Journal of Algebra 585 (2021) 723–758

Contents lists available at ScienceDirect

Journal of Algebra

www.elsevier.com/locate/jalgebra

Permutation group algorithms based on directed graphs

Christopher Jefferson [a], Markus Pfeiffer [a], Wilf A. Wilson [a], Rebecca Waldecker [b,*]

[a] *University of St Andrews, School of Computer Science, North Haugh, St Andrews, KY16 9SX, Scotland, United Kingdom*
[b] *Martin-Luther-Universität Halle-Wittenberg, Institut für Mathematik, 06099 Halle, Germany*

A R T I C L E   I N F O

Article history:
Received 17 December 2020
Available online 29 June 2021
Communicated by Derek Holt

Keywords:
Permutation groups
Search algorithms
Digraphs
Computational group theory
Graphs
Backtrack search

A B S T R A C T

We introduce a new framework for solving an important class of computational problems involving finite permutation groups, which includes calculating set stabilisers, intersections of subgroups, and isomorphisms of combinatorial structures. Our techniques are inspired by and generalise 'partition backtrack', which is the current state-of-the-art algorithm introduced by Jeffrey Leon in 1991. But, instead of ordered partitions, we use labelled directed graphs to organise our backtrack search algorithms, which allows for a richer representation of many problems while often resulting in smaller search spaces. In this article we present the theory underpinning our framework, we describe our algorithms, and we show the results of some experiments. An implementation of our algorithms is available as free software in the GraphBacktracking package for GAP.

© 2021 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

* Corresponding author.
E-mail addresses: caj21@st-andrews.ac.uk (C. Jefferson), markus.pfeiffer@st-andrews.ac.uk (M. Pfeiffer), waw7@st-andrews.ac.uk (W.A. Wilson), rebecca.waldecker@mathematik.uni-halle.de (R. Waldecker).
URLs: https://caj.host.cs.st-andrews.ac.uk (C. Jefferson), https://www.morphism.de/~markusp (M. Pfeiffer), https://wilf.me (W.A. Wilson), https://www2.mathematik.uni-halle.de/waldecker/index-english.html (R. Waldecker).

https://doi.org/10.1016/j.jalgebra.2021.06.015
0021-8693/© 2021 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Many of the most important problems in computational permutation group theory can be phrased as search problems, where we typically search for the intersection of subsets of a symmetric group. Standard problems that match this description are the computation of point stabilisers or set stabilisers, of transporter sets, or of normalisers or centralisers of subgroups. Searching for automorphisms and isomorphisms of a wide range of combinatorial structures can be done in this way as well as deciding whether or not combinatorial objects are in the same orbit under some group action, as is the case for element and subgroup conjugacy.

For many of these problems, the best known way to solve them is based on Leon's *partition backtrack algorithm* (see [12]), which often performs excellently, but has exponential worst-case complexity. Leon's algorithm conducts a backtrack search through the elements of the symmetric group, which it organises around a collection of ordered partitions (see [9] for more details). By encoding information about the given problem into those partitions, it is possible to cleverly prune (i.e., omit superfluous parts of) the search space. There have already been some extensions and improvements, inspired by graph-based ideas of McKay (see for example [14] and [8]). This leads us to believe that more powerful pruning, and ultimately better performance, could be obtained by using graphs directly, at the expense of the increased computation required at each node of the remaining search. In the present paper, we therefore place labelled directed graphs at the heart of backtrack search algorithms.

The basic idea is parallel to that described above for Leon's algorithm: When we search for an intersection of subsets of the symmetric group, we find suitable labelled digraph stacks such that the intersection can be viewed as the set of isomorphisms (induced from the symmetric group) from the first labelled digraph stack to the second (see Section 3). We encode information about the subsets into the labelled digraph stacks with refiners (see Section 4), and we just remark here that this generalises partition backtrack, because partition backtrack can be viewed as using vertex-labelled digraphs without arcs, where the vertex labels are in one-to-one correspondence with the cells of the ordered partitions. Approximators capture the fact that we typically overestimate the set of isomorphisms rather than calculate it exactly (see Section 5). The last ingredient comes into play when our approximation indicates that the refiners have encoded as much information as possible in a given moment and cannot restrict the search space further. Then we divide the search into smaller areas by defining new labelled digraph stacks, which is known as splitting (see Section 6). We discuss algorithms based on the method just described and prove their correctness in Sections 7 and 8, and in Section 9 we give details of various experiments that compare our algorithms with the current state-of-the-art techniques. We conclude, in Section 10, with brief comments on the results of this paper and the directions that they suggest for further investigation.

Finally, we would like to mention that we expect the reader to be familiar with basic concepts of permutation group theory and graph theory and that we only briefly explain

our notation before moving to the main content. There is an extended version of this article [9], where we give proofs that are omitted here, along with much more detail and background information. We also include additional examples there and new material that is currently in preparation for a separate publication.

Acknowledgments

The authors would like to thank the DFG (**Grant no. WA 3089/6-1**) and the VolkswagenStiftung (**Grant no. 93764**) for financially supporting this work and projects leading up to it. The first and third authors are supported by the Royal Society (**Grant codes RGF\EA\181005 and URF\R\180015**). Special thanks go to Paula Hähndel and Ruth Hoffmann for frequent discussions on topics related to this work, and for suggestions on how to improve this paper. Finally, the authors thank the referees for reading our drafts carefully and for making many helpful suggestions.

2. Preliminaries

Throughout this paper, Ω denotes some finite totally-ordered set on which we define all of our groups, digraphs, and related objects. For example, every group in this paper is a finite permutation group on Ω, i.e., a subgroup of $\mathrm{Sym}(\Omega)$, the symmetric group on Ω. We follow the standard group-theoretic notation and terminology from the literature, such as that used in [2], and write $\cdot$ for the composition of maps in $\mathrm{Sym}(\Omega)$, or we omit a symbol for this binary operation altogether. We write $\mathbb{N}$ for the set $\{1, 2, 3, \ldots\}$ of all natural numbers, and $\mathbb{N}_0 \coloneqq \mathbb{N} \cup \{0\}$. If $n \in \mathbb{N}$, then $\mathcal{S}_n \coloneqq \mathrm{Sym}(\{1, \ldots, n\})$. For many types of objects that we define on Ω, for example lists, sets, or graphs, we give a way of applying elements of $\mathrm{Sym}(\Omega)$ to them (denoted by exponentiation) in a structure-preserving way.

Let Γ and Δ be digraphs (which is short for directed graphs) with vertex set Ω. Then we say that a permutation $g \in \mathrm{Sym}(\Omega)$ *induces an isomorphism from* Γ *to* Δ if and only if it defines a structure-preserving map from Γ to Δ, in which case we write $\Gamma^g = \Delta$.

We use the notation $\mathrm{Iso}(\Gamma, \Delta)$ for the set of isomorphisms from Γ to Δ that are induced by elements of $\mathrm{Sym}(\Omega)$. If $\mathrm{Iso}(\Gamma, \Delta)$ is non-empty, then we call Γ and Δ *isomorphic*. Similarly, we write $\mathrm{Aut}(\Gamma) \coloneqq \mathrm{Iso}(\Gamma, \Gamma)$ for the subgroup of $\mathrm{Sym}(\Omega)$ consisting of all elements that induce *automorphisms* of Γ.

2.1. Labelled digraphs

Our techniques for searching in $\mathrm{Sym}(\Omega)$ are built around digraphs in which each vertex and arc (i.e. directed edge) is given a *label* from a set of labels $\mathfrak{L}$. We define a *vertex- and arc-labelled digraph*, or *labelled digraph* for short, to be a triple $(\Omega, A, \textsc{Label})$, where (Ω, A) is a digraph and $\textsc{Label}$ is a function from $\Omega \cup A$ to $\mathfrak{L}$. More precisely, for any $\delta \in \Omega$ and $(\alpha, \beta) \in A$, the label of the vertex δ is $\textsc{Label}(\delta) \in \mathfrak{L}$, and the label of the

arc (α, β) is $\textsc{Label}(\alpha, \beta) \in \mathfrak{L}$. We call such a function a *labelling function*. We point out that our notion of digraphs is very general and that loops are allowed. (For more details and some examples see [9].)

We fix $\mathfrak{L}$ as some non-empty set that contains every label that we require and serves as the codomain of every labelling function. For the equitable vertex labelling algorithm discussed in Section 5.2, we require some arbitrary but fixed total ordering on $\mathfrak{L}$.

The symmetric group on Ω acts on the sets of graphs and digraphs with vertex set Ω, respectively, and on their labelled variants, in a natural way. We give more details about this for labelled digraphs; the forthcoming notions are defined analogously for the other kinds of graphs and digraphs that we have mentioned. Let $\textsc{LabelledDigraphs}(\Omega, \mathfrak{L})$ denote the class of labelled digraphs on Ω with labels in $\mathfrak{L}$, and let $\Gamma = (\Omega, A, \textsc{Label}) \in \textsc{LabelledDigraphs}(\Omega, \mathfrak{L})$ and $g \in \mathrm{Sym}(\Omega)$. Then we define $\Gamma^g = (\Omega, A^g, \textsc{Label}^g) \in \textsc{LabelledDigraphs}(\Omega, \mathfrak{L})$, where:

(i) $A^g := \{(\alpha^g, \beta^g) : (\alpha, \beta) \in A\}$,
(ii) $\textsc{Label}^g(\delta) := \textsc{Label}(\delta^{g^{-1}})$ for all $\delta \in \Omega$, and
(iii) $\textsc{Label}^g(\alpha, \beta) := \textsc{Label}(\alpha^{g^{-1}}, \beta^{g^{-1}})$ for all $(\alpha, \beta) \in A^g$.

In other words, the arcs are mapped according to g, and the label of a vertex or arc in Γ^g is the label of its preimage in Γ. This gives rise to a group action of $\mathrm{Sym}(\Omega)$ on $\textsc{LabelledDigraphs}(\Omega, \mathfrak{L})$.

3. Stacks of labelled digraphs

In this section we introduce labelled digraph stacks, with the rough idea in mind that we use them to approximate the set of permutations we search for. More precisely, we attempt to choose suitable labelled digraph stacks in such a way that the set of isomorphisms from one to the other approximates the set we search for as closely as possible.

A *labelled digraph stack* on Ω is a finite (possibly empty) list of labelled digraphs on Ω. We denote the collection of all labelled digraph stacks on Ω by $\textsc{DigraphStacks}(\Omega)$. The *length* of a labelled digraph stack S, written as $|S|$, is the number of entries that it contains. A labelled digraph stack of length 0 is called *empty*, and is denoted by $\textsc{EmptyStack}(\Omega)$. We use notation typical for lists, whereby if $i \in \{1, \ldots, |S|\}$, then $S[i]$ denotes the ith labelled digraph in the stack S.

We allow any labelled digraph stack on Ω to be appended onto the end of another. If $S, T \in \textsc{DigraphStacks}(\Omega)$ have lengths k and l, respectively, then we define $S \| T$ to be the labelled digraph stack $[S[1], \ldots, S[k], T[1], \ldots, T[l]]$ of length $k + l$.

We define an action of $\mathrm{Sym}(\Omega)$ on $\textsc{DigraphStacks}(\Omega)$ via the action of $\mathrm{Sym}(\Omega)$ on the set of all labelled digraphs on Ω. More specifically, for all $S \in \textsc{DigraphStacks}(\Omega)$ and $g \in \mathrm{Sym}(\Omega)$, we define S^g to be the labelled digraph stack of length $|S|$ with $S^g[i] = S[i]^g$ for all $i \in \{1, \ldots, |S|\}$. An *isomorphism* from S to another labelled digraph

stack T (induced by $\mathrm{Sym}(\Omega)$) is therefore a permutation $g \in \mathrm{Sym}(\Omega)$ such that $S^g = T$. In particular, only digraph stacks of equal lengths can be isomorphic. We note that every permutation in $\mathrm{Sym}(\Omega)$ induces an automorphism of $\textsc{EmptyStack}(\Omega)$. As we do with digraphs, we use the notation $\mathrm{Iso}(S,T)$ for the set of isomorphisms from the stack S to the stack T induced by elements of $\mathrm{Sym}(\Omega)$, and $\mathrm{Aut}(S)$ for the group of automorphisms of S induced by elements of $\mathrm{Sym}(\Omega)$.

Remark 3.1. Let $S, T, U, V \in \textsc{DigraphStacks}(\Omega)$. It follows from the definitions that

$$\mathrm{Iso}(S,T) = \begin{cases} \varnothing & \text{if } |S| \neq |T|, \\ \bigcap_{i=1}^{|S|} \mathrm{Iso}(S[i], T[i]) & \text{if } |S| = |T|, \end{cases} \quad \text{and that } \mathrm{Aut}(S) = \bigcap_{i=1}^{|S|} \mathrm{Aut}(S[i])\,.$$

In addition $\mathrm{Aut}(S \,\|\, U) \leq \mathrm{Aut}(S)$, and if $|S| = |T|$, then $\mathrm{Iso}(S \,\|\, U, T \,\|\, V) \subseteq \mathrm{Iso}(S,T)$. Roughly speaking, the automorphism group of a labelled digraph stack, and the set of isomorphisms from one labelled digraph stack to another one of equal length, become potentially smaller as new entries are added to the stacks.

