, \mathcal{S}_k\}$ being any non-trivial normal subgroup of $\mathcal{S}_k$ (the group isomorphic to a maximal subgroup of J_k).*

Note that $\mathcal{A}_k$ and $\mathcal{S}_k$ will be used for every $k \in \{2, \dots, n\}$, but K_4 will only be used when $k = 4$. Since $\mathcal{A}_2$ is trivial, we have $R_{I_1,\mathcal{A}_2} = R_1$. Note also that, since there is only one bipartition in $\mathcal{I}_n$ of rank 0, we have $R_0 = \Delta_{\mathcal{I}_n}$. As an example, the congruences of $\mathcal{I}_6$ are as follows:

$$\begin{aligned}\Delta_{\mathcal{I}_6} = R_0 \subset R_1 & \\ & \subset R_{I_1,\mathcal{S}_2} \subset R_2 \\ & \subset R_{I_2,\mathcal{A}_3} \subset R_{I_2,\mathcal{S}_3} \subset R_3 \\ & \subset R_{I_3,K_4} \subset R_{I_3,\mathcal{A}_4} \subset R_{I_3,\mathcal{S}_4} \subset R_4 \\ & \subset R_{I_4,\mathcal{A}_5} \subset R_{I_4,\mathcal{S}_5} \subset R_5 \\ & \subset R_{I_5,\mathcal{A}_6} \subset R_{I_5,\mathcal{S}_6} \subset R_6 = \nabla_{\mathcal{I}_6}.\end{aligned}$$

Order-preserving partial permutation monoid $\mathcal{POI}_n$

The monoid $\mathcal{POI}_n$ of all order-preserving partial permutations embeds into $\mathcal{P}_n$ as the submonoid consisting of all the planar bipartitions with trivial kernel and cokernel. An element of $\mathcal{POI}_n$ is defined by its domain and codomain, so $\mathcal{POI}_n$ contains $\binom{n}{r}^2$ elements of rank r, or $\sum_{r=0}^{n} \binom{n}{r}^2 = \binom{2n}{n}$ elements in total [OEIS, A000984].

Its ideals have the same description as for $\mathcal{I}_n$, and its congruences are all Rees. Hence the congruences of $\mathcal{POI}_n$ form the chain

$$\Delta_{\mathcal{POI}_n} = R_0 \subset R_1 \subset R_2 \subset \ldots \subset R_n = \nabla_{\mathcal{POI}_n}.$$

This is shown in [Fer01, Proposition 2.6].

Planar bipartition monoid $\mathscr{P}\mathcal{P}_n$

The planar bipartition monoid $\mathscr{P}\mathcal{P}_n$ simply consists of all the planar bipartitions in $\mathcal{P}_n$. It therefore contains the Motzkin monoid, which has the additional restriction that a bipartition's blocks have size 1 or 2. The planar bipartition monoid $\mathscr{P}\mathcal{P}_n$ has a number of elements equal to the Catalan number C_{2n} [OEIS, A000108]. Its congruence lattice is in fact isomorphic to that of $\mathcal{M}_n$, and its congruences have the same descriptions (detailed in Theorem 5.23) [EMRT18, §7].

Brauer monoid $\mathcal{B}_n$

The Brauer monoid $\mathcal{B}_n$ consists of all the bipartitions in $\mathcal{P}_n$ whose blocks all have size 2. The Brauer monoid contains

$$(2n-1)!! = (2n-1) \cdot (2n-3) \cdots 5 \cdot 3 \cdot 1$$

elements in total [OEIS, A001147]. Since each block in $\mathcal{B}_n$ must have size 2, the monoid only contains elements with rank equal to $n \pmod 2$ – in particular, $\mathcal{B}_n$ never contains both an element of rank 0 and an element of rank 1. For this reason, in classifying the congruences of $\mathcal{B}_n$, we must consider two different cases: one in which n is odd, and one in which n is even.

The odd case is by far the simpler. All bipartitions in $\mathcal{B}_n$ are odd in this case, so the Rees congruences are $\{R_1, R_3, \ldots, R_n\}$. We also have lifted congruences $\{\delta_1, \lambda_1, \rho_1\}$ which are defined in the same way as for $\mathcal{M}_n$. Finally, via IN-pairs, we have $R_{I_{k-2},\mathcal{A}_k}$ and $R_{I_{k-2},\mathcal{S}_k}$ for each $k \in \{3, 5, \ldots, n\}$.

In the even case, there are many more congruences. All bipartitions are even, so the Rees congruences are $\{R_0, R_2, \ldots, R_n\}$. We also have lifted congruences

$$\{\zeta_{I_0,\Delta}, \zeta_{I_0,\mathscr{L}^{I_0}}, \zeta_{I_0,\mathscr{R}^{I_0}}, \zeta_{I_2,\Delta}, \zeta_{I_2,\mathscr{L}^{I_0}}, \zeta_{I_2,\mathscr{R}^{I_0}}\},$$

which we denote, in a way similar to the Motzkin monoid, as $\{\delta_0, \lambda_0, \rho_0, \delta_2, \lambda_2, \rho_2\}$. Some more congruences arise from IN-pairs: $(I_0, \mathcal{S}_2)$ gives rise to

$$\delta_{\mathcal{S}_2} = \zeta_{I_0,\mathcal{S}_2,\Delta}, \quad \lambda_{\mathcal{S}_2} = \zeta_{I_0,\mathcal{S}_2,\mathscr{L}^{I_0}}, \quad \rho_{\mathcal{S}_2} = \zeta_{I_0,\mathcal{S}_2,\mathscr{R}^{I_0}}, \quad R_{I_0,\mathcal{S}_2};$$

and (I_2, K_4) gives rise to

$$\delta_{K_4} = \zeta_{I_2,K_4,\Delta}, \quad \lambda_{K_4} = \zeta_{I_2,K_4,\mathscr{L}^{I_0}}, \quad \rho_{K_4} = \zeta_{I_2,K_4,\mathscr{R}^{I_0}}, \quad R_{I_2,K_4}.$$

Finally, we have $R_{I_{k-2},\mathcal{A}_k}$ and $R_{I_{k-2},\mathcal{S}_k}$ for each $k \in \{4, 6, \ldots, n\}$.

These results are proven in [EMRT18, §8], and the lattices are shown in Figure 5.38.

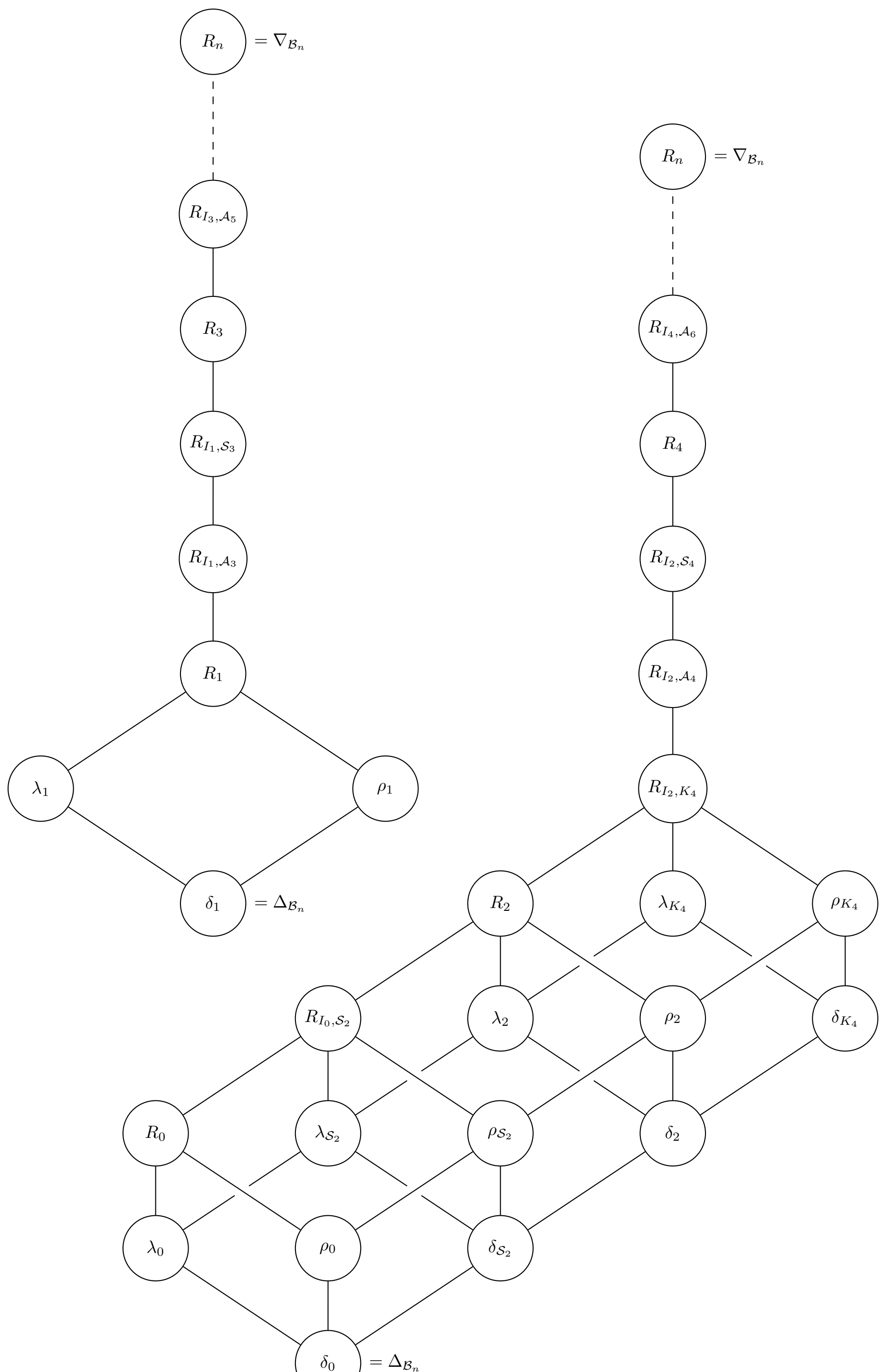


Figure 5.38: Congruence lattice of $\mathcal{B}_n$ for $n \geq 5$ when n is odd (upper left) and even (lower right).

Jones monoid $\mathcal{J}_n$

The Jones monoid $\mathcal{J}_n$ is the submonoid of $\mathcal{P}_n$ consisting of all planar bipartitions with blocks of size 2. By this definition, we can see that $\mathcal{J}_n = \mathscr{P}\mathcal{P}_n \cap \mathcal{B}_n$. Its size is given by the Catalan number C_n [OEIS, A000108].

As with the Brauer monoid, we consider two different cases based on whether n is odd or even; however, the congruence lattices are much simpler. If n is odd, then the only congruences are δ_1, λ_1, ρ_1, and the Rees congruences $\{R_1, R_3, \ldots, R_n\}$. If, on the other hand, n is even, then the congruence lattice is isomorphic to that of $\mathcal{M}_{n/2}$, and its description can be obtained by doubling each number in the description of that lattice. That is, the congruences are precisely

$$\{\delta_0, \delta_2, \lambda_0, \lambda_2, \rho_0, \rho_2, R_0, R_2, \ldots, R_n\}.$$

These results are proven in [EMRT18, §9], and the lattices are shown in Figure 5.39.

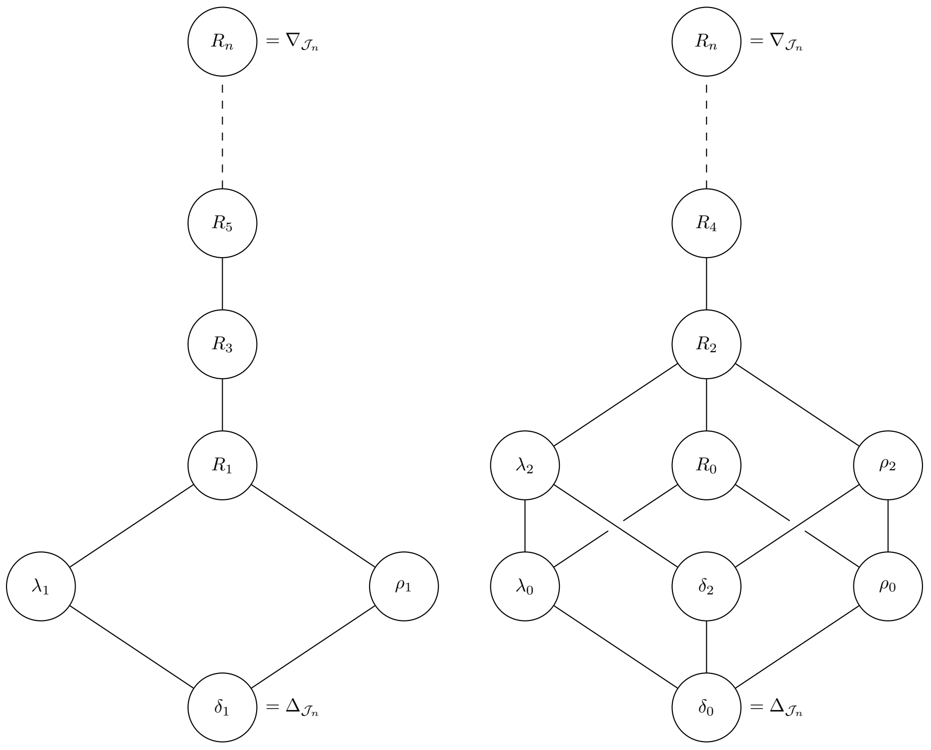


Figure 5.39: Congruence lattice of $\mathcal{J}_n$ for $n \geq 4$ when n is odd (left) and even (right).

Bipartition monoid $\mathcal{P}_n$ and partial Brauer monoid $\mathcal{PB}_n$

Finally, we can state the congruence lattice of the entire bipartition monoid $\mathcal{P}_n$. As in the case of the Motzkin monoid, we have Rees congruences $\{R_0, R_1, \ldots, R_n\}$ and lifted congruences $\{\delta_0, \delta_1, \lambda_0, \lambda_1, \rho_0, \rho_1\}$. The additional congruences on $\mathcal{P}_n$ come from IN-pairs: the retractable IN-pair $(I_1, \mathcal{S}_2)$ gives rise to congruences

$$\delta_{\mathcal{S}_2} = \zeta_{I_1,\mathcal{S}_2,\Delta}, \quad \lambda_{\mathcal{S}_2} = \zeta_{I_1,\mathcal{S}_2,\mathscr{L}^{I_0}}, \quad \rho_{\mathcal{S}_2} = \zeta_{I_1,\mathcal{S}_2,\mathscr{R}^{I_0}}, \quad R_{I_1,\mathcal{S}_2};$$

and the non-retractable IN-pairs $(I_{k-1}, \mathcal{A}_k)$ and $(I_{k-1}, \mathcal{S}_k)$ give us the congruences $R_{I_{k-1},\mathcal{A}_k}$ and $R_{I_{k-1},\mathcal{A}_k}$, for $k \in \{3, \ldots, n\}$. Uniquely for $k = 4$ we also have the IN-pair (I_3, K_4), yielding the congruence R_{I_3,K_4}. These are all the congruences on $\mathcal{P}_n$, as is proven in [EMRT18, §5]. The lattice is shown in Figure 5.40.

We should also mention the partial Brauer monoid $\mathcal{PB}_n$, the submonoid of $\mathcal{P}_n$ consisting of all the bipartitions with blocks of size 1 or 2. It has

$$\sum_{k=0}^{n} \binom{2n}{2k} (2k-1)!!$$

elements [OEIS, A066223], as shown in [Hd14, 2.1]. Again we can see that this monoid contains the Motzkin monoid; in fact, it is clear from the definitions that $\mathcal{M}_n = \mathscr{P}\mathcal{P}_n \cap \mathcal{PB}_n$. Its congruence lattice has the same description as that of $\mathcal{P}_n$ [EMRT18, §6], and is therefore also shown in Figure 5.40.

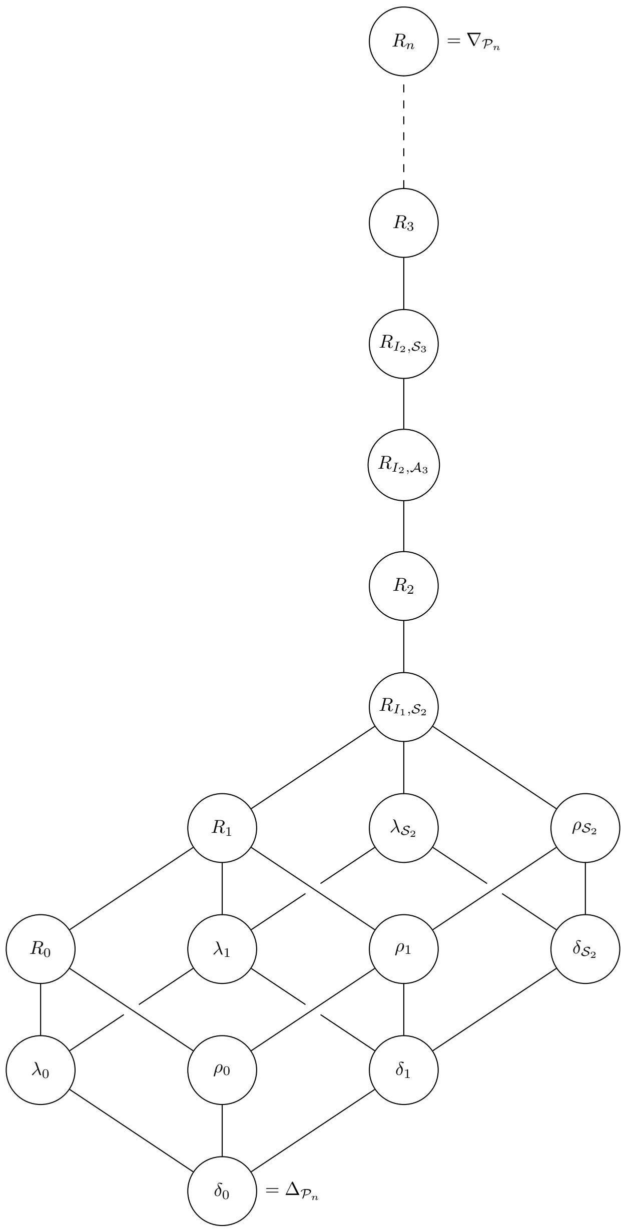


Figure 5.40: Congruence lattice of $\mathcal{P}_n$ (or $\mathcal{PB}_n$) for $n \geq 3$.

Chapter 6

Principal factors and counting congruences

In Chapter 5 we classified the congruences of the Motzkin monoid and several related diagram monoids. This classification was achieved by first calculating the congruence lattices for small values of n using the computational techniques described in Chapters 2 and 4, and then building up theory in order to prove a classification for general n. In this chapter we present some more results about congruences that were obtained in a similar way, by first looking for patterns in computational results, and then extending the results and attempting to prove them for larger semigroups. The libsemigroups library and the Semigroups and smallsemi packages for GAP were used to carry out the initial computations [MT$^+$18, M$^+$19, DM17, GAP18].

6.1 Congruences of principal factors

In this section, we will consider an interesting decomposition of a semigroup related to its $\mathscr{J}$-classes: a semigroup's *principal factors*. After defining this construction, we will consider the principal factors of the full transformation monoid $\mathcal{T}_n$, and classify their congruences. After this, we will look at the principal factors of some other, somewhat similar monoids, and classify their congruences using similar principles.

6.1.1 Principal factors

Recall that a semigroup's $\mathscr{J}$-classes have a natural partial order $\leq$, defined as follows: if J_1 and J_2 are $\mathscr{J}$-classes of S, then $J_1 \leq J_2$ if and only if $S^1J_1S^1 \subseteq S^1J_2S^1$. For finite semigroups we have $\mathscr{J} = \mathscr{D}$, and this partial order is shown on eggbox diagrams by the placement of $\mathscr{D}$-classes above and below each other, as in Figure 1.56. Given a $\mathscr{J}$-class J of a semigroup S, we can define the ideal I_J generated by J, which is given by $I_J = S^1JS^1$. If J is not minimal, we can also define the ideal of all $\mathscr{J}$-classes below J, which is given by $I_J \setminus J$. Since $I_J \setminus J$ is an ideal of I_J, we can use it to take a Rees congruence, and a Rees quotient (see Definition 1.51). This allows us to make the following definition.

Definition 6.1. Let S be a semigroup, and let J be a $\mathscr{J}$-class of S. The **principal factor** of J is denoted by $\overline{J}$, and defined by

$$\overline{J} = \begin{cases} J & \text{if } J \text{ is the minimal ideal;} \\ I_J/(I_J \setminus J) & \text{otherwise.} \end{cases}$$

If J is not the minimal ideal, then the principal factor $\overline{J}$ is isomorphic to the set $J \cup \{0\}$, with multiplication $\circ$ defined by

$$a \circ b = \begin{cases} ab & \text{if } a, b, ab \in J; \\ 0 & \text{otherwise.} \end{cases}$$

In the case that J is the minimal ideal of S, we will always have $ab \in J$, and so we do not have the element 0.

Since $\overline{J}$ is composed of a single $\mathscr{J}$-class, possibly with a zero appended, it is a simple or 0-simple semigroup. Hence, if S is finite, we may identify $\overline{J}$ with a Rees matrix semigroup $\mathcal{M}[G; I, \Lambda; P]$ or Rees 0-matrix semigroup $\mathcal{M}^0[G; I, \Lambda; P]$, by the Rees Theorem (Theorem 3.7). This will help us to classify its congruences later, using the concept of linked triples (see Definition 3.8).

6.1.2 Full transformation monoid $\mathcal{T}_n$

Now we will consider the principal factors of an important monoid, the full transformation monoid $\mathcal{T}_n$. Recall that $\mathcal{T}_n$ is the monoid consisting of all transformations on the set $\{1, \dots, n\}$, for some $n \in \mathbb{N}$ (Definition 1.62). In order to describe the principal factors of $\mathcal{T}_n$, we must first consider its Green's relations, as follows.

Proposition 6.2. *Let $n \in \mathbb{N}$, and let $\mathcal{T}_n$ be the full transformation monoid of degree n. For two mappings $\alpha, \beta \in \mathcal{T}_n$, the following hold:*

- *$\alpha \mathscr{L} \beta$ if and only if* $\operatorname{im} \alpha = \operatorname{im} \beta$*,*
- *$\alpha \mathscr{R} \beta$ if and only if* $\ker \alpha = \ker \beta$*,*
- *$\alpha \mathscr{D} \beta$ if and only if* $\operatorname{rank} \alpha = \operatorname{rank} \beta$*.*

The last part of the above proposition allows us to name the semigroup's $\mathscr{D}$-classes

$$D_1^n, D_2^n, \dots, D_n^n,$$

where each D_k^n is the $\mathscr{D}$-class of $\mathcal{T}_n$ consisting of transformations with rank k. Then the usual partial ordering of $\mathscr{J}$-classes (which in a finite semigroup are the same as $\mathscr{D}$-classes) gives $D_1^n < D_2^n < \dots < D_n^n$.

Inside a given $\mathscr{D}$-class D_k^n, elements are divided into $\mathscr{L}$-classes according to their image set; since all elements have rank k, their images must have size k, and so there are $\binom{n}{k}$ $\mathscr{L}$-classes in total. Similarly, the elements of D_k^n are divided into $\mathscr{R}$-classes according to their kernel; the possible kernels are all k-partitions of an n-set, so the total number of $\mathscr{R}$-classes is given by the Stirling number of the second kind, $S(n, k)$ [OEIS, A008277].

Each $\mathscr{H}$-class in D_k^n is the intersection of an $\mathscr{L}$-class and an $\mathscr{R}$-class, so each one corresponds to an image–kernel pair (hence we will talk about the *image and kernel of an $\mathscr{H}$-class*). For a given kernel with k classes and a given image with k elements, there are $k!$ different ways to assign image elements to kernel classes – hence there are $k!$ elements in each $\mathscr{H}$-class.