We illustrate some of the foregoing concepts in Example 3.3. Since we use an orbital graph, we briefly recall (see for example [2, Section 3.2]):

Definition 3.2 *(Orbital graph).* Let $G \leq \mathrm{Sym}(\Omega)$, and let $\alpha, \beta \in \Omega$ be such that $\alpha \neq \beta$. Then the *orbital graph of G with base-pair* (α, β) is the digraph $(\Omega, \{(\alpha^g, \beta^g) : g \in G\})$.

Example 3.3. Let $\Omega = \{1, \ldots, 6\}$. Here we define a labelled digraph stack S on Ω that has length 3, by describing each of its members.

We define the first entry of S via the orbital graph of $K := \langle (1\,2)(3\,4)(5\,6), (2\,4\,6) \rangle$ with base-pair $(1, 3)$. The automorphism group of this orbital graph (as always, induced by $\mathrm{Sym}(\Omega)$) is K itself; in other words, this orbital graph perfectly represents K via its automorphism group. In order to define $S[1]$, we convert this orbital graph into a labelled digraph by assigning the label *white* to each vertex and assigning the label *solid* to each arc. This does not change the automorphism group of the digraph.

We define the second entry of S to be the labelled digraph on Ω without arcs, whose vertices 1 and 2 are labelled *black*, and whose remaining vertices are labelled *white*. The automorphism group of this labelled digraph is the setwise stabiliser of $\{1, 2\}$ in $\mathrm{Sym}(\Omega)$.

We define the third entry of S to be the labelled digraph $S[3]$ shown in Fig. 3.4, with arcs and labels chosen from the set $\{$*black*, *white*, *solid*, *dashed*$\}$ as depicted there; its automorphism group is $\langle (1\,2), (3\,4)(5\,6) \rangle$.

Given the automorphism groups of the individual entries of S, as described above, it follows that the automorphism group of S consists of precisely those elements of K that stabilise the set $\{1, 2\}$, and that are automorphisms of the labelled digraph $S[3]$. Hence this group is $\langle (1\,2)(3\,4)(5\,6) \rangle$. Since $(1\,2)$ is an automorphism of $S[2]$ and $S[3]$, but not of $S[1]$, it follows that $S^{(1\,2)} = [S[1]^{(1\,2)}, S[2], S[3]] \neq S$. We also note that $\mathrm{Iso}\big(S, S^{(1\,2)}\big)$ is the right coset $\mathrm{Aut}(S) \cdot (1\,2) = \{(1\,2), (3\,4)(5\,6)\}$ of $\mathrm{Aut}(S)$ in $\mathrm{Sym}(\Omega)$.

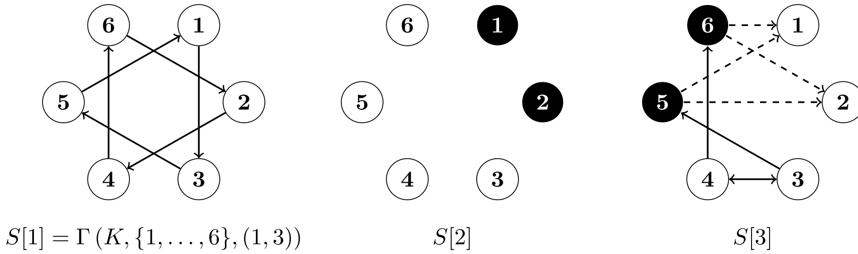

Fig. 3.4. Diagrams of the labelled digraphs in the labelled digraph stack S from Example 3.3. The vertices and arcs of these labelled digraphs are styled according to their labels, which are chosen from the set {*black*, *white*, *solid*, *dashed*}.

3.1. The squashed labelled digraph of a stack

For our exposition in Section 5, it is convenient to have a labelled digraph whose automorphism group is equal to that of a given labelled digraph stack. This is analogous to the final entry of an ordered partition stack [12, Section 4]. This special labelled digraph is a new object defined from the stack, but it is not part of the stack itself.

For this we fix a symbol # that is never to be used as the label of a vertex or an arc in any labelled digraph.

Definition 3.5. Let S be a labelled digraph stack on Ω, with $S[i] := (\Omega, A_i, \textsc{Label}_i)$ being some labelled digraph on Ω for each $i \in \{1, \ldots, |S|\}$. Then the *squashed labelled digraph* of S, denoted by $\textsc{Squash}(S)$, is the labelled digraph $(\Omega, A, \textsc{Label})$, where

- $A = \bigcup_{i=1}^{|S|} A_i$,
- $\textsc{Label}(\delta) = [\textsc{Label}_1(\delta), \ldots, \textsc{Label}_{|S|}(\delta)]$ for all $\delta \in \Omega$, and
- $\textsc{Label}(\alpha, \beta)$ is the list of length $|S|$ for all $(\alpha, \beta) \in \bigcup_{i=1}^{|S|} A_i$, where

$$\textsc{Label}(\alpha, \beta)[i] = \begin{cases} \textsc{Label}_i(\alpha, \beta) & \text{if } (\alpha, \beta) \in A_i, \\ \# & \text{if } (\alpha, \beta) \notin A_i, \end{cases} \quad \text{for all } i \in \{1, \ldots, |S|\}.$$

Note that the labelling function of the squashed labelled digraph of a stack can be used to reconstruct all information about the stack from which it was created. We also point out that $\textsc{Squash}(S)^g = \textsc{Squash}(S^g)$ for all $S \in \textsc{DigraphStacks}(\Omega)$ and $g \in \mathrm{Sym}(\Omega)$. Therefore the following lemma holds.

Lemma 3.6. *Let* $S, T \in \textsc{DigraphStacks}(\Omega)$*. Then*

$$\mathrm{Iso}(S, T) = \mathrm{Iso}(\textsc{Squash}(S), \textsc{Squash}(T)).$$

Example 3.7. Let S be the labelled digraph stack from Example 3.3. Since $|S| = 3$, the labels in $\textsc{Squash}(S)$ are lists of length 3. The vertex labels in $\textsc{Squash}(S)$ are:

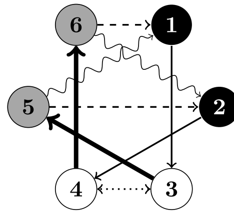

Fig. 3.8. A depiction of the squashed labelled digraph $\textsc{Squash}(S)$ from Example 3.7, which is constructed from the labelled digraph stack S from Example 3.3.

- $\textsc{Label}(1) = \textsc{Label}(2) = [\textit{white}, \textit{black}, \textit{white}]$, shown as *black* in Fig. 3.8,
- $\textsc{Label}(3) = \textsc{Label}(4) = [\textit{white}, \textit{white}, \textit{white}]$, shown as *white* in Fig. 3.8, and
- $\textsc{Label}(5) = \textsc{Label}(6) = [\textit{white}, \textit{white}, \textit{black}]$, shown as *grey* in Fig. 3.8.

There are ten arcs in $\textsc{Squash}(S)$, which in total have five different labels:

- $\textsc{Label}(1,3) = \textsc{Label}(2,4) = [\textit{solid}, \#, \#]$, shown as *thin* in Fig. 3.8,
- $\textsc{Label}(3,4) = \textsc{Label}(4,3) = [\#, \#, \textit{solid}]$, shown as *dotted* in Fig. 3.8,
- $\textsc{Label}(5,2) = \textsc{Label}(6,1) = [\#, \#, \textit{dashed}]$, shown as *dashed* in Fig. 3.8,
- $\textsc{Label}(3,5) = \textsc{Label}(4,6) = [\textit{solid}, \#, \textit{solid}]$, shown as *thick* in Fig. 3.8, and
- $\textsc{Label}(5,1) = \textsc{Label}(6,2) = [\textit{solid}, \#, \textit{dashed}]$, shown as *wavy* in Fig. 3.8.

Since automorphisms of labelled digraphs preserve the sets of vertices with any particular label, it is clear that $\mathrm{Aut}(\textsc{Squash}(S)) \leq \langle (1\,2), (3\,4), (5\,6) \rangle$. This containment is proper, since $\mathrm{Aut}(\textsc{Squash}(S)) = \mathrm{Aut}(S)$ by Lemma 3.6, and $\mathrm{Aut}(S) = \langle (1\,2)(3\,4)(5\,6) \rangle$, as discussed in Example 3.3. Indeed, inspection of the arc labels in $\textsc{Squash}(S)$ shows that any automorphism that interchanges the pair of points in any of $\{1,2\}$, $\{3,4\}$, or $\{5,6\}$ also interchanges the other pairs.

4. Adding information to stacks with refiners

In this section we introduce and discuss refiners for labelled digraph stacks. We use refiners to encode information about a search problem into the stacks around which the search is organised, in order to prune the search space.

Definition 4.1. A *refiner* for a set of permutations $U \subseteq \mathrm{Sym}(\Omega)$ is a pair (f_L, f_R) of functions from $\textsc{DigraphStacks}(\Omega)$ to itself such that for all isomorphic $S, T \in \textsc{DigraphStacks}(\Omega)$:

$$U \cap \mathrm{Iso}(S,T) \subseteq U \cap \mathrm{Iso}(f_L(S), f_R(T))\,.$$

Leon introduces the concept of refiners in [12]. Although the word "refiner" might seem counter-intuitive from the definition, Remark 3.1 makes it clear that the stacks

$S \| f_L(S)$ and $T \| f_R(T)$ do indeed give rise to a closer (or "finer") approximation of the set we search for.

While a refiner depends on a subset of $\mathrm{Sym}(\Omega)$, we do not include this in our notation in order to make it less complicated. As a trivial example, every pair of functions from DigraphStacks(Ω) to itself is a refiner for the empty set, and it is in fact relevant for practical applications to be able to search for the empty set.

In the following lemma, we formulate additional equivalent definitions of refiners.

Lemma 4.2. *Let (f_L, f_R) be a pair of functions from* DigraphStacks(Ω) *to itself and let $U \subseteq \mathrm{Sym}(\Omega)$. Then the following are equivalent:*

(i) *(f_L, f_R) is a refiner for U.*
(ii) *For all isomorphic $S, T \in$* DigraphStacks(Ω)*:*

$$U \cap \mathrm{Iso}(S,T) = U \cap \mathrm{Iso}(S \parallel f_L(S), T \parallel f_R(T))\,.$$

(iii) *For $S, T \in$* DigraphStacks(Ω) *and $g \in U$:*

$$\textit{if } S^g = T, \textit{ then } f_L(S)^g = f_R(T).$$

Proof. (i) $\Rightarrow$ (ii). Let $S, T \in$ DigraphStacks(Ω) be isomorphic. Then $U \cap \mathrm{Iso}(S,T) \subseteq U \cap \mathrm{Iso}(f_L(S), f_R(T))$ by assumption, and since S and T have equal lengths, then

$$\mathrm{Iso}(S,T) \cap \mathrm{Iso}(f_L(S), f_R(T)) = \mathrm{Iso}(S \parallel f_L(S), T \parallel f_R(T))$$

by Remark 3.1. Hence

$$\begin{aligned} U \cap \mathrm{Iso}(S,T) &= U \cap \mathrm{Iso}(S,T) \cap \big(U \cap \mathrm{Iso}(f_L(S), f_R(T))\big) \\ &= U \cap \big(\mathrm{Iso}(S,T) \cap \mathrm{Iso}(f_L(S), f_R(T))\big) \\ &= U \cap \mathrm{Iso}(S \parallel f_L(S), T \parallel f_R(T))\,. \end{aligned}$$

(ii) $\Rightarrow$ (iii). Let $S, T \in$ DigraphStacks(Ω) and let $g \in U$. If $S^g = T$, then $g \in \mathrm{Iso}(S,T)$ by definition, and so $g \in \mathrm{Iso}(S \parallel f_L(S), T \parallel f_R(T))$ by assumption. Since S and T have equal lengths, and $S \parallel f_L(S)$ and $T \parallel f_R(T)$ have equal lengths, it follows that so too do $f_L(S)$ and $f_R(T)$. Then $f_L(S)^g = f_R(T)$, since for each $i \in \{1, \ldots, |f_L(S)|\}$,

$$f_L(S)[i]^g = (S \parallel f_L(S))[|S| + i]^g = (T \parallel f_R(T))[|T| + i] = f_R(T)[i].$$

(iii) $\Rightarrow$ (i). This implication is immediate. □

Perhaps Lemma 4.2(ii) most clearly indicates the relevance of refiners to search.

Suppose we wish to search for the intersection $U_1 \cap \cdots \cap U_n$ of some subsets of $\mathrm{Sym}(\Omega)$. Let $i \in \{1, \dots, n\}$, let (f_L, f_R) be a refiner for U_i, and let S and T be isomorphic labelled digraph stacks on Ω, such that $\mathrm{Iso}(S, T)$ overestimates (i.e., contains) $U_1 \cap \cdots \cap U_n$.

We may use the refiner (f_L, f_R) to *refine* the pair of stacks (S, T): we apply the functions f_L and f_R, respectively, to the stacks S and T and obtain an extended pair of stacks $(S \parallel f_L(S), T \parallel f_R(T))$. We call this process *refinement.* Note that a refiner for U_i need not consider the other sets in the intersection.

By Lemma 4.2(ii), the set of induced isomorphisms $\mathrm{Iso}(S \parallel f_L(S), T \parallel f_R(T))$ contains the elements of U_i that belonged to $\mathrm{Iso}(S, T)$. Since U_i contains $U_1 \cap \cdots \cap U_n$, it follows that $\mathrm{Iso}(S \parallel f_L(S), T \| f_R(T))$ is a (possibly smaller) new overestimate for $U_1 \cap \cdots \cap U_n$ which is contained in the previous overestimate by Remark 3.1.

The following straightforward example illustrates how the condition in Lemma 4.2(iii) is useful for showing that a pair of functions is a refiner for some set.

Example 4.3 *(Labelled digraph automorphism and isomorphism).* Let Γ and Δ be labelled digraphs on Ω, and define constant functions f_Γ and f_Δ on DigraphStacks(Ω), whose images are the length-one digraph stacks $[\Gamma]$ and $[\Delta]$, respectively.

Since the permutations of Ω that induce isomorphisms from $[\Gamma]$ to $[\Delta]$ are exactly those that induce isomorphisms from Γ to Δ, it follows by Lemma 4.2(iii) that (f_Γ, f_Δ) is a refiner for $\mathrm{Iso}(\Gamma, \Delta)$. In particular, (f_Γ, f_Γ) is a refiner for $\mathrm{Aut}(\Gamma)$.

Example 4.3 illustrates the principle that the functions of the refiner are equal if it is a refiner for a subgroup. The next lemma states a slightly stronger observation. We omit the proofs of the next few results and refer to [9].

Lemma 4.4 *(cf. [11, Prop 2], [12, Lemma 6]). Let (f_L, f_R) be a refiner for a subset $U \subseteq$* $\mathrm{Sym}(\Omega)$ *that contains the identity map, id_Ω. Then $f_L = f_R$.*

Lemma 4.2(iii) implies:

Lemma 4.5. *Let f be a function from* DigraphStacks(Ω) *to itself, and let U be a subset of* $\mathrm{Sym}(\Omega)$ *containing id_Ω. Then (f, f) is a refiner for U if and only if $f(S^g) = f(S)^g$ for all $g \in U$ and $S \in$* DigraphStacks(Ω).

Next, we see that any refiner for a non-empty set can be derived from a function that satisfies the condition in Lemma 4.5.

Lemma 4.6. *Let U be a non-empty subset of* $\mathrm{Sym}(\Omega)$*, fix $x \in U$, and let f_L and f_R be functions from* DigraphStacks(Ω) *to itself. Then the following are equivalent:*

- *(f_L, f_R) is a refiner for U.*
- *(f_L, f_L) is a refiner for Ux^{-1} and $f_R(S) = f_L(S^{x^{-1}})^x$ for all $S \in$* DigraphStacks(Ω).

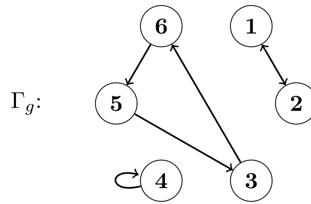

Fig. 4.9. The labelled digraph Γ_g for $g = (1\,2)(3\,6\,5)$, from Example 4.8.

In particular, if U is a right coset of a subgroup $G \leq \mathrm{Sym}(\Omega)$, then (f_L, f_R) is a refiner for the coset $U = Gx$ if and only if (f_L, f_L) is a refiner for the group G, and $f_R(S) = f_L(S^{x^{-1}})^x$ for all $S \in$ DigraphStacks(Ω).

For some pairs of functions, such as those in the upcoming Example 4.12, one may use the following results to show that a pair of functions gives a refiner.

Lemma 4.7. *Let $U \subseteq \mathrm{Sym}(\Omega)$, and let f_L, f_R be functions from* DigraphStacks(Ω) *to itself such that $U \subseteq \mathrm{Iso}(f_L(S), f_R(T))$ for all isomorphic $S, T \in$* DigraphStacks(Ω)*. Then (f_L, f_R) is a refiner for U.*

4.1. Examples of refiners

Here we give several further examples of refiners for subgroups and their cosets, for typical group theoretic problems. We use refiners from Example 4.10 in our experiments of Section 9.1. The refiners given in Examples 4.8 and 4.10 have in common that they perfectly capture all the information about the set that we search for. This is also the case for the refiners given in Example 4.12 for sets of pairwise disjoint subsets of Ω, and for sets of subsets of Ω with pairwise distinct sizes.

As we saw in Lemmas 4.5 and 4.6, the crucial step when creating a refiner for a subgroup $G \leq \mathrm{Sym}(\Omega)$ or one of its cosets is to define a function f from DigraphStacks(Ω) to itself such that $f(S^g) = f(S)^g$ for all $S \in$ DigraphStacks(Ω) and $g \in G$.

Example 4.8 *(Permutation centraliser and conjugacy).* For every $g \in \mathrm{Sym}(\Omega)$, let Γ_g be the labelled digraph on Ω whose set of arcs is $\{(\alpha, \beta) \in \Omega \times \Omega \,:\, \alpha^g = \beta\}$, and in which all labels are defined to be 0. For every $S \in$ DigraphStacks(Ω), define $f_g(S) = [\Gamma_g]$. Let $g, h \in \mathrm{Sym}(\Omega)$ be arbitrary. Then (f_g, f_g) is a refiner for the centraliser of g in $\mathrm{Sym}(\Omega)$, and (f_g, f_h) is a refiner for the set of conjugating elements $\{x \in \mathrm{Sym}(\Omega) \,:\, g^x = h\}$.

We illustrate one instance of this refiner. Let $g = (1\,2)(3\,6\,5) \in \mathcal{S}_6$ and let K denote the centraliser of g in $\mathcal{S}_6$. A diagram of Γ_g is shown in Fig. 4.9. Note that there is a unique loop, namely at vertex 4, because 4 is the unique fixed point of g on $\{1, \ldots, 6\}$. Since (f_g, f_g) is a refiner for K, Lemma 4.5 implies that $K \leq \mathrm{Aut}([\Gamma_g])$. We prove that in fact $\mathrm{Aut}([\Gamma_g]) = K$, and first note that $\mathrm{Aut}([\Gamma_g]) = \mathrm{Aut}(\Gamma_g)$. Every automorphism of Γ_g stabilises the connected components (because they have different sizes) and so it

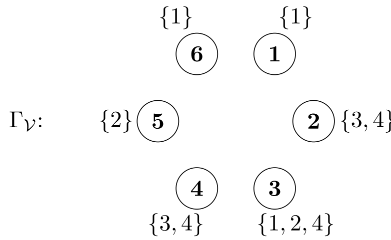

Fig. 4.11. The labelled digraph $\Gamma_{\mathcal{V}}$, for $\mathcal{V} := [\{1,3,6\},\{3,5\},\{2,4\},\{2,3,4\}]$, from Example 4.10.

induces automorphisms on them. Hence $\mathrm{Aut}(\Gamma_g) \leq \langle (1\,2),(3\,5),(3\,6)\rangle$. But none of the transpositions in $\langle (3\,5),(3\,6)\rangle$ is an automorphism of Γ_g, because the arcs between 3, 5, and 6 only go in one direction. Therefore $\mathrm{Aut}(\Gamma_g) = \langle (1\,2),(3\,6\,5)\rangle = K$, as stated.

Example 4.10 *(List of subsets stabiliser and transporter).* Whenever $k \in \mathbb{N}_0$ and $V_i \subseteq \Omega$ for each $i \in \{1,\ldots,k\}$ and $\mathcal{V} := [V_1,\ldots,V_k]$, we let $\Gamma_{\mathcal{V}}$ be the labelled digraph on Ω without arcs, where the label of each vertex $\alpha \in \Omega$ is $\{i \in \{1,\ldots,k\} : \alpha \in V_i\}$. For every $S \in \textsc{DigraphStacks}(\Omega)$, define $f_{\mathcal{V}}(S)$ to be the length-one stack $[\Gamma_{\mathcal{V}}]$. If $g \in \mathrm{Sym}(\Omega)$, then $\mathcal{V}^g := [V_1^g,\ldots,V_k^g]$.

Let $\mathcal{V}$ and $\mathcal{W}$ be arbitrary lists of subsets of Ω with notation as explained above. Then $(f_{\mathcal{V}}, f_{\mathcal{W}})$ is a refiner for the set $\{g \in \mathrm{Sym}(\Omega) : \mathcal{V}^g = \mathcal{W}\}$, and $(f_{\mathcal{V}}, f_{\mathcal{V}})$ is a refiner for the group $\{g \in \mathrm{Sym}(\Omega) : \mathcal{V}^g = \mathcal{V}\}$.

To demonstrate this, let $\mathcal{V} = [\{1,3,6\},\{3,5\},\{2,4\},\{2,3,4\}]$ be a list of subsets of $\{1,\ldots,6\}$. See Fig. 4.11. Since $(f_{\mathcal{V}}, f_{\mathcal{V}})$ is a refiner for $A := \{g \in \mathcal{S}_6 : \mathcal{V}^g = \mathcal{V}\}$, it follows that $\mathrm{Aut}([\Gamma_{\mathcal{V}}])$ (i.e., $\mathrm{Aut}(\Gamma_{\mathcal{V}})$) contains A; indeed, $A = \langle (1\,6),(2\,4)\rangle = \mathrm{Aut}(\Gamma_{\mathcal{V}})$.

Example 4.10 in particular gives refiners for the stabilisers and transporter sets of ordered partitions. If we encode a list $[x_1,\ldots,x_m]$ in Ω as $[\{x_1\},\ldots,\{x_m\}]$, and a subset $\{y_1,\ldots,y_n\} \subseteq \Omega$ as $[\{y_1,\ldots,y_n\}]$, then we see that Example 4.10 can be used to create refiners for the stabilisers and transporters of lists in Ω or subsets of Ω.

For unordered partitions, the following example is applicable.

Example 4.12 *(Refiner for set of subsets stabiliser and transporter).* Let $\mathcal{V}$ be an arbitrary set of subsets of Ω. Let $k \in \mathbb{N}_0$ and $V_i \subseteq \Omega$ for all $i \in \{1,\ldots k\}$ be such that $\mathcal{V} = \{V_1,\ldots,V_k\}$. We define $\Gamma_{\mathcal{V}}$ to be the labelled digraph on Ω whose set of arcs is

$$\{(\alpha,\beta) \in \Omega \times \Omega : \alpha \neq \beta \text{ and } \{\alpha,\beta\} \subseteq V_i \text{ for some } i\};$$

where the label of a vertex α is a list of length $\max\{|V_i| : i \in \{1,\ldots,k\}\}$ with i^{th} entry

$$\textsc{Label}(\alpha)[i] := (|\{j \in \{1,\ldots,k\} : \alpha \in V_j \text{ and } |V_j| = i\}|,\, k),$$

and the label of each arc (α,β) in $\Gamma_{\mathcal{V}}$ is a list of the same length, with i^{th} entry

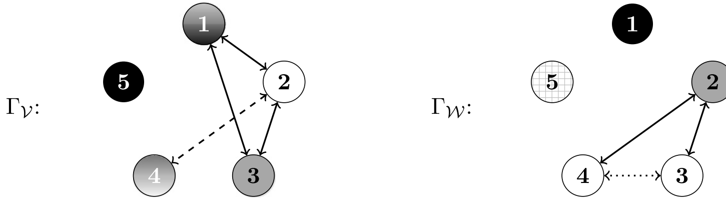

Fig. 4.13. Illustration of the labelled digraphs $\Gamma_{\mathcal{V}}$ and $\Gamma_{\mathcal{W}}$ from Example 4.12, for the sets of subsets $\mathcal{V} := \{\{1\}, \{1, 2, 3\}, \{2, 4\}\}$ and $\mathcal{W} := \{\{5\}, \{2, 3, 4\}, \{3, 4\}\}$ of $\{1, \ldots, 5\}$.

$$\textsc{Label}(\alpha, \beta)[i] := (|\{j \in \{1, \ldots, k\} : \alpha, \beta \in V_j \text{ and } |V_j| = i\}|, k).$$

The connected components of $\Gamma_{\mathcal{V}}$ with at least two vertices are the sets in $\mathcal{V}$ that are not singletons. The label of a vertex (or arc) encodes, for each size of subset, the number of subsets in $\mathcal{V}$ that have that size and contain that vertex (or arc).

For every $S \in \textsc{DigraphStacks}(\Omega)$, we define $f_{\mathcal{V}}(S)$ to be the length-one stack $[\Gamma_{\mathcal{V}}]$. In addition, for all $g \in \mathrm{Sym}(\Omega)$, we define $\mathcal{V}^g = \{V_1^g, \ldots, V_k^g\}$.

Let $\mathcal{V}$ and $\mathcal{W}$ be arbitrary sets of subsets of Ω. Since the labelled digraphs $\Gamma_{\mathcal{V}}$ and $\Gamma_{\mathcal{W}}$ were defined so that $\{g \in \mathrm{Sym}(\Omega) : \mathcal{V}^g = \mathcal{W}\} \subseteq \mathrm{Iso}(\Gamma_{\mathcal{V}}, \Gamma_{\mathcal{W}})$, it follows by Lemma 4.7 that $(f_{\mathcal{V}}, f_{\mathcal{W}})$ is a refiner for the set $\{g \in \mathrm{Sym}(\Omega) : \mathcal{V}^g = \mathcal{W}\}$; in particular, $(f_{\mathcal{V}}, f_{\mathcal{V}})$ is a refiner for the stabiliser group $\{g \in \mathrm{Sym}(\Omega) : \mathcal{V}^g = \mathcal{V}\}$.