Group $\mathscr{H}$-classes of $\mathcal{T}_n$

To understand the principal factor corresponding to a $\mathscr{D}$-class D_k^n, we need to understand which of its $\mathscr{H}$-classes are groups and which are not. To determine which $\mathscr{H}$-classes are groups, we recall that in any semigroup an $\mathscr{H}$-class H is a group if and only if it contains an idempotent (an element $\alpha \in H$ such that $\alpha\alpha = \alpha$). A transformation $\alpha \in \mathcal{T}_n$ is an idempotent if and only if each point in its image is mapped by α to itself, i.e.

$$i\alpha = i \quad (\forall i \in \operatorname{im} \alpha).$$

Given an image and a kernel, we can choose a transformation with this condition if and only if no pair of points in the image are in the same kernel-class – that is, each image point is in a different kernel-class. Hence an $\mathscr{H}$-class of D_k^n is a group if and only if its image contains one point from each class of its kernel (i.e. its image is a *cross-section* of its kernel).

Lemma 6.3. *Let $k, n \in \mathbb{N}$ with $k \leq n$, and let D_k^n be the $\mathscr{D}$-class of $\mathcal{T}_n$ consisting of the elements of rank k. The following hold:*

(i) *For any two distinct $\mathscr{R}$-classes R_1 and R_2 of D_k^n there is an $\mathscr{L}$-class L such that $L \cap R_1$ is a group $\mathscr{H}$-class, but $L \cap R_2$ is not.*

(ii) *If $k > 1$, then for any two distinct $\mathscr{L}$-classes L_1 and L_2 of D_k^n there is an $\mathscr{R}$-class R such that $L_1 \cap R$ is a group $\mathscr{H}$-class, but $L_2 \cap R$ is not.*

Proof. For (i), let R_1 and R_2 be distinct $\mathscr{R}$-classes of D_k^n. These two classes correspond to distinct kernels P_1 and P_2, each partitioning $\{1, \ldots, n\}$ into k classes. If $k = n$ then there is only one possible partition, $\big\{\{1\}, \ldots, \{n\}\big\}$, and so R_1 and R_2 cannot be distinct. If $k < n$ then there must be a pair of elements $i, j \in \{1, \ldots, n\}$ which are in different classes of P_1 but the same class of P_2. Let X be a k-set containing one element from each class of P_1, including i and j – clearly it is a cross-section of P_1. But now X contains two elements from one class of P_2, so it is not a cross-section of P_2. Hence, if L is the $\mathscr{L}$-class corresponding to image X, $L \cap R_1$ is a group $\mathscr{H}$-class but $L \cap R_2$ is not a group $\mathscr{H}$-class.

For (ii), let L_1 and L_2 be distinct $\mathscr{L}$-classes of D_k^n, with $1 < k \leq n$. These two classes correspond to distinct images of size k in $\{1, \ldots, n\}$; let us call these images I_1 and I_2 respectively. Without loss of generality, let $I_1 = \{1, 2, \ldots, k\}$. Since $I_1 \neq I_2$, there must be an element $i \in \{1, \ldots, k\}$ not in I_2. Now consider the k-partition P which puts each element from $\{1, \ldots, k\}$ in a class on its own, apart from one element $j \in \{1, \ldots, k\}$ not equal to i, which is in a class with all the elements $\{k + 1, \ldots, n\}$ (choosing $j \neq i$ requires $k > 1$). Now I_1 is a cross-section of P, having precisely one element from each class; but I_2 does not have an element from the class $\{i\}$, and so it is not a cross-section of P. Let R be the $\mathscr{R}$-class with kernel P, and we have that $L_1 \cap R$ is a group $\mathscr{H}$-class but $L_2 \cap R$ is not. $\square$

Principal factors of $\mathcal{T}_n$

As mentioned in Section 6.1.1, any principal factor is either simple or 0-simple, and so it can be identified with a Rees matrix semigroup or Rees 0-matrix semigroup. Hence, for any $k > 1$, let

$\overline{D_k^n} = \mathcal{M}^0[G; I, \Lambda; P]$, for some group G, index sets I and Λ, and regular matrix P. The rows and columns of P correspond respectively to the $\mathscr{L}$-classes and $\mathscr{R}$-classes of D_k^n, and G is the group isomorphic to each of the group $\mathscr{H}$-classes of D_k^n. Since the elements of an $\mathscr{H}$-class here correspond to all the permutations of its image (all the different ways to assign the k image points to the k classes of the kernel) this group is isomorphic to the symmetric group $\mathcal{S}_k$.

To consider the congruences of $\overline{D_k^n}$, we first recognise the universal congruence $\nabla_{\overline{D_k^n}}$. All the other congruences are in bijective correspondence with the linked triples of $\overline{D_k^n}$. Recall the definition of a linked triples $(N, \mathcal{S}, \mathcal{T})$, from Definition 3.8 – that is, a normal subgroup $N \trianglelefteq G$, an equivalence relation $\mathcal{S}$ on I and an equivalence relation $\mathcal{T}$ on Λ, such that the following are satisfied:

(i) $\mathcal{S} \subseteq \varepsilon_I$, where $\varepsilon_I = \{(i,j) \in I \times I \mid \forall \lambda \in \Lambda : p_{\lambda i} = 0 \iff p_{\lambda j} = 0\}$;

(ii) $\mathcal{T} \subseteq \varepsilon_\Lambda$, where $\varepsilon_\Lambda = \{(\lambda,\mu) \in \Lambda \times \Lambda \mid \forall i \in I : p_{\lambda i} = 0 \iff p_{\mu i} = 0\}$;

(iii) For all $i, j \in I$ and $\lambda, \mu \in \Lambda$ such that $p_{\lambda i}, p_{\lambda j}, p_{\mu i}, p_{\mu j} \neq 0$ and either $(i,j) \in \mathcal{S}$ or $(\lambda, \mu) \in \mathcal{T}$, we have that $q_{\lambda\mu ij} \in N$, where

$$q_{\lambda\mu ij} = p_{\lambda i} p_{\mu i}^{-1} p_{\mu j} p_{\lambda j}^{-1}.$$

We shall first find all the triples which satisfy conditions (i) and (ii), and then we shall show that in this case all of them satisfy condition (iii).

First, we should observe that an element $p_{\lambda i}$ is non-zero if and only if the corresponding $\mathscr{H}$-class is a group. To see this, let us denote the $\mathscr{H}$-class as $H_{\lambda i}$ and recall Proposition 1.55. First assume that $p_{\lambda i} \neq 0$: this gives us an idempotent $(i, p_{\lambda i}^{-1}, \lambda) \in H_{\lambda i}$, which shows that $H_{\lambda i}$ is a group. Conversely, assume that $p_{\lambda i} = 0$: any two elements (i, x, λ) and (i, y, λ) from $H_{\lambda i}$ multiply to give 0, violating closure, so $H_{\lambda i}$ is not a group.

By Lemma 6.3 we can see that for any pair of columns $i, j \in I$ there exists a row $\lambda \in \Lambda$ such that $p_{\lambda i} \neq 0 = p_{\lambda j}$. Hence $\varepsilon_I = \Delta_I$. Similarly, in the limited case that $k > 1$, Lemma 6.3 gives us that for any pair of rows $\lambda, \mu \in \Lambda$ there exists a column $i \in I$ such that $p_{\lambda i} \neq 0 = p_{\mu i}$. Hence if $k > 1$ then we have $\varepsilon_\Lambda = \Delta_\Lambda$.

Linked triples for rank 1

First let us consider the linked triples of $\overline{D_1^n}$. Since this $\mathscr{D}$-class consists of the transformations with rank 1, its elements have n possible images,

$$\{1\}, \{2\}, \ldots, \{n\}$$

and only one possible kernel,

$$\big\{\{1, \ldots, n\}\big\}.$$

Hence the matrix P of $\overline{D_1^n}$ has n rows and 1 column. Since $\overline{D_1^n}$ is simple, this means that it is a right zero semigroup. Later in this chapter we will see that every equivalence on a right zero semigroup is a congruence (see Theorem 6.16). For now we will classify the congruences using linked triples. Every element in D_1^n has the form

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ i & i & \cdots & i \end{pmatrix}$$

for some $i \in \{1, \dots, n\}$, so each element is an idempotent in its own $\mathscr{H}$-class. Hence each $\mathscr{H}$-class is a group, so the matrix P has no zeroes, and $\varepsilon_\Lambda = \Lambda \times \Lambda$. The underlying group G of the Rees 0-matrix semigroup $\overline{D_1^n}$ must be trivial, since each $\mathscr{H}$-class contains just one element.

Taking all this information together, we can classify all the triples $(N, \mathcal{S}, \mathcal{T})$ which satisfy conditions (i) and (ii) as follows:

- N must be a normal subgroup of the trivial group – hence N is $\{\mathrm{id}\}$, the trivial group itself.
- $\mathcal{S}$ must be a subset of the trivial relation Δ_I – hence $\mathcal{S} = \Delta_I$;
- $\mathcal{T}$ may be any equivalence on Λ.

This gives us all triples of the form $(\{\mathrm{id}\}, \Delta_I, \mathcal{T})$, where $\mathcal{T}$ can be any partition of the n rows in Λ. The number of these triples is the Bell number B_n. Now consider condition (iii): since the underlying group of $\overline{D_1^n}$ is trivial, and our chosen normal subgroup N is also trivial, we have that any four nonzero elements from the matrix P must multiply together to give the identity id, which will always be in N. Hence all the triples described are *linked*, and there are B_n of them.

Linked triples for rank 2 and higher

Now let us consider the linked triples of $\overline{D_k^n}$ for $k \geq 2$. By Lemma 6.3 we know that $\varepsilon_I = \Delta_I$ and $\varepsilon_\Lambda = \Delta_\Lambda$, so any triple satisfying conditions (i) and (ii) must have the form

$$(N, \Delta_I, \Delta_\Lambda)$$

with freedom only in the choice of a normal subgroup N of G. We may write this simply as (N, Δ, Δ) for brevity. This underlying group G is, as stated above, isomorphic to the symmetric group $\mathcal{S}_k$, so N can be chosen to be any normal subgroup of $\mathcal{S}_k$.

The only normal subgroups of $\mathcal{S}_k$ for $k = 3$ and $k \geq 5$ are the trivial group, the alternating group $\mathcal{A}_k$, and the symmetric group $\mathcal{S}_k$ itself. For $k = 2$ we have $\{\mathrm{id}\} = \mathcal{A}_2 < \mathcal{S}_2$, and for $k = 4$ alone we must add a fourth normal subgroup, $K_4 = \langle (1\ 2)(3\ 4), (1\ 3)(2\ 4) \rangle$.

To see that all these triples also fulfil condition (iii) we use the triviality of the relations $\mathcal{S} = \Delta_I$ and $\mathcal{T} = \Delta_\Lambda$. Observe that $(i, j) \in \mathcal{S}$ only if $i = j$, and $(\lambda, \mu) \in \mathcal{T}$ only if $\lambda = \mu$. In the former case, we have

$$q_{\lambda\mu ij} = p_{\lambda i}(p_{\mu i}^{-1} p_{\mu i}) p_{\lambda i}^{-1} = p_{\lambda i} p_{\lambda i}^{-1} = \mathrm{id} \in N,$$

and in the latter case,

$$q_{\lambda\mu ij} = (p_{\lambda i} p_{\lambda i}^{-1})(p_{\lambda j} p_{\lambda j}^{-1}) = \mathrm{id} \cdot \mathrm{id} = \mathrm{id} \in N.$$

Hence condition (iii) is fulfilled and all of the triples described are *linked.*

Numbers of Congruence Classes

The universal congruence $\nabla_{\overline{D_k^n}}$ has, by definition, one congruence class. Any other congruence on a principal factor has a linked triple $(N, \mathcal{S}, \mathcal{T})$, and we can use this triple to calculate the number of congruence classes. Each non-zero class corresponds to a triple $(Nx, [i]_{\mathcal{S}}, [\lambda]_{\mathcal{T}})$ consisting of a coset of N, a class of $\mathcal{S}$ and a class of $\mathcal{T}$, as described in [Tor14a, Theorem 3.2]. Hence the total number of classes is equal to the product of the index $|G : N|$, the number of classes of $\mathcal{S}$, and the number of classes of $\mathcal{T}$, plus 1 for the universal congruence. Many of our congruences have $\mathcal{S} = \Delta_I$ and $\mathcal{T} = \Delta_\Lambda$; the total number of classes for these congruences will be

$$|G : N| \cdot |I| \cdot |\Lambda| + 1 \quad = \quad \big|\mathcal{S}_k : N\big| S(n,k) \binom{n}{k} + 1.$$

Summary of Results

We can now describe all the congruences of the principal factors $\overline{D_k^n}$ of the full transformation monoid $\mathcal{T}_n$. If $(N, \mathcal{S}, \mathcal{T})$ is a linked triple on $\overline{D_k^n}$, then let $[N, \mathcal{S}, \mathcal{T}]$ be the non-universal congruence associated with that triple. For brevity, let $[N] = [N, \Delta_I, \Delta_\Lambda]$ and let $h_k^n = S(n,k) \cdot \binom{n}{k}$, the number of $\mathscr{H}$-classes in D_k^n.