As a specific example, we consider the sets $\mathcal{V} := \{\{1\}, \{1, 2, 3\}, \{2, 4\}\}$ and $\mathcal{W} := \{\{5\}, \{2, 3, 4\}, \{3, 4\}\}$. Both $\mathcal{V}$ and $\mathcal{W}$ contain three subsets, which have sizes 1, 2 and 3, so it seems superficially plausible that there exist elements of $\mathcal{S}_5$ that map $\mathcal{V}$ to $\mathcal{W}$.

In order to search for the set $\{g \in \mathcal{S}_5 : \mathcal{V}^g = \mathcal{W}\}$, then (with all the following notation as defined above) we can use the refiner $(f_{\mathcal{V}}, f_{\mathcal{W}})$ to produce stacks $[\Gamma_{\mathcal{V}}]$ and $[\Gamma_{\mathcal{W}}]$ such that $\mathrm{Iso}([\Gamma_{\mathcal{V}}], [\Gamma_{\mathcal{W}}]) = \mathrm{Iso}(\Gamma_{\mathcal{V}}, \Gamma_{\mathcal{W}})$ contains this transporter set. The labelled digraphs $\Gamma_{\mathcal{V}}$ and $\Gamma_{\mathcal{W}}$ are depicted in Fig. 4.13; although we do not give the correspondence explicitly, two vertices or two arcs have the same visual style if and only if they have the same label. There are many ways to show that $\Gamma_{\mathcal{V}}$ and $\Gamma_{\mathcal{W}}$ are non-isomorphic: for example, they have different numbers of arcs. Hence no element of $\mathcal{S}_5$ maps $\mathcal{V}$ to $\mathcal{W}$.

5. Approximating isomorphisms and fixed points of stacks

When searching with labelled digraphs stacks, it might be too expensive to compute the set of isomorphisms exactly, which is why we choose to only approximate this set instead. Our methods always lead to an overestimation of the set, and worse approximations typically lead to larger searches. As a consequence, there is a compromise to be made between the accuracy of such overestimates, and the amount of effort spent in computing them.

In Definition 5.1, we introduce the concept of an isomorphism approximator for pairs of labelled digraphs stacks, which is a vital component of the algorithms in Section 7. Later, we define the approximators that we use in our experiments.

Definition 5.1. An *isomorphism approximator* for labelled digraph stacks is a function Approx that maps a pair of labelled digraph stacks on Ω to either the empty set $\varnothing$, or a right coset of a subgroup of $\mathrm{Sym}(\Omega)$, such that the following statements hold for all $S, T \in \mathrm{DigraphStacks}(\Omega)$ (we usually abbreviate $\mathrm{Approx}(S, S)$ as $\mathrm{Approx}(S)$):

(i) $\mathrm{Iso}(S, T) \subseteq \mathrm{Approx}(S, T)$.
(ii) If $|S| \neq |T|$, then $\mathrm{Approx}(S, T) = \varnothing$.
(iii) If $\mathrm{Approx}(S, T) \neq \varnothing$, then $\mathrm{Approx}(S, T) = \mathrm{Approx}(S) \cdot h$ for some $h \in \mathrm{Sym}(\Omega)$.

Let Approx be an isomorphism approximator and let $S, T \in \mathrm{DigraphStacks}(\Omega)$. The set $\mathrm{Iso}(S, T)$ of isomorphisms induced by $\mathrm{Sym}(\Omega)$ is either empty, or it is a right coset of the induced automorphism group $\mathrm{Aut}(S)$. Since $\mathrm{id}_\Omega \in \mathrm{Iso}(S, S) = \mathrm{Aut}(S)$, it follows by definition that $\mathrm{Approx}(S)$ is a subgroup of $\mathrm{Sym}(\Omega)$ that contains $\mathrm{Aut}(S)$.

The value of $\mathrm{Approx}(S, T)$ should be interpreted as follows. By Definition 5.1(i), $\mathrm{Approx}(S, T)$ gives a true overestimate for $\mathrm{Iso}(S, T)$. Hence if $\mathrm{Approx}(S, T) = \varnothing$, then the approximator has correctly determined that S and T are non-isomorphic. By Definition 5.1(ii), an isomorphism approximator correctly determines that stacks of different lengths are non-isomorphic. Otherwise, the approximator returns a right coset in $\mathrm{Sym}(\Omega)$ of its overestimate for $\mathrm{Aut}(S)$.

In Section 8.1, we need the ability to compute fixed points of the automorphism group (induced by $\mathrm{Sym}(\Omega)$) of any labelled digraph stack. A point $\omega \in \Omega$ is *a fixed point of a subgroup* $G \leq \mathrm{Sym}(\Omega)$ if and only if $\omega^g = \omega$ for all $g \in G$. Computing fixed points is particularly useful when it comes to using orbits and orbital graphs in our search techniques. However, it can be computationally expensive to compute the fixed points exactly, and so we introduce the following definition.

Definition 5.2. A *fixed-point approximator* for labelled digraph stacks is a function Fixed that maps each labelled digraph stack on Ω to a finite list in Ω, such that for each $S \in \mathrm{DigraphStacks}(\Omega)$:

(i) Each entry in $\mathrm{Fixed}(S)$ is a fixed point of $\mathrm{Aut}(S)$, and
(ii) $\mathrm{Fixed}(S)^g = \mathrm{Fixed}(S^g)$ for all $g \in \mathrm{Sym}(\Omega)$.

5.1. Computing automorphisms and isomorphisms exactly

One way to approximate isomorphisms and fixed points of labelled digraph stacks is simply to compute them exactly. For example, we can convert labelled digraph stacks into their squashed labelled digraphs in order to take advantage of existing tools for computing with digraphs.

To describe this formally, we require the concept of a canoniser of labelled digraphs.

Definition 5.3. A *canoniser* of labelled digraphs is a function $\textsc{Canon}$ from the set of labelled digraphs on Ω to $\mathrm{Sym}(\Omega)$ such that, for all labelled digraphs Γ and Δ on Ω, $\Gamma^{\textsc{Canon}(\Gamma)} = \Delta^{\textsc{Canon}(\Delta)}$ if and only if Γ and Δ are isomorphic.

We can use the software BLISS [10] or NAUTY [13] to canonise labelled digraphs, after converting them into vertex-labelled digraphs in a way that preserves isomorphisms.

Definition 5.4 *(Canonising and computing automorphisms exactly).* Let $\textsc{Canon}$ be a canoniser of labelled digraphs. We define functions $\textsc{Fixed}_{\mathrm{C}}$ and $\textsc{Approx}_{\mathrm{C}}$: for all $S, T \in \textsc{DigraphStacks}(\Omega)$, let $g = \textsc{Canon}(\textsc{Squash}(S))$ and $h = \textsc{Canon}(\textsc{Squash}(T))$, let L be the list $[i \in \Omega : i \text{ is fixed by } \mathrm{Aut}(\textsc{Squash}(S)^g)]$, ordered as in Ω, and define

$$\textsc{Fixed}_{\mathrm{C}}(S) = L^{g^{-1}}, \text{ and}$$

$$\textsc{Approx}_{\mathrm{C}}(S, T) = \begin{cases} \mathrm{Aut}(\textsc{Squash}(S)) \cdot gh^{-1} & \text{if } \textsc{Squash}(S)^g = \textsc{Squash}(T)^h, \\ \varnothing & \text{otherwise.} \end{cases}$$

Lemma 5.5. *Let the functions* $\textsc{Approx}_{\mathrm{C}}$ *and* $\textsc{Fixed}_{\mathrm{C}}$ *be given as in Definition 5.4. Then* $\textsc{Approx}_{\mathrm{C}}$ *is an isomorphism approximator, and* $\textsc{Fixed}_{\mathrm{C}}$ *is a fixed-point approximator. Furthermore, for all* $S, T \in \textsc{DigraphStacks}(\Omega)$, $\textsc{Approx}_{\mathrm{C}}(S, T) = \mathrm{Iso}(S, T)$.

Proof. Throughout the proof, we repeatedly use Lemma 3.6 and Definition 5.3. As in Definition 5.4, let $g = \textsc{Canon}(\textsc{Squash}(S))$ and $h = \textsc{Canon}(\textsc{Squash}(T))$.

First, we show that $\textsc{Approx}_{\mathrm{C}}(S, T) = \mathrm{Iso}(S, T)$, which implies that Definition 5.1(i) and (ii) hold. If S and T are non-isomorphic, then $\textsc{Squash}(S)^g \neq \textsc{Squash}(T)^h$, and so $\textsc{Approx}_{\mathrm{C}}(S, T) = \mathrm{Iso}(S, T) = \varnothing$. Otherwise $gh^{-1} \in \mathrm{Iso}(\textsc{Squash}(S), \textsc{Squash}(T)) = \mathrm{Iso}(S, T)$. Therefore

$$\textsc{Approx}_{\mathrm{C}}(S, T) = \mathrm{Aut}(\textsc{Squash}(S)) \cdot gh^{-1} = \mathrm{Aut}(S) \cdot gh^{-1} = \mathrm{Iso}(S, T)\,.$$

Definition 5.1(iii) clearly holds. Therefore $\textsc{Approx}_{\mathrm{C}}$ is an isomorphism approximator.

Define $L = [i \in \Omega : i \text{ is fixed by } \mathrm{Aut}(\textsc{Squash}(S)^g)]$, ordered as usual in Ω. Since $\mathrm{Aut}(S)^g = \mathrm{Aut}(\textsc{Squash}(S))^g = \mathrm{Aut}(\textsc{Squash}(S)^g)$, it follows that L consists of fixed points of $\mathrm{Aut}(S)^g$, and so $\textsc{Fixed}_{\mathrm{C}}(S)$ (which equals $L^{g^{-1}}$) consists of fixed points of $\mathrm{Aut}(S)$. Therefore Definition 5.2(i) holds. To show that Definition 5.2(ii) holds, let $x \in \mathrm{Sym}(\Omega)$ be arbitrary and define $r = \textsc{Canon}(\textsc{Squash}(S^x))$. Since $\textsc{Squash}(S)$ and $\textsc{Squash}(S^x)$ are isomorphic, it follows that $\textsc{Squash}(S)^g = \textsc{Squash}(S^x)^r$. In particular, $g^{-1}xr$ is an automorphism of $\textsc{Squash}(S)^g$, and so $g^{-1}xr$ fixes L. Thus

$$\textsc{Fixed}_{\mathrm{C}}(S)^x = L^{g^{-1}x} = L^{(g^{-1}xr)r^{-1}} = L^{r^{-1}} = \textsc{Fixed}_{\mathrm{C}}(S^x). \quad \square$$

5.2. Approximations via equitable labelled digraphs

We can use vertex labels to overestimate the set of isomorphisms from one labelled digraph to another, because these isomorphisms map the set of vertices with any particular label onto a set of vertices with the same label. In this section we use the term *vertex labelling* as an abbreviation for the restriction of a digraph labelling function to the set of vertices.

In order to present the following approximator functions, we require the notion of an equitable labelled digraph.

5.2.1. Equitable labelled digraphs

Definition 5.6. A labelled digraph $(\Omega, A, \textsc{Label})$ is *equitable* if and only if, for all vertices $\alpha, \beta \in \Omega$ with $\textsc{Label}(\alpha) = \textsc{Label}(\beta)$, and for all labels y and z:

$$|\{(\alpha,\delta) \in A : \textsc{Label}(\delta) = y \text{ and } \textsc{Label}(\alpha,\delta) = z\}| = \\ |\{(\beta,\delta) \in A : \textsc{Label}(\delta) = y \text{ and } \textsc{Label}(\beta,\delta) = z\}|, \text{ and}$$

$$|\{(\delta,\alpha) \in A : \textsc{Label}(\delta) = y \text{ and } \textsc{Label}(\delta,\alpha) = z\}| = \\ |\{(\delta,\beta) \in A : \textsc{Label}(\delta) = y \text{ and } \textsc{Label}(\delta,\beta) = z\}|.$$

In other words, the labelled digraph is equitable if and only if, for all labels x, y, and z, every vertex with label x has some common number of *out-neighbours* with label y via arcs with label z, and similarly, every vertex with label x has some common number of *in-neighbours* with label y via arcs with label z.

Definition 5.6 extends the well-known concepts of equitable colourings [13, Section 3.1] and partitions [8, Definition 29] of vertex-labelled graphs and digraphs. The traditional notion requires that for all labels y and z, there are constants for the number of arcs from each vertex with label y to vertices with label z, and for the number of arcs in the other direction. Definition 5.6 additionally takes arc labels into account.

It is possible to define a procedure that takes a labelled digraph $\Gamma := (\Omega, A, \textsc{Label})$ and 'refines' the vertex labelling to obtain a labelled digraph $\Gamma' := (\Omega, A, \textsc{Label}')$, which uses the fewest possible labels such that: arc labels are unchanged, vertices with the same label in Γ' have the same label in Γ, and Γ' is equitable. Moreover, this can be done consistently between labelled digraphs Γ and Δ, such that the overestimate of $\mathrm{Iso}(\Gamma, \Delta)$ that can be obtained from the equitable labelled digraphs is contained in the overestimate from the original vertex labels. We present an example of such a procedure, phrased as an algorithm, as Algorithm 4.8 in [9]. Here we just describe the idea of such an "equitable vertex labelling algorithm" and abbreviate it as EVLA.

Given a labelled digraph, EVLA repeatedly tests whether each set of vertices with the same label satisfies the condition in Definition 5.6. For each such set, either the condition

is satisfied, and a new label for this set is devised that encodes information about how the condition was satisfied, or the condition is not satisfied, and the vertices are given new labels accordingly, which encode information about why they were created.

We define EQUITABLE to be a function defined by EVLA that maps each labelled digraph to a list of pairs of the form (x, W), for some label $x \in \mathfrak{L}$ and non-empty $W \subseteq \Omega$, sorted by first component (recall that $\mathfrak{L}$ is totally ordered). This list encodes that the vertices in W are those with label x in the equitable digraph given by EVLA.

In the following lemma, we present some properties of EQUITABLE. For a more detailed discussion we refer to [9], and we omit the proof of the lemma because it is mathematically straightforward.

Lemma 5.7. *Let Γ and Δ be labelled digraphs on Ω, and define $k, l \in \mathbb{N}_0$, labels $x_1, \ldots, x_k, y_1, \ldots, y_l$, and partitions $\{V_1, \ldots, V_k\}$ and $\{W_1, \ldots, W_l\}$ of Ω such that*

$$\textsc{Equitable}(\Gamma) = [(x_1, V_1), \ldots, (x_k, V_k)] \textit{ and } \textsc{Equitable}(\Delta) = [(y_1, W_1), \ldots, (y_l, W_l)].$$

Then the following hold:

(i) $\textsc{Equitable}(\Gamma^g) = [(x_1, V_1^g), \ldots, (x_k, V_k^g)]$ *for all* $g \in \mathrm{Sym}(\Omega)$.

(ii) $\mathrm{Iso}(\Gamma, \Delta) \begin{cases} = \varnothing, & \textit{if } k \neq l, \textit{ or } k = l \textit{ and } x_i \neq y_i \textit{ for some } i, \\ \subseteq \{g \in \mathrm{Sym}(\Omega) : [V_1^g, \ldots, V_k^g] = [W_1, \ldots, W_k]\}, & \textit{otherwise.} \end{cases}$

By choosing meaningful new vertex labels as described, we can distinguish more pairs of labelled digraphs as non-isomorphic via Lemma 5.7(ii) than we can by defining new labels arbitrarily. The next example illustrates this principle.

Example 5.8. Let Γ be the labelled digraph on Ω with all possible arcs, and let Δ be the labelled digraph on Ω without arcs, where every vertex and arc in Γ and Δ has the label x, for some arbitrary but fixed label $x \in \mathfrak{L}$.

Then we may use EVLA to deduce that Γ and Δ are non-isomorphic, even though both are regular (i.e. every vertex has a common number of in-neighbours, and a common number of out-neighbours). The new labels encode that each vertex in Γ and Δ has $|\Omega|$ in- and out-neighbours, or zero in- or out-neighbours, respectively. Therefore, the labels given by EQUITABLE(Γ) and EQUITABLE(Δ) are different, and so Γ and Δ are non-isomorphic by Lemma 5.7(ii).

A note of warning: the choice of new labels plays a role! If new labels were instead, say, chosen to be incrementally increasing integers starting at 1, then we would have EQUITABLE(Γ) = EQUITABLE(Δ), and the above deduction would not be possible.

In the previous example it is obvious to us that the digraphs are non-isomorphic, but for many more complicated examples, Lemma 5.7(ii) can still be used to detect less obvious non-isomorphism.

5.2.2. Strong and weak approximations via equitable labelled digraphs

We describe two strategies for using EVLA (which operates on labelled digraphs) to approximate isomorphisms and fixed points of *stacks* of labelled digraphs. In the first approach, we first combine the entries of a stack into a single digraph, namely the squashed labelled digraph of the stack, and then apply EVLA; in the other, we first apply EVLA to each of the entries in the stack, and then combine the information that we obtain. We call these approaches *strong* and *weak equitable approximation*, respectively, and we give an example of their use in Section 5.3.

Definition 5.9 *(Strong equitable approximation).* We define functions $\textsc{Approx}_\mathrm{S}$ and $\textsc{Fixed}_\mathrm{S}$ as follows. Let $S, T \in \textsc{DigraphStacks}(\Omega)$. Then there exist $k, l \in \mathbb{N}_0$, labels $x_1, \ldots, x_k$, $y_1, \ldots, y_l$, and partitions $\{V_1, \ldots, V_k\}$ and $\{W_1, \ldots, W_l\}$ of Ω such that

$$\textsc{Equitable}(\textsc{Squash}(S)) = [(x_1, V_1), \ldots, (x_k, V_k)], \text{ and}$$
$$\textsc{Equitable}(\textsc{Squash}(T)) = [(y_1, W_1), \ldots, (y_l, W_l)].$$

Let G denote the stabiliser of the list $[V_1, \ldots, V_k]$ in $\mathrm{Sym}(\Omega)$, and define

$$\textsc{Approx}_\mathrm{S}(S, T) = \begin{cases} G \cdot h & \text{if } |S| = |T|,\ k = l, \text{ and for all } i,\ x_i = y_i \text{ and } |V_i| = |W_i|; \\ \varnothing & \text{otherwise,} \end{cases}$$

where $h \in \mathrm{Sym}(\Omega)$ is any permutation such that $V_i^h = W_i$ for all $i \in \{1, \ldots, k\}$. This is well-defined because, for all $g, h \in \mathrm{Sym}(\Omega)$, we have that $V_i^g = V_i^h$ for all i if and only if g and h represent the same right coset of G in $\mathrm{Sym}(\Omega)$. Finally, we define

$$\textsc{Fixed}_\mathrm{S}(S) = [v_{i_1}, \ldots, v_{i_m}],$$

where $i_1 < \cdots < i_m$ and the sets $V_{i_j} = \{v_{i_j}\}$ for each $j \in \{1, \ldots, m\}$ are exactly the singletons amongst $V_1, \ldots, V_k$.

Definition 5.10 *(Weak equitable approximation).* We define functions $\textsc{Approx}_\mathrm{W}$ and $\textsc{Fixed}_\mathrm{W}$ as follows. Let $S, T \in \textsc{DigraphStacks}(\Omega)$. For each $i \in \{1, \ldots, |S|\}$, $j \in \{1, \ldots, |T|\}$, there exist $k_i, l_j \in \mathbb{N}_0$, labels $x_{i,1}, \ldots, x_{i,k_i}, y_{j,1}, \ldots, y_{j,l_j}$, and partitions $\{V_{i,1}, \ldots, V_{i,k_i}\}$ and $\{W_{j,1}, \ldots, W_{j,l_j}\}$ of Ω such that

$$\textsc{Equitable}(S[i]) = [(x_{i,1}, V_{i,1}), \ldots, (x_{i,k_i}, V_{i,k_i})], \text{ and}$$
$$\textsc{Equitable}(T[j]) = [(y_{j,1}, W_{j,1}), \ldots, (y_{j,l_j}, W_{j,l_j})].$$

If $|S| \neq |T|$, or else if $k_i \neq l_i$ for some $i \in \{1, \ldots, |S|\}$, or else if $x_{i,j} \neq y_{i,j}$ for some $i \in \{1, \ldots, |S|\}$ and $j \in \{1, \ldots, k_i\}$, then we define $\textsc{Approx}_\mathrm{W}(S, T) = \varnothing$.

Suppose otherwise. We define functions f and g that map vertices to lists of length $|S|$ with entries in $\mathbb{N}$. For each $\alpha \in \Omega$, the list entry $f(\alpha)[i]$ is the unique $j \in \{1, \ldots, k_i\}$

such that $\alpha \in V_{i,j}$, and $g(\alpha)[i]$ is the unique $j \in \{1, \ldots, k_i\}$ such that $\alpha \in W_{i,j}$. Thus f and g encode the 'equitable' label of a vertex in each entry of S and T, respectively. Then we partition Ω into subsets $A_1, \ldots, A_m$ according to, and ordered lexicographically by, f-value, and similarly we partition Ω into subsets $B_1, \ldots, B_n$ via g.

Given all of this, we let G denote the stabiliser of $[A_1, \ldots, A_m]$ in $\mathrm{Sym}(\Omega)$ and define

$$\textsc{Approx}_{\mathrm{W}}(S, T) = \begin{cases} G \cdot h & \text{if } |S| = |T|,\ m = n, \text{ and for all } i, \\ & \quad |A_i| = |B_i| \text{ and } f(\min(A_i)) = g(\min(B_i)), \\ \varnothing & \text{otherwise,} \end{cases}$$

where $h \in \mathrm{Sym}(\Omega)$ is any permutation such that $A_i^h = B_i$ for all $i \in \{1, \ldots, m\}$, and $\min(A_i)$ is the minimum vertex in A_i with respect to the ordering of Ω. We also define

$$\textsc{Fixed}_{\mathrm{W}}(S) = [a_{i_1}, \ldots, a_{i_t}],$$

where $i_1 < \cdots < i_t$ and the sets $A_{i_j} = \{a_{i_j}\}$ for each $j \in \{1, \ldots, t\}$ are exactly the singletons amongst $A_1, \ldots, A_m$.

The following lemma holds by Lemma 5.7.

Lemma 5.11. *The functions from Definitions 5.9 and 5.10,* $\textsc{Approx}_{\mathrm{S}}$ *and* $\textsc{Approx}_{\mathrm{W}}$*, and* $\textsc{Fixed}_{\mathrm{S}}$ *and* $\textsc{Fixed}_{\mathrm{W}}$*, are isomorphism and fixed-point approximators, respectively.*

5.3. Comparing approximators

In this section, we give a simple example to compare the isomorphism approximators from Sections 5.1 and 5.2. We present the example in more detail in [9].

Weak equitable approximations should be the least accurate but cheapest to compute, whereas computing isomorphisms exactly should be the most expensive. Weak equitable approximation distinguishes vertices by distinguishing them in the individual entries of the stacks. Strong equitable approximation sometimes gives better results than this, because it considers the entire stacks simultaneously.

Example 5.12. Let $\Gamma_1, \Gamma_2, \Delta_1$, and Δ_2 be labelled digraphs on $\{1, \ldots, 6\}$ whose arcs and arc labels are defined as in Fig. 5.13, and where each vertex is labelled *white*. We approximate the isomorphisms from the stack $S := [\Gamma_1, \Gamma_2]$ to the stack $T := [\Delta_1, \Delta_2]$.

Weak equitable approximation. The labelled digraphs $\Gamma_1, \Gamma_2, \Delta_1$, and Δ_2 are equitable, and their vertices are all *white*. Therefore EVLA makes no progress, and so weak equitable approximation gives the worst possible result $\textsc{Approx}_{\mathrm{W}}(S, T) = \mathcal{S}_6$.

Strong equitable approximation. To see that the labelled digraphs $\textsc{Squash}(S)$ and $\textsc{Squash}(T)$ are not equitable, note for example that there are vertices in each of these

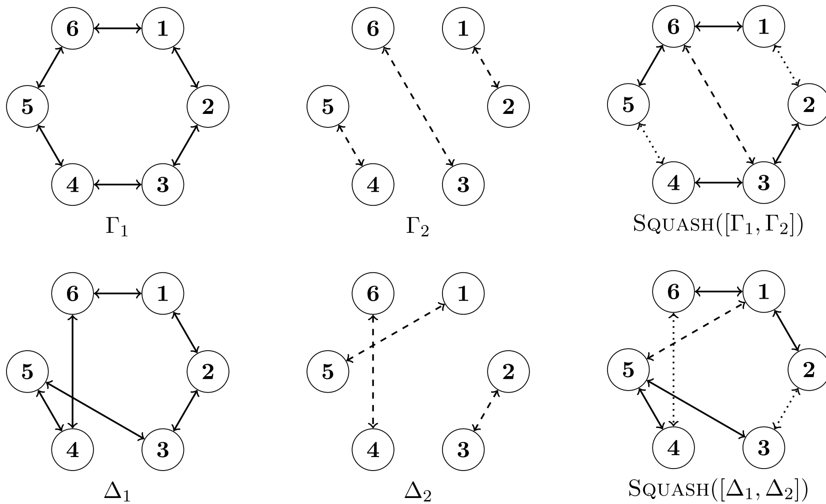

Fig. 5.13. Pictures of the labelled digraphs on $\{1, \ldots, 6\}$ from Example 5.12. Each arc in $\Gamma_1, \Gamma_2, \Delta_1$, and Δ_2 is labelled *solid* or *dashed* according to its depiction. Every vertex in Squash($[\Gamma_1, \Gamma_2]$) and Squash($[\Delta_1, \Delta_2]$) has the same label [*white*, *white*]; arcs with label [*solid*, #] are shown as *solid*, arcs with label [#, *dashed*] are shown as *dashed*, and arcs with label [*solid*, *dashed*] are shown as *dotted*.

digraphs with different numbers of out-neighbours, yet all vertices have the same label. There exist labels x and y such that the EVLA assigns x to $\{3, 6\}$ and y to $\{1, 2, 4, 5\}$ in Squash(S), and it assigns x to $\{1, 5\}$ and y to $\{2, 3, 4, 6\}$ in Squash(T).