Theorem 6.4. *The congruences of $\overline{D_k^n}$ are shown in Table 6.5.*

k	**Congruences of $\overline{D_k^n}$**	**Number**	**Number of classes**
1	$[\{\mathrm{id}\}, \Delta_I, \mathcal{T}] (\forall \mathcal{T})$	B_n	from 1 to n
2	$[\{\mathrm{id}\}], [\mathcal{S}_2], \nabla$	3	$2h_2^n + 1, h_2^n + 1, 1$
3	$[\{\mathrm{id}\}], [\mathcal{A}_3], [\mathcal{S}_3], \nabla$	4	$6h_3^n + 1, 2h_3^n + 1, h_3^n + 1, 1$
4	$[\{\mathrm{id}\}], [K_4], [\mathcal{A}_4], [\mathcal{S}_4], \nabla$	5	$24h_4^n + 1, 6h_4^n + 1, 2h_4^n + 1, h_4^n + 1, 1$
≥ 5	$[\{\mathrm{id}\}], [\mathcal{A}_k], [\mathcal{S}_k], \nabla$	4	$k!h_k^n + 1, 2h_k^n + 1, h_k^n + 1, 1$

Table 6.5: Congruences of the principal factors of $\mathcal{T}_n$.

We can now summarise the numbers of congruence classes for some small values of n. Table 6.6 gives the number of classes of each congruence on each principal factor $\overline{D_k^n}$ of $\mathcal{T}_n$, for n up to 7. Note that for $k = 1$ only the set of distinct values has been given, since there are B_n different congruences which must be considered.

6.1.3 Other semigroups

Now that we have considered the principal factors of the full transformation monoid, we can go on to consider the principal factors of some other semigroups related to $\mathcal{T}_n$, and classify their congruences. The proofs are broadly similar to those for $\mathcal{T}_n$, so we will only summarise the arguments, highlighting the parts where they differ from those in Section 6.1.2. We start by extending our consideration of transformations to partial transformations and then to partial permuations; then we consider the three corresponding order-preserving submonoids.

Partial transformation monoid $\mathcal{PT}_n$

Recall that $\mathcal{PT}_n$ is the monoid of all partial transformations on the set $\mathbf{n} = \{1, \ldots, n\}$, that is, all transformations on some subset of $\mathbf{n}$. In many respects a partial transformation behaves like a transformation: it has an image, a rank (the size of the image), and a kernel. However, we should also consider a partial transformation's domain: the set of points which it maps. The

	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$k=1$	1	1 to 2	1 to 3	1 to 4	1 to 5
$k=2$	–	3, 2, 1	19, 10, 1	85, 43, 1	301, 151, 1
$k=3$	–	–	7, 3, 2, 1	145, 49, 25, 1	1501, 501, 251, 1
$k=4$	–	–	–	25, 7, 3, 2, 1	1201, 301, 101, 51, 1
$k=5$	–	–	–	–	121, 3, 2, 1

	$n=6$	$n=7$
$k=1$	1 to 6	1 to 7
$k=2$	931, 466, 1	2647, 1324, 1
$k=3$	10801, 3601, 1801, 1	63211, 21071, 10536, 1
$k=4$	23401, 5851, 1951, 976, 1	294001, 73501, 24501, 12251, 1
$k=5$	10801, 181, 91, 1	352801, 5881, 2941, 1
$k=6$	721, 3, 2, 1	105841, 295, 148, 1
$k=7$	–	5041, 3, 2, 1

Table 6.6: Number of classes of the congruences on the principal factors of $\mathcal{T}_n$, for n up to 7.

kernel is a partition of the domain. Using these definitions, the Green's relations of $\mathcal{PT}_n$ are described in the same way as those of $\mathcal{T}_n$: the $\mathscr{D}$-classes are determined by rank, the $\mathscr{L}$-classes by image, the $\mathscr{R}$-classes by kernel, and the $\mathscr{H}$-classes by image and kernel. Idempotents are also described in the same way: a partial transformation $\alpha \in \mathcal{PT}_n$ is an idempotent if and only if its image is a cross-section of its kernel.

The $\mathscr{D}$-classes of $\mathcal{PT}_n$ are somewhat different from $\mathcal{T}_n$. Firstly, there exists an element $0 = \begin{pmatrix} 1 & 2 & \cdots & n \\ - & - & \cdots & - \end{pmatrix}$ with rank 0, so we have an additional $\mathscr{D}$-class D_0^n. For a given rank k, there are still $\binom{n}{k}$ possible images with size k, so the $\mathscr{D}$-class D_k^n has $\binom{n}{k}$ $\mathscr{L}$-classes, like $\mathcal{T}_n$. However, the possibility of points not being in the domain means that there are not just $S(n,k)$ possible kernels, but $S(n+1,k+1)$; the intuition behind this is that, instead of considering all k-partitions of $\{1,\ldots,n\}$, we are considering all $k+1$-partitions of $\{1,\ldots,n+1\}$, where the class containing $n+1$ represents those points outside the domain.

Lemma 6.3 holds true for $\mathcal{PT}_n$ – in fact, both parts hold even when $k=0$ or 1. For $k=0$ we simply observe that D_0^n is trivial, and the two statements follow. The proof given for (i) is sufficient for $\mathcal{PT}_n$ when $k \geq 1$, so (i) is proven. For (ii), the proof given is only sufficient for $\mathcal{PT}_n$ when $k \geq 2$. For $k=1$, it is proven as follows. We must have $I_1 = \{i\}$ and $I_2 = \{j\}$ with $i \neq j$; simply let $P = \big\{\{i\}\big\}$, a cross-section of I_1 but not I_2. This is not possible in $\mathcal{T}_n$ because any kernel has to contain all the elements of $\mathbf{n}$ somewhere; but by missing out points from the domain, it is possible in $\mathcal{PT}_n$.

Since Lemma 6.3 holds for all k, we have $\varepsilon_I = \Delta_I$ and $\varepsilon_\Lambda = \Delta_\Lambda$ in each principal factor $\overline{D_k^n}$. Hence every linked triple on $\overline{D_k^n}$ must have the form (N, Δ, Δ). Each group $\mathscr{H}$-class in D_k^n is isomorphic to $\mathcal{S}_k$, since it corresponds to all the ways of mapping the k classes of the kernel onto the k points in the image. Hence the possible normal subgroups N are again the normal subgroups of $\mathcal{S}_k$. The congruences are summarised in Table 6.7, where h_k^n is equal to

$$S(n+1,k+1)\binom{n}{k},$$

the number of $\mathscr{H}$-classes in D_k^n.

k	Congruences of $\overline{D_k^n}$	Number	Number of classes
0	$[\{\mathrm{id}\}]$	1	1
1	$[\{\mathrm{id}\}], \nabla$	2	$h_1^n+1, 1$
2	$[\{\mathrm{id}\}], [\mathcal{S}_2], \nabla$	3	$2h_2^n+1, h_2^n+1, 1$
3	$[\{\mathrm{id}\}], [\mathcal{A}_3], [\mathcal{S}_3], \nabla$	4	$6h_3^n+1, 2h_3^n+1, h_3^n+1, 1$
4	$[\{\mathrm{id}\}], [K_4], [\mathcal{A}_4], [\mathcal{S}_4], \nabla$	5	$24h_4^n+1, 6h_4^n+1, 2h_4^n+1, h_4^n+1, 1$
≥ 5	$[\{\mathrm{id}\}], [\mathcal{A}_k], [\mathcal{S}_k], \nabla$	4	$k!h_k^n+1, 2h_k^n+1, h_k^n+1, 1$

Table 6.7: Congruences of the principal factors of $\mathcal{PT}_n$ or $\mathcal{I}_n$.

Symmetric inverse monoid $\mathcal{I}_n$

Recall that the symmetric inverse monoid $\mathcal{I}_n$ consists of all partial permutations on the set $\mathbf{n}$; that is, $\mathcal{I}_n$ is the submonoid of $\mathcal{PT}_n$ consisting of the injective maps. The Green's relations of $\mathcal{I}_n$ are determined by rank, image and kernel, as for $\mathcal{PT}_n$, but we can think of the $\mathscr{R}$ relation in a slightly simpler way. Since each element of $\mathcal{I}_n$ is a partial permutation, its kernel must be the diagonal relation on the domain; hence, two elements are $\mathscr{R}$-related if and only if they have the same domain. This creates a certain symmetry between the $\mathscr{L}$ and $\mathscr{R}$ relations: if an element is written in two-row notation, the set of points in the top row determine its $\mathscr{R}$-class, and the set of points in the bottom row determine its $\mathscr{L}$-class.

This symmetry makes the classification of the principal factors' congruences quite straightforward. The $\mathscr{D}$-class D_k^n of elements with rank k contains $\binom{n}{k}$ $\mathscr{L}$-classes and $\binom{n}{k}$ $\mathscr{R}$-classes. The idempotents of $\mathcal{I}_n$ are simply the identity maps (that is, the elements α such that $i\alpha = i$ for all $i \in \operatorname{dom}\alpha$) so an $\mathscr{H}$-class is a group if and only if its image and its domain are equal. Hence each $\mathscr{L}$-class and each $\mathscr{R}$-class contains precisely one group $\mathscr{H}$-class. This is enough to prove the whole of Lemma 6.3, for all k from 0 to n. Hence, as for $\mathcal{PT}_n$, all linked triples on $\overline{D_k^n}$ are of the form (N, Δ, Δ).

There are $k!$ elements with a given image and domain of size k, and if the image and domain are equal they form a group isomorphic to $\mathcal{S}_k$, so like $\mathcal{PT}_n$ we have that the choices for N are all the normal subgroups of $\mathcal{S}_k$. The result is that the congruences of the principal factors of $\mathcal{I}_n$ have the same description as those of the principal factors of $\mathcal{PT}_n$. They can be seen in Table 6.7, where the number of $\mathscr{H}$-classes h_k^n is in this case given by $\binom{n}{k}^2$.

Order-preserving partial transformations $\mathcal{PO}_n$

Recall that a partial transformation $\alpha \in \mathcal{PT}_n$ is called *order-preserving* if, for points $i, j \in \operatorname{dom}\alpha$, we have $i \leq j$ if and only if $i\alpha \leq j\alpha$. The order-preserving partial transformations in $\mathcal{PT}_n$ form a submonoid which we call $\mathcal{PO}_n$.

The Green's relations have the same description as in $\mathcal{PT}_n$, being based on rank, image and kernel. However, some partitions of $\mathbf{n}$ do not occur as kernels in $\mathcal{PO}_n$, since they cannot preserve order. Let P be a partition of $\mathbf{n}$ which contains three points $i < j < k$ such that i and k are in the same kernel class, and j is in a different kernel class. Any partial transformation α with kernel P cannot preserve order, since it must have either $j\alpha < i\alpha = k\alpha$ or $i\alpha = k\alpha < j\alpha$. A partition is a valid kernel for $\mathcal{PO}_n$ if and only if it observes the following rule: a point i is either in the same class as $i - 1$, or it is the lowest point in its class. Hence, there are not $S(n+1, k+1)$ $\mathscr{R}$-classes in D_k^n, as there are for $\mathcal{PT}_n$. The actual number of $\mathscr{R}$-classes in D_k^n is given by

$$\sum_{i=k}^{n} \binom{n}{i} \binom{i-1}{k-1},$$

as shown in [LU04, Lemma 4.1]. This can be understood in the following way. Since an element in D_k^n has rank k, the domain can have any size from k to n. Given a domain size i, there are $\binom{n}{i}$ choices for the domain. Once we have chosen a domain, we must split the i domain points into classes. By the above description of a valid kernel, this involves simply choosing which i of the n points are the lowest in their kernel-class. Point 1 must be lowest, so we have $\binom{n-1}{i-1}$ choices. The exception to this rule is D_0^n, which simply has one $\mathscr{R}$-class.

Idempotents have the same characterisation as for $\mathcal{PT}_n$. Lemma 6.3 holds for all k from 0 to n, as follows. D_0^n is trivial, so both statements hold for $k = 0$. The proof given for (i) is sufficient in this case for all $k \geq 1$. To prove (ii) for $k = 1$, we use the same approach described for $\mathcal{PT}_n$. To prove (ii) for $k \geq 2$, let I_1 and I_2 be the images of L_1 and L_2 respectively; if we take the kernel Δ_{I_1}, then I_1 is a cross-section of it but I_2 is not, so the $\mathscr{R}$-class corresponding to that kernel satisfies the requirement.

Perhaps the most important difference between $\mathcal{PO}_n$ and $\mathcal{PT}_n$ is that in $\mathcal{PO}_n$ a given kernel and image determines a single element, not $k!$ elements, since order must be preserved. This means that the underlying group of D_k^n is not $\mathcal{S}_k$, but simply the trivial group $\{\mathrm{id}\}$. This result puts the principal factors $\overline{D_k^n}$ into the category of *congruence-free* semigroups by Proposition 6.11, meaning that the only congruences on $\overline{D_k^n}$ are Δ and ∇. Indeed, the only linked triple of $\overline{D_k^n}$ is $(\{\mathrm{id}\}, \Delta, \Delta)$, corresponding to the trivial congruence. This result is summarised in Table 6.8, where h_k^n is the number of $\mathscr{H}$-classes in D_k^n, given by

$$\binom{n}{k} \sum_{i=k}^{n} \binom{n}{i} \binom{i-1}{k-1}.$$

k	**Congruences of $\overline{D_k^n}$**	**Number**	**Number of classes**
0	$[\{\mathrm{id}\}]$	1	1
≥ 1	$[\{\mathrm{id}\}], \nabla$	2	$h_k^n + 1, 1$

Table 6.8: Congruences of the principal factors of $\mathcal{PO}_n$ or $\mathcal{POI}_n$.

Order-preserving partial permutations $\mathcal{POI}_n$

Next we consider the order-preserving partial permuations, which form the monoid $\mathcal{POI}_n = \mathcal{PO}_n \cap \mathcal{I}_n$. Like $\mathcal{I}_n$, the $\mathscr{L}$ and $\mathscr{R}$ relations are determined by image and domain, and so

Lemma 6.3 is proven in the same way as for $\mathcal{I}_n$, and applies to all k from 0 to n. Since the kernel of a partial permutation is always a diagonal relation, we do not encounter any kernels which cannot preserve order; hence the Green's class structure of $\mathcal{POI}_n$ is isomorphic to that of $\mathcal{I}_n$. In particular, there are $\binom{n}{k}^2$ $\mathscr{H}$-classes in D_k^n. The main difference between $\mathcal{POI}_n$ and $\mathcal{I}_n$ is that each $\mathscr{H}$-class contains just one element, since each domain–image pair defines only one order-preserving element. Hence the underlying group of $\overline{D_k^n}$ is the trivial group $\{\mathrm{id}\}$, and so $\mathcal{POI}_n$ is congruence-free like $\mathcal{PO}_n$. This information is summarised in Table 6.8, where in this case the number of $\mathscr{H}$-classes h_k^n is equal to $\binom{n}{k}^2$.

Order-preserving transformations $\mathcal{O}_n$

Finally, we consider the submonoid of $\mathcal{T}_n$ consisting of the order-preserving transformations, $\mathcal{O}_n = \mathcal{T}_n \cap \mathcal{PO}_n$. Since $\mathcal{O}_n$ consists of transformations, an element's kernel includes every point in $\mathbf{n}$, as for $\mathcal{T}_n$. Its Green's relations $\mathscr{L}$, $\mathscr{R}$ and $\mathscr{D}$ are again based on image, kernel and rank, as for $\mathcal{T}_n$, so we do not have a $\mathscr{D}$-class D_0^n. Some $\mathscr{R}$-classes in $\mathcal{T}_n$ are not present $\mathcal{O}_n$, since certain kernels cannot preserve order: like in $\mathcal{PO}_n$, the valid kernels are those such that a point i either is in the same class as $i-1$, or is minimal in its class. Hence D_k^n contains $\binom{n-1}{k-1}$ $\mathscr{R}$-classes, since our only choice is which $k-1$ of the $n-1$ points in $\mathbf{n}$ are minimal, apart from 1. There are still $\binom{n}{k}$ $\mathscr{L}$-classes, as in $\mathcal{T}_n$. Lemma 6.3 applies to $\mathcal{O}_n$ in the same way as it applies to $\mathcal{T}_n$, with statement (ii) only applying when $k > 1$. Hence the linked triples for $k \geq 2$ have the form (N, Δ, Δ) while the linked triples for $k = 1$ have the form $(N, \Delta, \mathcal{T})$ for other possible values of $\mathcal{T}$.

Two elements are $\mathscr{H}$-related if and only if they share the same image and kernel. Since all elements are order-preserving, there is only one choice of element for a given image and kernel; hence $\mathcal{O}_n$ is $\mathscr{H}$-trivial. So the only choice of N for linked triples is the trivial group $\{\mathrm{id}\}$. Hence, when $k \geq 2$ the only linked triple on $\overline{D_k^n}$ is $(\{\mathrm{id}\}, \Delta, \Delta)$, corresponding to the trivial congruence; if $k = 1$, as for $\mathcal{T}_n$, we have a linked triple $(\{\mathrm{id}\}, \Delta, \mathcal{T})$ for any relation $\mathcal{T}$ on the n $\mathscr{L}$-classes of D_1^n. The congruences are summarised in Table 6.9, where the number of $\mathscr{H}$-classes h_k^n is given by $\binom{n}{k}\binom{n-1}{k-1}$.

k	**Congruences of $\overline{D_k^n}$**	**Number**	**Number of classes**
1	$[\{\mathrm{id}\}, \Delta_I, \mathcal{T}] (\forall \mathcal{T})$	B_n	from 1 to n
≥ 2	$[\{\mathrm{id}\}], \nabla$	2	$h_k^n + 1, 1$

Table 6.9: Congruences of the principal factors of $\mathcal{O}_n$.

6.1.4 Further work

We have considered the monoids of partial transformations $\mathcal{PT}_n$, transformations $\mathcal{T}_n$, and partial permutations $\mathcal{I}_n$, and for each of those monoids we have considered the submonoid of order-preserving elements. In the future, the ideas presented here could perhaps be extended to other similar submonoids of these three, such as the following submonoids, considered in [EKMW18, §1.2]:

- the monoids of order-preserving *or order-reversing* elements: $\mathcal{POD}_n$, $\mathcal{OD}_n$, and $\mathcal{PODI}_n$ respectively;

- the monoids of *orientation*-preserving elements: $\mathcal{POP}_n$, $\mathcal{OP}_n$, and $\mathcal{POPI}_n$ respectively;
- the monoids of orientation-preserving or or orientation-reversing elements: $\mathcal{POR}_n$, $\mathcal{OR}_n$, and $\mathcal{PORI}_n$ respectively.

We could also consider some important monoids which do not consist of partial transformations. After the results in Chapter 5, it would be interesting to learn about the congruences of the principal factors of the Motzkin monoid and other bipartition monoids such as $\mathcal{P}_n$. These monoids are not as straightforward as the ones we have so far considered; certainly, identifying the idempotents in a semigroup of bipartitions is more complicated than for partial transformations [DEE$^+$15, Theorem 5]. However, it is possible that their principal factors' congruences could be classified in a similar way.

6.2 The number of congruences of a semigroup

In light of the methods in Chapters 2 and 4 to compute the congruences of a semigroup, and their implementation in libsemigroups and the Semigroups package, we may be interested in the number of congruences a given semigroup possesses. At the very least, a semigroup S must have congruences Δ_S and ∇_S, which are equal if and only if $|S| = 1$; so any semigroup has at least one congruence. For an upper bound, consider that a congruence is an equivalence; the number of equivalences on a finite set is given by the Bell numbers [OEIS, A000110], so a finite semigroup S cannot have more congruences than the Bell number $B_{|S|}$. All finite semigroups have a number of congruences between these two bounds, but the precise number depends on the structure of the semigroup.

In this section, we consider how many congruences there are on various semigroups, showing some computational results on small semigroups, as well as proving some more general results.

6.2.1 Congruence-free semigroups

The notion of a *congruence-free* semigroup has long been understood, but is presented here for completeness. Note that any semigroup must have at least the trivial and universal congruences Δ and ∇; the definition of a congruence-free semigroup is as follows.

Definition 6.10. A semigroup S is **congruence-free** if it has no congruences other than Δ_S and ∇_S.

By the above definition, any congruence-free semigroup has 2 congruences, except the trivial semigroup, which has only 1 congruence. Note that the language used here differs from that used in group theory: if a group is congruence-free (and therefore has no proper non-trivial normal subgroups) it is called a *simple group*. This is not to be confused with the concept of a *simple semigroup* (Definition 3.3).

It is relatively easy to determine whether a finite semigroup is congruence-free, using the following theorem based partly on material in [How95, §3.7]. A full proof is included in [Tor14a, Chapter 5].

Proposition 6.11. *A finite semigroup S is congruence-free if and only if one of the following holds:*

(i) *S has no more than 2 elements;*

(ii) *S is a simple group;*

(iii) *S is isomorphic to a Rees 0-matrix semigroup $\mathcal{M}^0[G; I, \Lambda; P]$ where G is the trivial group, and P is regular, with all its rows pairwise distinct and all its columns pairwise distinct.*

6.2.2 Congruence-full semigroups

We now define a new concept: a *congruence-full* semigroup, by analogy with the *congruence-free* semigroups described in the last section.

Definition 6.12. A semigroup S is **congruence-full** if every equivalence relation on S is a congruence.

Since the number of equivalences on a finite set is given by the sequence of Bell numbers $(B_n)_{n\in\mathbb{N}}$, we can say that a finite semigroup is congruence-full if and only if it has precisely B_n congruences, where n is the size of the semigroup.

We have already classified all the finite congruence-free semigroups in Proposition 6.11. In this section we explore finite congruence-full semigroups, culminating in a complete classification in Theorem 6.16. First we need to build up some knowledge about the Green's relations of congruence-full semigroups.

Lemma 6.13. *A finite congruence-full semigroup of size greater than 2 has $\mathscr{H}$-trivial minimal ideal.*

Proof. Let S be a finite semigroup with more than 2 elements, and let M be its minimal ideal. Since M is simple, every $\mathscr{H}$-class of M is a group. We will proceed by considering possible sizes of the $\mathscr{H}$-classes of M, and showing that any $\mathscr{H}$-class size greater than 1 implies that S is not congruence-full.

Firstly, let H be an $\mathscr{H}$-class in M with at least 3 elements. Let 1_H be the group identity of H, and let $g, h \in H \setminus \{1_H\}$ with $g \neq h$. Now let $\sim$ be $(1_H, g)^e$, the equivalence whose only non-singleton class is $\{1_H, g\}$. Since g is not the identity, we know that $gh \neq h$. Hence we have $1_H \sim g$ but $1_H h \nsim gh$, so $\sim$ is not a congruence. Hence S is not congruence-full.

Instead, let H be an $\mathscr{H}$-class in M with precisely 2 elements. Since $|S| \geq 3$ we know that $S \setminus H$ is non-empty. If there exists some $x \in S \setminus M$, then let $h \in H \setminus \{1_H x\}$, and let $\sim$ be $(x, h)^e$; since $h \neq 1_H x$ we have $x \sim h$ but $1_H x \nsim 1_H h$ (since $(1_H x, 1_H h)$ is not equal to (x, h) or (h, x) and is not reflexive) so $\sim$ is not a congruence. If on the other hand $S \setminus M$ is empty, then M must contain an $\mathscr{H}$-class other than H. Choose some $x \in M \setminus H$ such that $x \mathrel{\mathscr{L}} 1_H$ (if this is not possible, we can choose x such that $x \mathrel{\mathscr{R}} 1_H$, and a similar argument holds). Let $h \in H \setminus \{1_H\}$, and let $\sim$ be $(x, 1_H)^e$. We have $xh \mathrel{\mathscr{R}} x \mathrel{\not\mathscr{R}} h$, so $xh \mathrel{\not\mathscr{R}} h$ and in particular $xh \neq h$. Hence $x \sim 1_H$ but $xh \nsim 1_H h$, so $\sim$ is not a congruence. Either of these cases shows that S is not congruence-full. $\square$

Lemma 6.14. *A finite congruence-full semigroup of size greater than 2 has a minimal ideal which is either $\mathscr{L}$-trivial or $\mathscr{R}$-trivial.*

Proof. Let S be a congruence-full semigroup with more than 2 elements, with a minimal ideal M which is neither $\mathscr{L}$-trivial nor $\mathscr{R}$-trivial.