Let G be the stabiliser of $[\{3, 6\}, \{1, 2, 4, 5\}]$ in $\mathcal{S}_6$, and let $g \in \mathcal{S}_6$ be any permutation that maps $\{3, 6\}$ to $\{1, 5\}$ and $\{1, 2, 4, 5\}$ to $\{2, 3, 4, 6\}$. Then strong equitable approximation gives $\text{Approx}_\text{S}(S, T) = G \cdot h$. Note that $|\text{Approx}_\text{S}(S, T)| = |G| = 2! \cdot 4! = 48$.

Canonising and computing exactly. We compute with Bliss [10] via the GAP [3] package Digraphs [1] that $\text{Aut}(\text{Squash}(S)) = \langle (1\,2)(3\,6)(4\,5), (1\,4)(2\,5)(3\,6) \rangle =: G$ and that $\text{Squash}(S)^{(1\,2\,3\,5\,6)} = \text{Squash}(T)$. Thus $\text{Iso}(S, T) = G \cdot (1\,2\,3\,5\,6)$. In particular, $|\,\text{Iso}(S, T)\,| = |G| = 4$, which reveals the inaccuracy of the other approximators.

6. Distributing stack isomorphisms across new stacks

In a backtrack search, when it is not clear how to further prune a search space, we divide the search across a number of smaller areas that can be searched more easily. We call this process *splitting*, and in this section we define the notion of a *splitter* for labelled digraph stacks. A splitter takes a pair of stacks that represents a (potentially large) search space, and defines new stacks that divide the space in a sensible way.

Definition 6.1. A *splitter* for an isomorphism approximator Approx is a function Split that maps a pair of labelled digraph stacks on Ω to a finite list of stacks, where for all $S, T \in \text{DigraphStacks}(\Omega)$ with $|\text{Approx}(S, T)| \geq 2$,

$$\text{Split}(S, T) = [S_1, T_1, T_2, \ldots, T_m]$$

for some $m \in \mathbb{N}_0$ and $S_1, T_1, \ldots, T_m \in \textsc{DigraphStacks}(\Omega)$, such that:

(i) $\mathrm{Iso}(S, T) = \mathrm{Iso}(S \,\|\, S_1, T \,\|\, T_1) \cup \cdots \cup \mathrm{Iso}(S \,\|\, S_1, T \,\|\, T_m)$.
(ii) $|\textsc{Approx}(S \,\|\, S_1, T \,\|\, T_i)| < |\textsc{Approx}(S, T)|$ for all $i \in \{1, \ldots, m\}$.
(iii) For all $U \in \textsc{DigraphStacks}(\Omega)$ with $|\textsc{Approx}(S, U)| \geq 2$, the first entry of $\textsc{Split}(S, U)$ is S_1.

For this paragraph and the following remark, we keep the notation from Definition 6.1, with $|\textsc{Approx}(S, T)| \geq 2$. The search space corresponding to the pair (S, T) is $\textsc{Approx}(S, T)$. By Definition 6.1(i), if $|\textsc{Approx}(S, T)| \geq 2$, then the splitter produces the search spaces $\textsc{Approx}(S \,\|\, S_1, T \,\|\, T_i)$ for each $i \in \{1, \ldots, m\}$; note that the left-hand stack $S \,\|\, S_1$ does not vary here. Each one of these new search spaces is smaller than $\textsc{Approx}(S, T)$, by Definition 6.1(ii). This is required to show that our algorithms terminate. Definition 6.1(iii) means that the first stack given by a splitter is independent of the given right-hand stack. This is required by the technique in Section 8.

Remark 6.2. If $S = T$, then it follows by Definition 6.1(i) that $S_1 = T_i$ for some i. Thus we may assume without loss of generality that $S_1 = T_1$ in this case.

The following lemma shows a way of giving a splitter by specifying its behaviour on the left stack that it is given. The proof is straightforward and therefore omitted; see [9].

Lemma 6.3. *Let* $\textsc{Approx}$ *be an isomorphism approximator, and let* f *be any function from* $\textsc{DigraphStacks}(\Omega)$ *to itself such that, for all* $S \in \textsc{DigraphStacks}(\Omega)$*:*

$$\text{if } |\textsc{Approx}(S)| \geq 2, \text{ then } |\textsc{Approx}(S \,\|\, f(S))| < |\textsc{Approx}(S)|.$$

Let $S, T \in \textsc{DigraphStacks}(\Omega)$*, and fix an ordering* $T_1, \ldots, T_m$*, for some* $m \in \mathbb{N}_0$*, of the set* $\{f(S)^g : g \in \textsc{Approx}(S, T)\}$*. Finally, let* $\textsc{Split}_f(S, T) = [f(S), T_1, \ldots, T_m]$*. Then* $\textsc{Split}$ *is a splitter for* $\textsc{Approx}$*.*

Let the notation of Lemma 6.3 hold. Splitting by appending the stack $f(S)$ to the stack S corresponds to stabilising $f(S)$ in the current approximation of $\mathrm{Aut}(S)$; the stacks of the form T_i give the images of the stack $f(S)$ under $\textsc{Approx}(S, T)$.

Note that the set $\{f(S)^g : g \in \textsc{Approx}(S, T)\}$ can be computed via the orbit of $f(S)$ under $\textsc{Approx}(S)$. Indeed, if $h \in \textsc{Approx}(S, T)$, then since $\textsc{Approx}(S, T) = \textsc{Approx}(S) \cdot h$ by Definition 5.1(iii), we have

$$\begin{aligned}\{f(S)^g : g \in \textsc{Approx}(S, T)\} &= \{f(S)^g : g \in \textsc{Approx}(S) \cdot h\} \\ &= \{f(S)^x : x \in \textsc{Approx}(S)\}^h = \left(f(S)^{\textsc{Approx}(S)}\right)^h.\end{aligned}$$

In the following definition, we present a splitter that can be obtained with Lemma 6.3. We use a version of this splitter for our experiments in Section 9.

Definition 6.4 *(Fixed point splitter).* For all $\alpha \in \Omega$, let $\Gamma_\alpha = (\Omega, \varnothing, \textsc{Label})$ be the labelled digraph on Ω where $\textsc{Label}(\alpha) = 1$ and $\textsc{Label}(\beta) = 0$ for all $\beta \in \Omega \setminus \{\alpha\}$. Note that $\Gamma_\alpha^g = \Gamma_{\alpha^g}$ for all $g \in \mathrm{Sym}(\Omega)$. Let $\textsc{Approx}$ be any isomorphism approximator such that $\textsc{Approx}(U \,\|[\Gamma_\alpha]) \leq \textsc{Approx}(U) \cap \{g \in \mathrm{Sym}(\Omega) : \alpha^g = \alpha\}$ for all $\alpha \in \Omega$ and $U \in \textsc{DigraphStacks}(\Omega)$. We define a function σ from $\textsc{DigraphStacks}(\Omega)$ to itself by

$$\sigma(S) = \begin{cases} \textsc{EmptyStack}(\Omega) & \text{if } |\textsc{Approx}(S)| \leq 1, \\ [\Gamma_\alpha] & \text{otherwise, where } \alpha := \min\{\min(\mathcal{O}) : \mathcal{O} \text{ is an orbit of} \\ & \qquad \textsc{Approx}(S) \text{ of minimal size, subject to } |\mathcal{O}| \geq 2\}, \end{cases}$$

for all $S \in \textsc{DigraphStacks}(\Omega)$. Finally, define $\textsc{Split}_\sigma$ as in Lemma 6.3, for the isomorphism approximator $\textsc{Approx}$ and the function σ.

The following corollary holds by Lemma 6.3, bearing in mind that the function σ has the required property by our choice of isomorphism approximator in Definition 6.4.

Corollary 6.5. *The function* $\textsc{Split}_\sigma$ *from Definition 6.4 is a splitter for any isomorphism approximator that satisfies the condition in Definition 6.4.*

7. The search algorithm

Let $U_1, \ldots, U_k \subseteq \mathrm{Sym}(\Omega)$. In this section, we present our main algorithms, which combine the tools of Sections 3–6 to compute the intersection $U_1 \cap \cdots \cap U_k$. In Section 7.1, we show how to perform a backtrack search for one or all of the elements of $U_1 \cap \cdots \cap U_k$. In Section 7.2, when the result is known to form a group, we show how to search for a base and strong generating set instead (see [2, p. 101] for a definition). We explain these algorithms in further detail in [9]. A version of our algorithms is implemented in the GraphBacktracking package [6] for GAP [3].

7.1. The basic method

We begin with a high-level description of Algorithm 7.1, which comprises the Search and Refine procedures. We say that an algorithm *backtracks* when it finishes executing a recursive call to a procedure, and continues executing from where the call was initiated.

The algorithm begins with a call to the Search procedure on line 21. This procedure, when given labelled digraph stacks S and T, finds those elements of $U_1 \cap \cdots \cap U_k$ that induce isomorphisms from S to T (Lemma 7.4). It does so by searching in $\textsc{Approx}(S, T)$ rather than in $\mathrm{Iso}(S, T)$, because we do not necessarily wish to compute $\mathrm{Iso}(S, T)$ exactly.

The Search procedure first calls Refine. This applies the refiners in turn, aiming to prune the search space (Lemma 7.3). Then, if the remaining search space contains at most one element, the Search procedure backtracks, having potentially returned an element, if appropriate. Otherwise, it divides the search with a splitter, and recurses.

Algorithm 7.1 A recursive algorithm using labelled digraph stacks to search in $\mathrm{Sym}(\Omega)$.

Input: a sequence of subsets $U_1, \ldots, U_k \subseteq \mathrm{Sym}(\Omega)$;
a sequence $(f_{L,1}, f_{R,1}), \ldots, (f_{L,m}, f_{R,m})$, where each pair is a refiner for some U_j;
an isomorphism approximator APPROX and a splitter SPLIT for APPROX.
Output: all elements of the intersection $U_1 \cap \cdots \cap U_k$, which we refer to as *solutions*.

```
 1: procedure SEARCH(S, T)                      ▷ The main recursive search procedure (Lemma 7.4).
 2:     (S, T) ← REFINE(S, T)                                      ▷ Refine the given stacks.
 3:     case APPROX(S, T) = ∅:                ▷ Nothing found in the present branch: backtrack.
 4:         return ∅
 5:     case APPROX(S, T) = {h} for some h:           ▷ h is the sole potential solution here.
 6:         if S^h = T and h ∈ U_1 ∩ ··· ∩ U_k then
 7:             return {h}                  ▷ h is the unique solution in Iso(S, T): backtrack.
 8:         else
 9:             return ∅                     ▷ h is not a solution in Iso(S, T): backtrack.
10:     case |APPROX(S, T)| ≥ 2:                               ▷ Multiple potential solutions.
11:         [S_1, T_1, ..., T_t] ← SPLIT(S, T)                      ▷ Split the search space.
12:         return ⋃_{i∈{1,...,t}} SEARCH(S ‖ S_1, T ‖ T_i)               ▷ Search recursively.
13: procedure REFINE(S, T)                   ▷ Attempt to prune the search space (Lemma 7.3).
14:     while APPROX(S, T) ≠ ∅ do                ▷ Proceed while there are potential solutions.
15:         (S', T') ← (S, T)                 ▷ Save the stacks before the next round of refining.
16:         for i ∈ {1, ..., m} and while |S| = |T| do
17:             (S, T) ← (S ‖ f_{L,i}(S), T ‖ f_{R,i}(T))              ▷ Apply each refiner in turn.
18:         if |APPROX(S, T)| ≮ |APPROX(S', T')| then
19:             return (S', T')       ▷ Stop: the last refinements seemingly made no progress.
20:     return (S, T)               ▷ Stop: APPROX(S, T) = ∅: no solutions in this branch.
21: return SEARCH(EMPTYSTACK(Ω), EMPTYSTACK(Ω))
```

We assume the given approximator, splitter, and refiners are possible to compute.

We thus claim that, given the specified inputs, and after a finite number of steps, Algorithm 7.1 returns the intersection $U_1 \cap \cdots \cap U_k$.

Theorem 7.2. *Algorithm 7.1 is correct.*

The proof of Theorem 7.2 relies on the following lemmas. In particular, it follows from Lemma 7.4 by setting $S = T = \textsc{EmptyStack}(\Omega)$.

Lemma 7.3. *Let the notation of Algorithm 7.1 hold. Then the* REFINE *procedure terminates after a finite number of steps, with*

(i) $|\textsc{Approx}(\textsc{Refine}(S, T))| \le |\textsc{Approx}(S, T)|$, *and*
(ii) $(U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(S, T) = (U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(\textsc{Refine}(S, T))$.

Proof. The REFINE procedure performs finitely many iterations of its *while* loop, and so terminates, since a new iteration occurs only if the prior one yields a smaller search space. It is evident in the definition of REFINE that (i) holds. To prove (ii), note that $\textsc{Refine}(S, T)$ is obtained from (S, T) by the repeated application of refiners to stacks of equal length (line 17). Thus it suffices to show that if $i \in \{1, \ldots, m\}$ and $|S| = |T|$, then

$$(U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(S, T) = (U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(S \parallel f_{L,i}(S), T \parallel f_{R,i}(T)) .$$

If $\mathrm{Iso}(S, T) = \varnothing$, then also $\mathrm{Iso}(S \parallel f_{L,i}(S), T \parallel f_{R,i}(T)) = \varnothing$ by Remark 3.1, thereby satisfying the equation. Otherwise, the equation holds by Lemma 4.2(ii). □

Lemma 7.4. *Let the notation of Algorithm 7.1 hold. Then the* Search *procedure terminates with* $\textsc{Search}(S, T) = (U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(S, T)$.

Proof. We proceed by induction on $|\textsc{Approx}(\textsc{Refine}(S, T))|$, bearing in mind Definition 5.1(i) and Lemma 7.3(ii). If $|\textsc{Approx}(\textsc{Refine}(S, T))| \in \{0, 1\}$, then it is straightforward to verify that the Search procedure terminates with the correct value.

Let $n \in \mathbb{N}$ with $n \geq 2$, and assume the statement holds for all stacks S and T with $|\textsc{Approx}(\textsc{Refine}(S, T))| < n$. If $|\textsc{Approx}(\textsc{Refine}(S, T))| = n$, then on line 11, the splitter gives a finite list of stacks, giving a finite number of calls to Search. By Definition 6.1(i) and (ii), by Lemma 7.3(i), and by assumption, these recursive calls terminate with values whose union is $(U_1 \cap \cdots \cap U_k) \cap \mathrm{Iso}(\textsc{Refine}(S, T))$. The result follows by Lemma 7.3(ii), and by induction. □

Remark 7.5. Algorithm 7.1 finds *all* elements of $U_1 \cap \cdots \cap U_k$. To search for a *single* element, if one exists, one can modify the Search procedure to return a result on line 7, as soon as the first solution is found. We name this modified procedure SearchSingle. Thus $\textsc{SearchSingle}(S, T)$ gives a single element of $\mathrm{Iso}(S, T) \cap (U_1 \cap \cdots \cap U_k)$, if one exists, else $\varnothing$. This is especially useful when one wishes to find an isomorphism from one combinatorial structure to another, or to prove that they are non-isomorphic.

Algorithm 7.6 Search for a base and strong generating set of a subgroup of $\mathrm{Sym}(\Omega)$.

Input: as in Algorithm 7.1, plus the assumption that $U_1 \cap \cdots \cap U_k$ is a subgroup.
Output: a base and strong generating set of the subgroup $U_1 \cap \cdots \cap U_k$.

```
1: procedure SearchBSGS(S)                                  ▷ See Lemma 7.9.
2:     (S, S) ← Refine(S, S)                                ▷ Refine the given stacks.
3:     case Approx(S, S) = {id_Ω}:
4:         return (EmptyBase, {id_Ω})                       ▷ A BSGS for the trivial group.
5:     case |Approx(S, S)| ≥ 2:
6:         [S_1, S_1, ..., S_t] ← Split(S, S)               ▷ See Remark 6.2.
7:         (Base, X) ← SearchBSGS(S ‖ S_1)                  ▷ Find BSGS for the stabiliser of S_1.
8:         Base ← [S_1] ‖ Base                              ▷ Prepend S_1 to the base for the stabiliser of S_1.
9:         for i ∈ {2, ..., t} do
10:            if S_i ∉ S_j^⟨X⟩ for any j ∈ {1, ..., i − 1} then   ▷ Pruning.
11:                X ← X ∪ SearchSingle(S ‖ S_1, S ‖ S_i)   ▷ Search for a coset rep.
12:        return (Base, X)
13: procedure Refine(S, T)                       ▷ The Refine procedure from Algorithm 7.1.
14: procedure SearchSingle(S, T)                 ▷ The procedure from Remark 7.5.
15: return SearchBSGS(EmptyStack(Ω))
```

7.2. Searching for a generating set of a subgroup

It is usually most efficient to compute with a permutation group via a base and strong generating set (BSGS). In the standard definition (see [2, p. 101]), a base for a subgroup G of $\mathrm{Sym}(\Omega)$ is a list of points in Ω whose stabiliser in G is trivial. Here, we use a broader definition, where the list may contain *any* objects on which G acts.

In this section we present Algorithm 7.6, which, given subsets $U_1, \ldots, U_k \subseteq \mathrm{Sym}(\Omega)$ whose intersection is a subgroup, returns a base and strong generating set for $U_1 \cap \cdots \cap U_k$. The base is given as a list of labelled digraph stacks. This algorithm is derived from Algorithm 7.1: the first two cases simplify since $\textsc{Approx}(S, S)$ is a subgroup, and the recursive case turns into a search for a stabiliser and coset representatives. Note that Algorithm 7.6 uses the partially-constructed generating set to prune the search on line 10. It thus usually performs a smaller search than does Algorithm 7.1 with the same input.

Remark 7.7. To obtain a base consisting of points in Ω, one can use the splitter from Definition 6.4: using its notation, Algorithm 7.6 returns a base $[[\Gamma_{\alpha_1}], \ldots, [\Gamma_{\alpha_r}]]$. For each $\alpha \in \Omega$, a permutation fixes $[\Gamma_\alpha]$ if and only if it fixes α, so a generating set is strong with respect to $[[\Gamma_{\alpha_1}], \ldots, [\Gamma_{\alpha_r}]]$ if and only if it is strong with respect to $[\alpha_1, \ldots, \alpha_r]$.

Remark 7.8. Algorithm 7.6 is useful when searching for an intersection of cosets. Let the notation of Algorithm 7.6 hold, and suppose that each set U_i is a coset of a subgroup of $\mathrm{Sym}(\Omega)$. We can use the $\textsc{SearchSingle}$ procedure to find some $g \in U_1 \cap \cdots \cap U_k$, or to prove that no such element exists. In the former case, then for all $i \in \{1, \ldots, k\}$, $(f_{L,i}, f_{L,i})$ is a refiner for the group $U_i g^{-1}$ by Lemma 4.6. Therefore we may use Algorithm 7.6 to search for a base and strong generating set of $U_1 g^{-1} \cap \cdots \cap U_k g^{-1}$, which in combination with g compactly describes $U_1 \cap \cdots \cap U_k$.

Lemma 7.9. *Let the notation of Algorithm 7.6 hold. Then the* $\textsc{SearchBSGS}$ *procedure, given S, terminates with a base and strong generating set of* $\mathrm{Aut}(S) \cap (U_1 \cap \cdots \cap U_k)$.

Proof. The $\textsc{SearchBSGS}$ procedure first calls the $\textsc{Refine}$ procedure to prune the search (see Lemma 7.3). This returns a pair of equal stacks, since it is given equal stacks, and each refiner is a pair of equal functions (see Lemma 4.4). That one of the conditions on either line 3 or line 5 is satisfied follows by Lemma 5.1(i). The procedure accordingly returns a trivial solution, or searches recursively.

The $\textsc{SearchBSGS}$ procedure only differs significantly from the $\textsc{Search}$ procedure of Algorithm 7.1 in its recursive step. Let $\textsc{Split}(S, S) = [S_1, S_1, S_2, \ldots, S_t]$ as on line 6. Then $\mathrm{Aut}(S \,\|\, S_1)$ is the stabiliser of S_1 in $\mathrm{Aut}(S)$ by Remark 3.1, and by Definition 6.1 and Remark 6.2, its right cosets in $\mathrm{Aut}(S)$ are the non-empty sets $\mathrm{Iso}(S \,\|\, S_1, S \,\|\, S_i)$ for $i \in \{2, \ldots, t\}$. Analogous statements hold for the stabiliser $\mathrm{Aut}(S \,\|\, S_1) \cap (U_1 \cap \cdots \cap U_k)$ of S_1 in $\mathrm{Aut}(S) \cap (U_1 \cap \cdots \cap U_k)$, and for the sets $\mathrm{Iso}(S \,\|\, S_1, S \,\|\, S_i) \cap (U_1 \cap \cdots \cap U_k)$. It is thus possible to build a BSGS of $\mathrm{Aut}(S) \cap (U_1 \cap \cdots \cap U_k)$ recursively from a BSGS of

the stabiliser of S_1 in $\mathrm{Aut}(S) \cap (U_1 \cap \cdots \cap U_k)$, along with representatives of a sufficient selection of the right cosets of the stabiliser, as is done in lines 7–11.

The validity of the recursion in the SEARCHBSGS procedure can be shown by induction on $|\text{APPROX}(\text{REFINE}(S, S))|$, as in the proof of Lemma 7.4. It follows by Remark 7.5 that, on line 11, the SEARCHSINGLE procedure returns the desired representatives of the sets $\mathrm{Iso}(S \,\|\, S_1, S \,\|\, S_i) \cap (U_1 \cap \cdots \cap U_k)$. □

Theorem 7.10. *Algorithm 7.6 is correct.*

Proof. Set $S = T = \text{EMPTYSTACK}(\Omega)$ in Lemma 7.9. □

8. Searching with a fixed sequence of left-hand stacks

In this section, we discuss a consequence of our definitions and the setup of our algorithms, which enables a significant performance optimisation, and which allows us to use a special kind of refiner. This idea was inspired by, and is closely related to, the $\mathfrak{R}$-base technique of Jeffrey Leon [12, Section 6] for partition backtrack search, although we present the idea quite differently.

Recall that Algorithms 7.1 and 7.6 are organised around a pair of labelled digraph stacks (the stacks are equal in the SEARCHBSGS procedure), with both stacks initially equal to $\text{EMPTYSTACK}(\Omega)$. We observe that when each of these algorithms is executed with a particular input, then in each branch of the search, the left-hand stack is modified by appending the same sequence of stacks to it, up to the end of branch. (Note that different branches can have different lengths.)

This is because the stacks in Algorithms 7.1 and 7.6 are only modified by appending stacks produced by refiners and splitters, and because decisions about the progression of the algorithm are made according to the size of the value of the isomorphism approximator. By the definition of a refiner as a pair of functions of one variable (Definition 4.1), the left-hand stack returned by a refiner depends only on the left-hand stack it is given; by Definition 6.1(iii), the left-hand stack defined by a splitter depends only on the given left-hand stack; and by Definition 5.1(iii), the size of the value of an isomorphism approximator is either zero, in which case the current branch immediately ends, or it depends only on the left-hand stack that is given.

Therefore we can store the new stacks that are appended to the left-hand stack as they are constructed, and simply recall them as they are needed later. This means that, on most occasions, when applying a refiner, we recall the value for the left stack, and compute only the value for the right-hand stack. This optimisation significantly improves the performance of the REFINE procedure.

8.1. Constructing and applying refiners via the fixed sequence of left-hand stacks

We saw in Lemmas 4.5 and 4.6 that any refiner for a non-empty set is derived from a function f from DigraphStacks(Ω) to itself satisfying $f(S^g) = f(S)^g$ for certain $g \in \mathrm{Sym}(\Omega)$. In this section, we give an example that demonstrates a difficulty in satisfying this condition, and a general method for giving refiners that overcome this difficulty. We use such refiners for groups and cosets in our experiments.

Example 8.1. Let $\Omega = \{1, \ldots, 6\}$ and $G = \langle (1\,2), (3\,4), (5\,6), (1\,3\,5)(2\,4\,6) \rangle$, and let Γ be the labelled digraph on Ω without arcs where Label(1) = *black*, Label(2) = *grey*, and the remaining vertices have label *white*. Finally, define $S = [\Gamma]$ and $T = [\Gamma^{(1\,3\,5)(2\,4\,6)}]$.

Suppose that we are searching for an intersection D of subsets of $\mathrm{Sym}(\Omega)$, one of which is G, and suppose that $\mathrm{Iso}(S, T)$ overestimates the solution. We wish to give a refiner (f, f) for G, where the function f takes into account that the elements of D respect the orbit structure of G.

Since $D \subseteq \mathrm{Iso}(S, T)$, elements of D induce isomorphisms from S to T. In particular, if Fixed is a fixed-point approximator (see Definition 5.2), then each element of D maps the list Fixed(S) to the list Fixed(T). For illustration, assume that Fixed(S) = $[1, 2]$ and Fixed(T) = $[3, 4]$, and let $G_{[1,2]} = \langle (3\,4)(5\,6) \rangle$ and $G_{[3,4]} = \langle (1\,2)(5\,6) \rangle$ denote the stabilisers of $[1, 2]$ and $[3, 4]$ in G, respectively. Therefore, since each element $x \in D$ maps $[1, 2]$ to $[3, 4]$ and is contained in G, it follows that $x \in G_{[1,2]} \cdot h$, where h is any permutation in G that maps $[1, 2]$ to $[3, 4]$, for instance $h := (1\,3\,5)(2\,4\,6) \in G$. This means that we can define $f(S)$ and $f(T)$ in terms of the orbits of $G_{[1,2]}$ and $G_{[3,4]}$.