We know by Lemma 6.13 that M is $\mathscr{H}$-trivial. Since M is simple and $\mathscr{H}$-trivial, it is a rectangular band (see Definition 1.59). Let $x_{11}, x_{12}, x_{22} \in M$ be pairwise distinct elements with $x_{11} \mathscr{R} x_{12} \mathscr{L} x_{22}$, and let $\sim$ be the relation $(x_{11}, x_{22})^e$. Since M is a rectangular band, we have $x_{11}x_{22} = x_{12}$ and $x_{11}x_{11} = x_{11}$. Hence $x_{11} \sim x_{22}$ but $x_{11}x_{11} \nsim x_{11}x_{22}$, and so $\sim$ is not a congruence. This means that S is not congruence-full, a contradiction. □

Lemma 6.15. *A finite congruence-full semigroup of size greater than* 2 *is either simple or a zero semigroup.*

Proof. Let S be a finite congruence-full semigroup with more than 2 elements, with minimal ideal M. Let us assume S is not simple; this means that $S \setminus M$ is non-empty. By Lemma 6.14, M is either $\mathscr{L}$-trivial or $\mathscr{R}$-trivial; without loss of generality let us assume that M is $\mathscr{L}$-trivial (a similar argument applies for $\mathscr{R}$-triviality). We will start by proving that S contains a zero, and then we will go on to prove that S is a zero semigroup. Firstly, aiming for a contradiction, let us assume that $|M| > 1$.

If $S \setminus M$ contains an idempotent, call it x. Choose $m, n \in M$ with $m \neq n$. Now, either $mx = m$ or $mx \neq m$. If $mx = m$, then let $\sim$ be $(n, x)^e$: since by $\mathscr{L}$-triviality $mn = n$, and since $mx = m$, we have $n \sim x$ but $mn \nsim mx$, so $\sim$ is not a congruence, a contradiction. If on the other hand $mx \neq m$, then let $\sim$ be $(m, x)^e$: since $mx \neq x$ and $mx \neq m$ and $xx = x$, we have $m \sim x$ but $mx \nsim xx$, so $\sim$ is not a congruence, a contradiction.

If instead, $S \setminus M$ does not contain an idempotent, then there must exist some $x \in S \setminus M$ such that $x^2 \in M$. Let $m \in M \setminus \{x^2\}$ (this is possible since $|M| > 1$) and let $\sim$ be $(m, x)^e$. Since by $\mathscr{L}$-triviality $xm = m$, we have $m \sim x$ but $xm \nsim xx$, so $\sim$ is not a congruence, a contradiction.

We have now shown that $|M| > 1$ violates the condition that S is congruence-full. Hence the minimal ideal M must contain precisely one element, 0: we have $0x = x0 = 0$ for any $x \in S$, so 0 is a zero for S. Next we will show that S is a zero semigroup, i.e. that $xy = 0$ for all $x, y \in S$. Clearly if x or y is 0 then $xy = 0$.

Let $x, y \in S \setminus \{0\}$ with $x \neq y$. The product xy cannot be equal to both x and y, so without loss of generality let us assume that $xy \neq x$. Assume, aiming for a contradiction, that $xy \neq 0$. Let $\sim$ be the relation $(x, 0)^e$; since $0y = 0$ and $xy \neq x$ we have $x \sim 0$ but $xy \nsim 0y$, so $\sim$ is not a congruence, a contradiction. Hence for distinct $x, y \in S$ we have $xy = 0$.

It only remains to consider whether $x^2 = 0$ for every $x \in S$. Let $x \in S \setminus \{0\}$ and assume, aiming for a contradiction, that $x^2 \neq 0$. Let $y \in S \setminus \{0, x\}$ (possible since $|S| > 2$) and let $\sim$ be $(x, y)^e$; since $xy = 0$ but $x^2 \neq 0$, we have $x \sim y$ but $xx \nsim xy$, so $\sim$ is not a congruence, a contradiction. Hence $xy = 0$ for all $x, y \in S$, so S is a zero semigroup. □

We can now state the main theorem of this section, a classification of all the finite congruence-full semigroups.

Theorem 6.16. *A finite semigroup S is congruence-full if and only if one of the following holds:*

(i) *S has no more than 2 elements;*

(ii) *S is a zero semigroup;*

(iii) *S is a left zero semigroup;*

(iv) *S is a right zero semigroup.*

Proof. First, observe that if S has 1 or 2 elements, then the only equivalences on S are Δ_S and ∇_S, both of which are congruences; hence, S is congruence-full.

Instead, let S be a finite congruence-full semigroup of size greater than 2. If S is not simple, then by Lemma 6.15 it is a zero semigroup. If S is simple, it is equal to its minimal ideal. Therefore, by Lemma 6.14, S is either $\mathscr{L}$-trivial or $\mathscr{R}$-trivial. For any simple semigroup we have $x \mathscr{R} xy \mathscr{L} y$ for all $x, y \in S$. If S is $\mathscr{L}$-trivial, then $xy = y$ for all $x, y \in S$, so S is a right zero semigroup. If S is $\mathscr{R}$-trivial, then $xy = x$ for all $x, y \in S$, so S is a left zero semigroup.

To prove the converse, we consider zero, left zero, and right zero semigroups in turn. First, let S be a zero semigroup and let $\sim$ be an equivalence relation on S. Let $x, y, s, t \in S$ such that $x \sim y$ and $s \sim t$. We have $xs = 0 = yt$, so $xs \sim yt$ and therefore $\sim$ is a congruence. Hence S is congruence-full.

Alternatively, let S be a left zero semigroup and let $\sim$ be an equivalence relation on S. Let $x, y, a \in S$ such that $x \sim y$. We have $ax = a = ay$ and $xa = x \sim y = ya$, so $ax \sim ay$ and $xa \sim ya$. Hence $\sim$ is a congruence, so S is congruence-full. A similar argument proves the statement for right zero semigroups. $\square$

Note that some semigroups of size 2 fall into categories (ii), (iii), or (iv) of the above theorem. However, some semigroups of size 2 are not zero, left zero, or right zero semigroups – for example, $1^{(1)}$, the trivial group with an identity attached – but these are still congruence-free. Hence all four cases are required.

6.2.3 Semigroups with fewer congruences

If a semigroup of finite size n is not congruence-full, it has fewer than B_n congruences. If n is 2 or 3, there exist semigroups of size n with precisely $B_n - 1$ congruences. However, for $4 \leq n \leq 7$, computational experiments show that there is no semigroup of size n with $B_n - 1$ congruences, and it seems unlikely that such a semigroup could be found for any higher n. In this section we propose a value for the second highest number of congruences possible on a semigroup of size n – that is, the highest number of congruences on a semigroup that is not congruence-full.

Conjecture 6.17. *A finite semigroup that is not congruence-full has at most $2B_{n-1}$ congruences.*

This conjecture, which does not yet have a proof, is supported by experimental investigation. An exhaustive analysis of all semigroups up to isomorphism and anti-isomorphism shows that the conjecture holds for $n \leq 7$, and also reveals a pattern in the semigroups which attain the limit. This pattern is stated in the next conjecture.

Conjecture 6.18. *Let $n > 3$. There are precisely 7 semigroups (up to isomorphism and anti-isomorphism) of size n which have $2B_{n-1}$ congruences.*

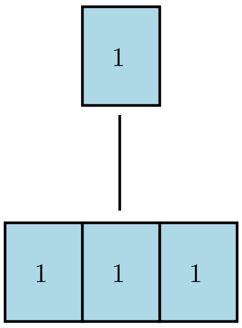


Figure 6.19: Nearly congruence-full semigroup 1. Left-zero semigroup $\mathcal{LZ}_{n-1}$ with an idempotent c appended. There is a distinguished $p \in M$ such that $cx = p$ and $xc = x$ for all $x \in M$.

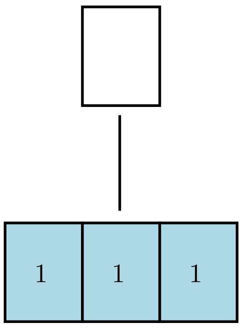


Figure 6.20: Nearly congruence-full semigroup 2. Left-zero semigroup $\mathcal{LZ}_{n-1}$ with a non-idempotent element c appended. Multiplication defined by $cx = c^2$ and $xc = x$ for all $x \in M$.

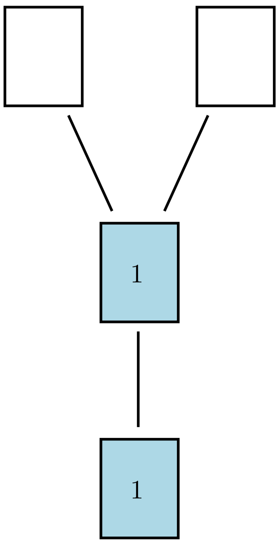


Figure 6.21: Nearly congruence-full semigroup 3. Zero semigroup $\mathcal{Z}_{n-1}$ with a zero appended.

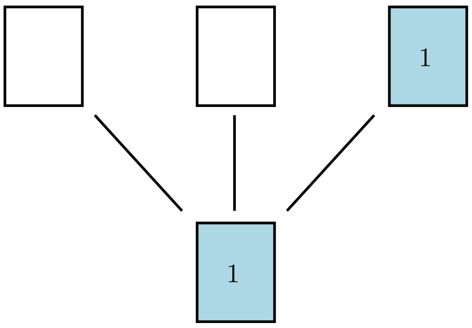


Figure 6.22: Nearly congruence-full semigroup 4. Zero semigroup $\mathcal{Z}_{n-1}$ with an idempotent c appended above the minimal ideal. Multiplication defined by $cx = xc = z$ for all $x \in \mathcal{Z}_{n-1}$.

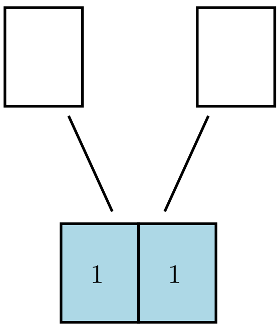


Figure 6.23: Nearly congruence-full semigroup 5. Zero semigroup $\mathcal{Z}_{n-1}$ with an idempotent c appended in the minimal ideal. Multiplication defined by $cx = c$ and $xc = z$ for all $x \in \mathcal{Z}_{n-1}$.

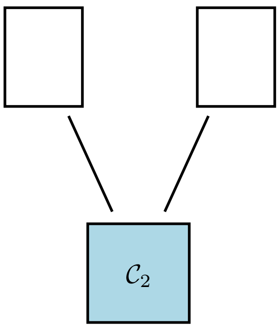


Figure 6.24: Nearly congruence-full semigroup 6. Zero semigroup $\mathcal{Z}_{n-1}$ with an element c appended in the same $\mathscr{H}$-class as z so that $\{z, c\} \cong \mathcal{C}_2$. Multiplication defined by $c^2 = z$ and $xc = cx = c$ for all $x \in \mathcal{Z}_{n-1}$.

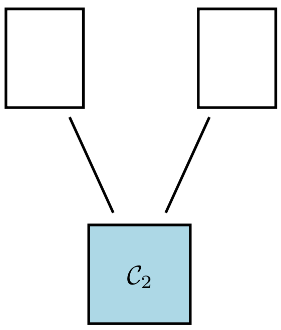


Figure 6.25: Nearly congruence-full semigroup 7. Cyclic group $\mathcal{C}_2 = \{\mathrm{id}, g\}$ with $n-2$ elements appended such that $xy = \mathrm{id}$, $x\,\mathrm{id} = \mathrm{id}\,x = g$ (for all $x, y \in S \setminus \mathcal{C}_2$).

The seven semigroups are shown and described in Figures 6.19 to 6.25, including diagrams for $n = 4$. In the descriptions, M is the minimal ideal of the semigroup. Also, when the zero semigroup with $n-1$ elements $\mathcal{Z}_{n-1}$ is a subsemigroup of S, the zero of $\mathcal{Z}_{n-1}$ is called z.

Of the seven semigroups shown, each of the first 6 contain a copy of either the zero semigroup $\mathcal{Z}_{n-1}$ or the left zero semigroup $\mathcal{LZ}_{n-1}$ as a subsemigroup. The one element outside this subsemigroup, which we will call c (the *child element*) is key to understanding why these semigroups have $2B_{n-1}$ congruences. In each semigroup, there is another element p (the *parent element*) such that an equivalence $\sim$ is a congruence if and only if $c \sim p$ or $[c]_\sim$ is a singleton. In other words, the child has to be alone or with its parent. Now, since $S \setminus \{c\}$ is congruence-full, we can take any congruence (any equivalence) $\sim$ on $S \setminus \{c\}$ and extend it to a congruence on S in two different ways: by including c as a singleton, or by including c in the same congruence class as p. Since there are B_{n-1} choices for $\sim$, this gives us precisely $2B_{n-1}$ congruences on S.

The seventh semigroup (Figure 6.25) is unique in that it does not contain $\mathcal{Z}_{n-1}$ or $\mathcal{LZ}_{n-1}$ as a subsemigroup. However, it still fulfils the *child–parent* condition above, where $c = \mathrm{id}$ and $p = g$.

Conjecture 6.18 does not have a proof, and it is certainly possible that there may be more than just these seven semigroups when $n > 7$. However, the statement is certainly true for sizes 4, 5, 6 and 7, so it seems likely that the pattern may continue. The feasibility of computational experiments for higher values of n is discussed at the end of Section 6.3.

6.3 Small semigroups

The smallsemi package [DM17] provides a library of all the semigroups of size no more than 8, up to isomorphism. Using this library, it was possible to calculate the congruences of all 1658439 semigroups of size no more than 7, revealing some interesting information about the numbers of congruences of the semigroups, as well as about the properties of those congruences. Some of these findings are presented here.

The average number of congruences on a semigroup of size n is shown in Table 6.26. Since we only have the first few values here, it is hard to make a conjecture about the growth of this sequence. However, it appears to increase rapidly, as might be expected for a value with upper bound given by the Bell numbers (see Section 6.2.2).

n	Average number
1	1.00
2	2.00
3	3.67
4	6.38
5	11.25
6	22.71
7	78.51

Table 6.26: Average number of congruences on a semigroup of size n.

It is perhaps surprising to see how many congruences on small semigroups are principal (i.e. generated by a single pair). Table 6.27 shows, for each size n, the average proportion of a semigroup's congruences which are principal. As can be seen, this proportion is very high for small semigroups, but declines greatly as we increase the size, reaching 29% for an average semigroup of size 7. Also shown is the number of semigroups whose congruences are all principal, a number which also decreases rapidly.

The number of principal congruences compares curiously to the number of Rees congruences (see Definition 1.51): Table 6.28 shows the proportion of congruences that are Rees on an average semigroup of size n, as well as the number of semigroups whose congruences are all Rees. For n from 2 to 6, there are fewer Rees congruences than principal congruences, but when $n = 7$ there are more Rees than principal; indeed, when $n = 7$ there are more than twice as many semigroups with all Rees congruences as all principal congruences. This may indicate that the proportion of principal congruences decreases faster than the rate of Rees congruences, as n grows; however, with only the first 7 values, it is difficult to reach any reliable conclusions.

There are 3,684,030,417 semigroups of size 8, more than 2000 times as many as there are semigroups of size 1 to 7. It would require a very long time to compute all the congruences of all these semigroups on current hardware, with the algorithms and implementations that have been described. However, it would be possible to calculate the congruences given enough time, particularly using a very fast computer – and since each semigroup is processed independently, it would also be trivial to split the task between multiple computers to speed up the process. It would be interesting to examine the congruences on all the semigroups of size 8, firstly to see how the trends in Tables 6.26, 6.27 and 6.28 continue, but also to test Conjectures 6.17 and 6.18.

There are 105,978,177,936,292 semigroups of size 9 [OEIS, A027851], a huge number. No computational library of all these semigroups currently exists, and if one did, it is likely that it would take an unreasonably long time to compute all their congruences with anything like the algorithms described in this thesis. A more feasible area of future work would be to consider the congruences of all small simple or 0-simple semigroups, or of all small inverse semigroups. Analysing these categories might reveal interesting trends distinct from those in the generic case.

It is worth mentioning that, up to isomorphism, almost all semigroups are 3-*nilpotent* – that is, they contain a zero, and any product of three elements is equal to zero. In [KRS76] a construction is given which allows us to enumerate all 3-nilpotent semigroups, and in [Dis10,

n	Semigroups	Semigroups with just principal congruences	Average proportion of principal congruences
1	1	1 (100%)	100%
2	5	5 (100%)	100%
3	24	21 (88%)	98%
4	188	85 (45%)	90%
5	1915	194 (10%)	77%
6	28634	300 (1.0%)	60%
7	1627672	494 (0.030%)	29%

Table 6.27: Number of principal congruences on semigroups of size n. The first column is a size n; the second column is the number of semigroups of this size up to isomorphism; the third column is the number of these semigroups that have only principal congruences; and the final column is the percentage of a semigroup's congruences that are principal, on average.

n	Semigroups	Semigroups with just Rees congruences	Average proportion of Rees congruences
1	1	1 (100%)	100%
2	5	2 (40%)	70%
3	24	6 (25%)	67%
4	188	16 (8.5%)	57%
5	1915	64 (3.3%)	49%
6	28634	239 (0.83%)	41%
7	1627672	1046 (0.064%)	31%

Table 6.28: Number of Rees congruences on semigroups of size n. The first column is a size n; the second column is the number of semigroups of this size up to isomorphism; the third column is the number of these semigroups that have only Rees congruences; and the final column is the percentage of a semigroup's congruences that are Rees, on average.

Theorem 2.3.5] this is used to produce a formula for the number of 3-nilpotent semigroups up to isomorphism. As n increases, the proportion of semigroups that are 3-nilpotent increases, until at $n = 9$ they account for over 99.9% of all the semigroups of size n: this is shown in [Dis10, Table 2.1], and can also be calculated using the appropriate entries in [DM12, Table 3] and [OEIS, A027851], or by using the smallsemi software package [DM17]. Hence, the trends shown in Tables 6.26, 6.27 and 6.28 become increasingly dominated by 3-nilpotent semigroups, and so anything we can say about a congruence on a 3-nilpotent semigroup could help explain the patterns we see. So far, little has been written about the congruences on a 3-nilpotent semigroup, but in the future it might be possible to develop a new representation for congruences on 3-nilpotent semigroups, akin to those described in Section 3.1. Such a representation could make it possible to compute a close approximation to the number of congruences, Rees congruences, and principal congruences on an average semigroup of size n without having to consider every semigroup in turn. This would be desirable, since it would allow us to extend the tables to higher values of n.

Bibliography

[ABG18] J. Araújo, W. Bentz, and G. M. S. Gomes. Congruences on direct products of transformation and matrix monoids. *Semigroup Forum*, Apr 2018.

[AHT84] M. D. Atkinson, R. A. Hassan, and M. P. Thorne. Group theory on a microcomputer. In *Computational Group Theory*, pages 275–280. London Mathematical Society, Academic Press, 1984.

[AKM14] J. Araújo, J. Konieczny, and A. Malheiro. Conjugation in semigroups. *Journal of Algebra*, 403:93–134, 2014.

[BB08] B. Bockholt and P. E. Black. ‘adjacency list representation’. in *Dictionary of Algorithms and Data Structures* [online], V. Pieterse and P. E. Black eds., 14 August 2008. `https://www.nist.gov/dads/HTML/adjacencyListRep.html` (accessed 18 May 2018).

[BC76] M. J. Beetham and C. M. Campbell. A note on the Todd–Coxeter coset enumeration algorithm. *Proceedings of the Edinburgh Mathematical Society*, 20(1):73–79, 1976.

[BCP97] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[BH97] M. Breen and D. Hume. Properties of a standard form for a boolean matrix. *Linear Algebra and its Applications*, 254(1):49–65, 1997. Proceeding of the Fifth Conference of the International Linear Algebra Society.

[C$^+$86] D. J. Collins et al. A simple presentation of a group with unsolvable word problem. *Illinois Journal of Mathematics*, 30(2):230–234, 1986.

[CH97] J. Cannon and D. Holt. Computing chief series, composition series and socles in large permutation groups. *Journal of Symbolic Computation*, 24(3):285–301, 1997.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Cij57] G. S. Cijtin. An associative calculus with an insoluble problem of equivalence. *Trudy Mat. Inst. Steklov*, 52:172–189, 1957.

[CM09] A. J. Cain and V. Maltcev. Decision problems for finitely presented and one-relation semigroups and monoids. *International Journal of Algebra and Computation*, 19(06):747–770, 2009.

[Cut01] A. Cutting. *Todd-Coxeter Methods for Inverse Monoids.* PhD thesis, University of St Andrews, 2001.

[DEE$^+$15] I. Dolinka, J. East, A. Evangelou, D. FitzGerald, N. Ham, J. Hyde, and N. Loughlin. Enumeration of idempotents in diagram semigroups and algebras. *Journal of Combinatorial Theory, Series A*, 131:119–152, 2015.

[DEG17] I. Dolinka, J. East, and R. D. Gray. Motzkin monoids and partial brauer monoids. *Journal of Algebra*, 471:251–298, 2017.

[Deh11] M. Dehn. Über unendliche diskontinuierliche gruppen. *Mathematische Annalen*, 71(1):116–144, 1911.

[Dis10] A. Distler. *Classification and enumeration of finite semigroups.* PhD thesis, University of St Andrews, 2010.

[DM12] A. Distler and J. D. Mitchell. The number of nilpotent semigroups of degree 3. *Electr. J. Comb.*, 19(2):P51, 2012.

[DM17] A. Distler and J. D. Mitchell. *Smallsemi – GAP package, Version 0.6.11*, Apr 2017.

[Dyc82] W. Dyck. Gruppentheoretische studien. *Mathematische Annalen*, 20(1):1–44, Mar 1882.

[Eas11] J. East. Generators and relations for partition monoids and algebras. *Journal of Algebra*, 339(1):1–26, 2011.

[EENFM15] J. East, A. Egri-Nagy, A. R. Francis, and J. D. Mitchell. Finite diagram semigroups: Extending the computational horizon. *arXiv preprint arXiv:1502.07150*, 2015.

[EENMP18] J. East, A. Egri-Nagy, J. D. Mitchell, and Y. Péresse. Computing finite semigroups. *Journal of Symbolic Computation*, 2018.

[EKMW18] J. East, J. Kumar, J. D. Mitchell, and W. A. Wilson. Maximal subsemigroups of finite transformation and diagram monoids. *Journal of Algebra*, 504:176–216, 2018.

[EMRT18] J. East, J. D. Mitchell, N. Ruškuc, and M. Torpey. Congruence lattices of finite diagram monoids. *Advances in Mathematics*, 333:931–1003, 2018.

[End01] H. Enderton. *A mathematical introduction to logic.* Academic press, 2001.

[Fer00] V. H. Fernandes. The monoid of all injective orientation preserving partial transformations on a finite chain. *Communications in Algebra*, 28(7):3401–3426, 2000.

[Fer01] V. H. Fernandes. The monoid of all injective order preserving partial transformations on a finite chain. *Semigroup Forum*, 62(2):178–204, Mar 2001.

[FGJ05] V. H. Fernandes, G. M. S. Gomes, and M. M. Jesus. Congruences on monoids of order-preserving or order-reversing transformations on a finite chain. *Glasgow Mathematical Journal*, 47(2):413–424, 2005.

[FGJ09] V. H. Fernandes, G. M. S. Gomes, and M. M. Jesus. Congruences on monoids of transformations preserving the orientation on a finite chain. *Journal of Algebra*, 321(3):743–757, 2009.

[FJK18] P. Fenner, M. Johnson, and M. Kambites. NP-completeness in the gossip monoid. *International Journal of Algebra and Computation*, to appear 2018.

[FL11] D. G. FitzGerald and K. W. Lau. On the partition monoid and some related semigroups. *Bulletin of the Australian Mathematical Society*, 83(2):273–288, 2011.

[For55] G. E. Forsythe. SWAC computes 126 distinct semigroups of order 4. *Proceedings of the American Mathematical Society*, 6(3):443–447, 1955.

[FP97] V. Froidure and J.-E. Pin. Algorithms for computing finite semigroups. In *Foundations of Computational Mathematics*, pages 112–126, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[GAP18] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.10.0*, 2018.

[Gar91] J. I. Garcia. The congruence extension property for algebraic semigroups. *Semigroup Forum*, 43:1–18, 1991.

[GF64] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964.

[GI91] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, September 1991.

[Göd31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, Dec 1931.

[Gre51] J. A. Green. On the structure of semigroups. *Annals of Mathematics*, 54(1):163–172, 1951.

[Gur00] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, July 2000.

[Hd14] T. Halverson and E. delMas. Representations of the rook–brauer algebra. *Communications in Algebra*, 42(1):423–443, 2014.

[HEO05] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of computational group theory.* CRC Press, 2005.

[HHKR99] G. Havas, D. F. Holt, P. E. Kenne, and S. Rees. Some challenging group presentations. *Journal of the Australian Mathematical Society. Series A. Pure Mathematics and Statistics*, 67(2):206–213, 1999.

[Hol19] D. Holt. *kbmag – GAP package, Version 1.5.7*, Feb 2019.

[How95] J. M. Howie. *Fundamentals of Semigroup Theory*, volume 12. Oxford Science Publications, 1995.

[HU73] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[Hul98] A. Hulpke. Computing normal subgroups. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 194–198. ACM, 1998.

[JMP18] J. Jonušas, J. D. Mitchell, and M. Pfeiffer. Two variants of the Froidure–Pin algorithm for finite semigroups. *Portugaliae Mathematica*, 74(3):173–200, 2018.

[Jür77] H. Jürgensen. Computers in semigroups. *Semigroup Forum*, 15(1):1–20, Dec 1977.

[KB83] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[KRS76] D. J. Kleitman, B. R. Rothschild, and J. H. Spencer. The number of semigroups of order *n*. *Proceedings of the American Mathematical Society*, 55(1):227–232, 1976.

[Lee63] J. Leech. Coset enumeration on digital computers. *Mathematical Proceedings of the Cambridge Philosophical Society*, 59(2):257–267, 1963.

[Lib53] A. E. Liber. On symmetric generalized groups. *Matematicheskii Sbornik*, 75(3):531–544, 1953.

[LS99] T. Lavers and A. Solomon. The endomorphisms of a finite chain form a Rees congruence semigroup. *Semigroup Forum*, 59(2):167–170, Sep 1999.

[LU04] A. Laradji and A. Umar. Combinatorial results for semigroups of order-preserving partial transformations. *Journal of Algebra*, 278(1):342–359, 2004.

[M$^+$19] J. D. Mitchell et al. *Semigroups – GAP package, Version 3.1.1*, Feb 2019.

[Mak66] G. Makanin. The identity problem for finitely presented semigroups. *Soviet Math. Dokl*, 7:1478–1480, 1966.

[Mal52] A. I. Mal′cev. Symmetric groupoids. *Mat. Sbornik N.S.*, 31(73):136–151, 1952.

[Mal53] A. I. Malcev. Multiplicative congruences of matrices. In *Doklady Akad. Nauk SSSR*, volume 90, pages 333–335, 1953.

[Mar94] P. Martin. Temperley–Lieb algebras for non-planar statistical mechanics – the partition algebra construction. *Journal of Knot Theory and Its Ramifications*, 03(01):51–82, 1994.

[McA99] D. B. McAlister. *Semigroup for Windows – C++ program, Version 1.0*, Mar 1999.

[Mel95] J. D. P. Meldrum. *Wreath products of groups and semigroups*, volume 74. CRC Press, 1995.

[Mil92] C. F. Miller. Decision problems for groups – survey and reflections. In *Algorithms and classification in combinatorial group theory*, pages 1–59. Springer, 1992.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[MS56] T. S. Motzkin and J. L. Selfridge. Semigroups of order five. *Bulletin of the American Mathematical Society*, 62:14, 1956.

[MT$^+$18] J. D. Mitchell, M. Torpey, et al. *libsemigroups – C++ library, Version 0.6.4*, Sep 2018.

[Neu67] B. H. Neumann. Some remarks on semigroup presentations. *Canad. J. Math*, 19:1018–1026, 1967.

[NS78] T. E. Nordahl and H. E. Scheiblich. Regular $\star$ semigroups. *Semigroup Forum*, 16(1):369–377, Dec 1978.

[OEIS] N. J. A. Sloane et al. The on-line encyclopedia of integer sequences. Published electronically at `http://oeis.org/`.

[Pet84] M. Petrich. *Inverse semigroups.* Wiley-Interscience, 1984.

[Pin09] J.-E. Pin. *Semigroupe – C program, Version 2.01*, Apr 2009.

[PP86] F. Pastijn and M. Petrich. Congruences on regular semigroups. *Transactions of the American Mathematical Society*, 295(2):607–633, 1986.

[RH09] C. Ramsay and G. Havas. *ACE (Advanced Coset Enumerator) – C program, Version 4.1*, June 2009.

[Rot65] J. J. Rotman. *The Theory of Groups: An Introduction.* Allyn and Bacon, Inc., 1965.

[Ruš95] N. Ruškuc. *Semigroup presentations.* PhD thesis, University of St Andrews, 1995.

[Shu88] E. G. Shutov. Homomorphisms of the semigroup of all partial transformations. *Nineteen Papers on Algebraic Semigroups*, 139:183, 1988.

[Sim94] C. C. Sims. *Computation with finitely presented groups*, volume 48. Cambridge University Press, 1994.

[Sus28] A. Suschkewitsch. Über die endlichen gruppen ohne das gesetz der eindeutigen umkehrbarkeit. *Mathematische Annalen*, 99(1):30–50, 1928.

[Tam53] T. Tamura. Some remarks on semi-groups and all types of semi-groups order 2,3. *Journal of Gakugei, Tokushima University. Natural science (Mathematics)*, 3:1–11, 1953.

[TC36] J. A. Todd and H. S. M. Coxeter. A practical method for enumerating cosets of a finite abstract group. *Proceedings of the Edinburgh Mathematical Society*, 5(1):26–34, 1936.

[Tor14a] M. Torpey. Computing with congruences on finite 0-simple semigroups. MT5991 Report, University of St Andrews, 2014.

[Tor14b] M. Torpey. Computing with semigroup congruences. Master's thesis, University of St Andrews, 2014.

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[TvL84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[vLW77] J. van Leeuwen and R. Weide. Alternative path compression techniques. *RUU-CS*, 77(3), 1977.

[Wal92] T. Walker. *Semigroup Enumeration – Computer Implementation and Applications.* PhD thesis, University of St Andrews, 1992.

[War90] S. Warner. *Modern algebra.* Courier Corporation, 1990.

List of notation

!!	Double factorial (product of every other integer from n down to 1)	159
*	Free monoid	28
*	Star operation on bipartitions	40
$^+$	Free semigroup	28
$\overline{}$	Principal factor	164
$^{-1}$	Inverse of an element	17
$^{-1}$	Relation with pairs swapped	24
$\hookrightarrow$	Monomorphism	20
$\twoheadrightarrow$	Epimorphism	20
$\hookrightarrow\!\!\!\!\!\twoheadrightarrow$	Isomorphism	20
$\to_{\mathbf{R}}$	Rewriting system operation	78
$\xrightarrow{*}_{\mathbf{R}}$	Reflexive transitive closure of rewriting system operation	78
$\overset{*}{\leftrightarrow}_{\mathbf{R}}$	Equivalence of rewriting system operation	78
$\cdot$	Binary operation in a semigroup	17
$/$	Quotient semigroup	22
$\subset$	Strict subset containment (excluding equality)	16
$\subseteq$	Subset containment (including equality)	16
$\langle\ \rangle$	Semigroup by generators	19
$\langle\!\langle\ \rangle\!\rangle$	Normal closure	20
$^\infty$	Transitive closure of a relation	24
$\frown$	Bipartition of rank 0 with this kernel and cokernel	145
$\cap$	Intersection of two sets	23
$\vee$	Join of two equivalences	23
^{1}	Semigroup with a possible identity appended	17
$^{(1)}$	Semigroup with an identity definitely appended	17
$[\ ]$	Equivalence class containing an element	16
Δ	Diagonal relation	16

Γ	Linked triple function	103
∇	Universal relation	16
ϕ	Homomorphism	20
π	Natural homomorphism	22
Ψ	Bijection to kernel–trace pairs	105
ρ_I	Rees congruence	31
σ	Liftable congruence	142
$\zeta_{I,\sigma}$	Lifted congruence	143
$\zeta_{I,N,\sigma}$	Congruence defined by an IN-pair and liftable congruence	157
$\mathcal{B}_n$	Brauer monoid	159
B_n	Bell number (number of partitions on an n-set)	42
^{c}	Least compatible relation containing a relation	25
codom	Codomain of a bipartition	39
coker	Cokernel of a bipartition	40
$\mathscr{D}$	Green's $\mathscr{D}$-relation	31
D_k^n	$\mathscr{D}$-class of rank k in $\mathcal{T}_n$ and other monoids	165
dom	Domain of a (partial) transformation	37
dom	Domain of a bipartition	39
^{e}	Least equivalence containing a relation	25
e	Identity of a monoid	17
G	Group	17
$\mathscr{H}$	Green's $\mathscr{H}$-relation	31
id_n	Identity map	35
I_J	Ideal generated by a $\mathscr{J}$-class	164
I_k	Ideal of the Motzkin monoid $\mathcal{M}_n$	141
im	Image of a (partial) transformation	37
im	Image of a homomorphism	21
$\mathcal{I}_n$	Symmetric inverse monoid	37
$\mathscr{J}$	Green's $\mathscr{J}$-relation	31
$\mathcal{J}_n$	Jones monoid	161
K_4	Klein 4-group	158
ker	Kernel of a (partial) transformation	37
ker	Kernel of a bipartition	39
ker	Kernel of a homomorphism	21

^{l}	Least left-compatible relation containing a relation	25
$\mathscr{L}$	Green's $\mathscr{L}$-relation	31
$\mathscr{L}^{I_0}$	Green's $\mathscr{L}$-relation retricted to the ideal I_0	144
$\mathcal{LZ}_n$	Left zero semigroup	19
M	Monoid	17
$\mathcal{M}^0[G; I, \Lambda; P]$	Rees 0-matrix semigroup	102
$\mathcal{M}[G; I, \Lambda; P]$	Rees matrix semigroup	104
$\mathcal{M}_n$	Motzkin monoid	140
$\mathbb{N}$	Natural numbers $\{1, 2, 3, \ldots\}$	16
N	Normal subgroup	18
$(N, \mathcal{S}, \mathcal{T})$	Linked triple	103
$\mathbf{n}$	$\{1, 2, \ldots, n\}$ for some $n \in \mathbb{N}$	16
$\mathbf{n}'$	$\{1', 2', \ldots, n'\}$ for some $n \in \mathbb{N}$	38
$\mathcal{O}_n$	Monoid of order-preserving transformations	37
$\mathcal{P}_n$	Bipartition monoid	40
$\mathcal{PB}_n$	Partial Brauer monoid	161
$\mathcal{P}B_n$	Full PBR monoid	85
$\mathcal{POI}_n$	Monoid of order-preserving partial permutations	37
$\mathcal{PO}_n$	Monoid of order-preserving partial transformations	37
$\mathscr{P}\mathcal{P}_n$	Planar bipartition monoid	159
$\mathcal{PT}_n$	Partial transformation monoid	36
^{r}	Least right-compatible relation containing a relation	25
$\mathscr{R}$	Green's $\mathscr{R}$-relation	31
$\mathscr{R}^{I_0}$	Green's $\mathscr{R}$-relation retricted to the ideal I_0	144
rank	Rank of a (partial) transformation	37
$\mathbf{R}$	Relation on a semigroup	21
$\mathbf{R}$	Rewriting system	78
$R_{I,N}$	Congruence defined by an IN-pair	157
R_k	Rees congruence of the Motzkin monoid $\mathcal{M}_n$	144
$\mathcal{RZ}_n$	Right zero semigroup	19
$\mathcal{S}$	Relation on column set I	103
S	Semigroup	17
$\mathcal{S}_n$	Symmetric group	36
$\mathcal{T}$	Relation on row set Λ	103
$\mathcal{T}_n$	Full transformation monoid	35
$\mathcal{Z}_n$	Zero semigroup	19

Index