One option is to define $f(S) = [\Gamma_{\mathcal{U}}]$ as in Example 4.12, for the set of orbits $\mathcal{U} := \{\{1\}, \{2\}, \{3, 4\}, \{5, 6\}\}$ of $G_{[1,2]}$ on Ω, and to define $f(T) = [\Gamma_{\mathcal{V}}]$ similarly for the set $\mathcal{V} := \{\{1, 2\}, \{3\}, \{4\}, \{5, 6\}\}$ of orbits of $G_{[3,4]}$ on Ω. This is valid, but not ideal, since a permutation could map $[\Gamma_{\mathcal{U}}]$ to $[\Gamma_{\mathcal{V}}]$ while mapping an orbit in $\mathcal{U}$ to any orbit of the same size in $\mathcal{V}$. However, every element of $G_{[1,2]} \cdot h$ maps the orbit $O \in \mathcal{U}$ to O^h. Therefore, this refiner does not eliminate some elements that, to us, are obviously not in D.

This is unsatisfactory, and so we would like to define $f(S) = f_{\mathcal{U}}(S)$ as in Example 4.10 for some *ordered* list $\mathcal{U}$ of the orbits of $G_{[1,2]}$. But then how should we order the orbits of $G_{[3,4]}$ in the corresponding way, to obtain a stack $f(T)$ such that $D \subseteq \mathrm{Iso}(f(S), f(T))$?

To address the problem discussed in Example 8.1, we use a technique similar to that of Leon [12]. In essence, we specify a refiner iteratively during search, when applying it to a new version of the left-hand stack. We can make choices as we do so (about the ordering of orbits, for example). Then, for all right-hand stacks that we encounter, we consult the choice made for the current left-hand stack, and remain consistent with that.

In more detail, we create such a refiner (f, f) for a subgroup G of $\mathrm{Sym}(\Omega)$ as follows. Let Fixed be a fixed point approximator, and initially let V_i and F_i be empty lists for all $i \in \mathbb{N}_0$. We describe how to apply (f, f) to stacks (S, T) of length $i := |S| = |T|$.

If V_i is still empty, then we redefine F_i to be $\textsc{Fixed}(S)$ and V_i to be a non-empty labelled digraph stack on Ω whose automorphism group contains G_{F_i}, the stabiliser of F_i in G. For example, V_i could be a list of orbital graphs of G_{F_i} on Ω, represented as labelled digraphs, or it could be the length-one stack $[\Gamma_{\mathcal{U}}]$ from Example 4.10, for some arbitrarily-ordered list $\mathcal{U}$ of the orbits of G_{F_i} on Ω.

If V_i is no longer empty, then we have already applied the refiner to stacks of length i. Since a search has at most one left-hand stack of any particular length, this means that we have already seen the left-hand stack S, and defined F_i and V_i in terms of it.

The refiner gives $f(S) = V_i$ and either $f(T) = V_i^a$ (if $\textsc{Fixed}(S)^a = \textsc{Fixed}(T)$, for some $a \in G$) or $f(T) = \textsc{EmptyStack}(\Omega)$. Note that, by Definition 5.2(ii), if no such element a exists, then S and T are not isomorphic via G, and so there are no solutions in the current branch. Therefore the algorithm should backtrack.

The mathematical foundation of this kind of refiner is given in Lemma 8.2. The notation in this lemma corresponds to the notation of the preceding paragraphs.

We may use this lemma with Lemma 4.6 to give refiners for cosets of subgroups.

Lemma 8.2. *Let* $G \leq \mathrm{Sym}(\Omega)$ *and let* $\textsc{Fixed}$ *be a fixed-point approximator. For all* $i \in \mathbb{N}_0$, *let* $V_i \in \textsc{DigraphStacks}(\Omega)$ *be a labelled digraph stack on* Ω, *and let* F_i *be a list of points in* Ω *whose stabiliser in* G *is a subgroup of* $\mathrm{Aut}(V_i)$.

We define a function f *from* $\textsc{DigraphStacks}(\Omega)$ *to itself as follows. For each* $S \in \textsc{DigraphStacks}(\Omega)$, *let*

$$f(S) = \begin{cases} (V_{|S|})^a & \text{if } {F_{|S|}}^a = \textsc{Fixed}(S) \text{ for some } a \in G, \\ \textsc{EmptyStack}(\Omega) & \text{if no such element } a \in G \text{ exists.} \end{cases}$$

Then (f, f) *is a refiner for* G.

Proof. Note that f is well-defined: if $S \in \textsc{DigraphStacks}(\Omega)$ and $a, b \in G$ both map $F_{|S|}$ to $\textsc{Fixed}(S)$, then ab^{-1} stabilises $F_{|S|}$, and so $ab^{-1} \in \mathrm{Aut}\big(V_{|S|}\big)$ by assumption; therefore $(V_{|S|})^a = (V_{|S|})^b$.

Let $S \in \textsc{DigraphStacks}(\Omega)$ and $g \in G$. By Lemma 4.5, it suffices to show that $f(S^g) = f(S)^g$. There exists an element $a \in G$ mapping $F_{|S|}$ to $\textsc{Fixed}(S)$ if and only if there exists an element of $b \in G$ mapping $F_{|S^g|} = F_{|S|}$ to $\textsc{Fixed}(S)$, since g maps $\textsc{Fixed}(S)$ to $\textsc{Fixed}(S^g)$ by Definition 5.2(ii). In that case that no such a and b exist, then $f(S^g) = \textsc{EmptyStack}(\Omega) = f(S)^g$. Otherwise $f(S) = (V_{|S|})^a$ and $f(S^g) = (V_{|S|})^b$. In this case, agb^{-1} fixed $F_{|S|}$ pointwise, and so $agb^{-1} \in \mathrm{Aut}\big(V_{|S|}\big)$ by assumption. Therefore

$$f(S)^g = (V_{|S|})^{ag} = (V_{|S|})^b = (V_{|S^g|})^b = f(S^g). \quad \square$$

9. Experiments

In this section, we provide experimental data comparing the behaviour of our algorithms against partition backtrack, in order to highlight the potential of our techniques.

We repeat the experiments of [8, Section 6], which demonstrated improvements from using orbital graphs, and we also investigate some additional challenging problems. In many cases we observe a significant advancement with our new techniques. We decided not to investigate classes of problems where partition backtrack already performs very well, or problems where we would expect all techniques (including ours) to perform badly. Instead we have chosen problems that are interesting and important in their own right, including ones that we expect to be hard for many search techniques.

At the time of writing, we have focused on the mathematical theory of our algorithms, but not on the speed of our implementations. Therefore, we only analyse the size of the search required by an algorithm to solve a problem, and not the time required. We define a *search node* of a search to be an instance of the main searching procedure being called recursively during its execution; the *size of a search* is then its number of search nodes. If an algorithm requires zero search nodes to solve a problem, then it solves the problem without entering recursion, which in our situation implies that the problem has either no solutions, or exactly one.

The size of a search should depend only on the mathematical foundation of the algorithm, rather than on the proficiency of the programmer who implements it, and so it allows a fair basis for comparisons. That said, we expect that where our algorithms require significantly smaller searches, then with an optimised implementation, the increased time spent at each node will be out-weighed by the smaller number of nodes in total, giving faster searches than partition backtrack. This is because, in general, a backtrack search algorithm spends time at each search node to prune the search tree and organise the search. The computations at each node of our algorithms are largely digraph-based, and the very high performance of digraph-based computer programs such as BLISS [10] and NAUTY [13] suggests that, in practice, these kinds of computations should be cheap.

For the problems that we investigate in Sections 9.1–9.3, we compare the following techniques:

(i) LEON: Standard partition backtrack search, as described by Jeffrey Leon [11,12].
(ii) ORBITAL: Partition backtrack search with orbital graph refiners, as in [8].
(iii) STRONG: Backtrack search with labelled digraphs, using the isomorphism and fixed-point approximators from Definition 5.9 and the splitter from Definition 6.4.
(iv) FULL: Backtrack search with labelled digraphs, using the isomorphism and fixed-point approximators from Definition 5.4 and the splitter from Definition 6.4.

The LEON technique is roughly the same as backtrack search with labelled digraphs, where the digraphs are not allowed to have arcs. The ORBITAL technique is roughly the same as backtrack search with labelled digraphs using the 'weak equitable approximation' isomorphism and fixed-point approximators from Definition 5.10.

The STRONG technique considers all labelled digraphs in the stack simultaneously to make its approximations, while the FULL technique computes isomorphisms and fixed

points exactly, rather than approximating them, and so in principle it is the most expensive of the four methods.

We require refiners for groups given by generators, for cosets of such groups, for set stabilisers, and for unordered partition stabilisers. We describe the refiners for set and unordered partition stabilisers in Section 9.1. For Leon we use the group and coset refiner described in [11]. For Orbital we use the **DeepOrbital** group and coset refiner from [8], although we get similar results for all the refiners described in that paper. For the Strong and Full techniques, we use refiners of the kind described in Section 8.1 for groups and cosets, using orbits and orbital graphs. These algorithms are similar to **DeepOrbital**, except they return the created digraphs instead of filtering them internally. For Strong and Full, we use the splitter given in Definition 6.4.

We performed our experiments using the GraphBacktracking [6] and BacktrackKit [7] packages for GAP [3]. BacktrackKit provides a simple implementation of the algorithms in [8,11,12], and provides a base for GraphBacktracking. We note that where we reproduce experiments from [8], we produce the same sized searches.

9.1. Set stabilisers and partition stabilisers in grid groups

We first explore the behaviour of the four techniques on stabiliser problems in grid groups. This setting was previously considered in [8, Section 6.1], and as mentioned there, these kinds of problems arise in numerous real-world situations.

Definition 9.1 *(Grid group [8, Definition 36]).* Let $n \in \mathbb{N}$ and $\Omega = \{1, \dots, n\}$. The direct product $\mathrm{Sym}(\Omega) \times \mathrm{Sym}(\Omega)$ acts faithfully on the Cartesian product $\Omega \times \Omega$ via $(\alpha, \beta)^{(g,h)} = (\alpha^g, \beta^h)$ for all $\alpha, \beta \in \Omega$ and $g, h \in \mathrm{Sym}(\Omega)$. The $n \times n$ *grid group* is the image of the embedding of $\mathrm{Sym}(\Omega) \times \mathrm{Sym}(\Omega)$ into $\mathrm{Sym}(\Omega \times \Omega)$ defined by this action.

Let $n \in \mathbb{N}$ and $\Omega = \{1, \dots, n\}$, and let $G \leq \mathrm{Sym}(\Omega \times \Omega)$ be the $n \times n$ grid group. If we consider $\Omega \times \Omega$ to be an $n \times n$ grid, where the sets of the form $\{(\alpha, \beta) : \beta \in \Omega\}$ and $\{(\beta, \alpha) : \beta \in \Omega\}$ for each $\alpha \in \Omega$ are the rows and columns of the grid, respectively, then G is the subgroup of $\mathrm{Sym}(\Omega \times \Omega)$ that preserves the set of rows and the set of columns.

We repeat the experiments in [8], and add an unordered partition stabiliser problem:

(i) Compute the stabiliser in G of a subset of $\Omega \times \Omega$ of size $\lfloor n^2/2 \rfloor$.
(ii) Compute the stabiliser in G of a subset of $\Omega \times \Omega$ with $\lfloor n/2 \rfloor$ entries in each grid-row.
(iii) If $2 \mid n$, then compute the stabiliser in G of an unordered partition of $\Omega \times \Omega$ that has two cells, each of size $n^2/2$.

As in [8, Section 6.1], we compute with the $n \times n$ grid group as a subgroup of $\mathcal{S}_{n^2}$, and the algorithms have no prior knowledge of the grid structure that the group preserves.

For Leon and Orbital, we refine for a set stabiliser as in [11]. For Strong and Full, our refiner for set stabiliser is the one from Example 4.10. The stabiliser in $\mathcal{S}_{n^2}$

Table 9.2
Search sizes for 50 instances of Problems (i) and (ii) in the $n \times n$ grid group.

n	Problem (i): LEON	Problem (i): ORBITAL, STRONG, FULL		Problem (ii): LEON	Problem (ii): ORBITAL, STRONG, FULL	
	Median	Median	Zero%	Median	Median	Zero%
3	4	2	22	7	2	0
4	8	0	50	8	2	0
5	16	2	44	13	2	0
6	23	0	68	34	2	20
7	34	0	74	41	0	54
8	46	0	90	92	0	68
9	58	0	92	108	0	54
10	75	0	88	290	0	86
11	107	0	94	262	0	90
12	124	0	100	1085	0	92
13	155	0	100	788	0	98
14	185	0	96	21774	0	96
15	216	0	98	2471	0	100

Table 9.3
Search sizes for 50 instances of Problem (iii) in the $n \times n$ grid group.

n	LEON	ORBITAL	STRONG, FULL	
	Median	Median	Median	Zero%
4	16	16	5	24
6	44	36	0	66
8	82	64	0	82
10	129	100	0	88
12	206	144	0	96
14	317	196	0	100
16	504	256	0	100
18	664	324	0	98

of an unordered partition with two parts of size $n^2/2$ is a subgroup isomorphic to the wreath product $\mathcal{S}_{n^2/2} \wr \mathcal{S}_2$. For each technique, for an unordered partition stabiliser, we directly use the group refiner for this subgroup.

Tables 9.2 and 9.3 show the results concerning the search size required to solve 50 random problems each of types (i), (ii), and (iii). An entry in the 'Zero%' column shows the percentage of problems that an algorithm solved with a search of size zero. These columns are omitted when they are all-zero.

In [8, Section 6.1], the ORBITAL algorithm was much faster than the classical LEON algorithm at solving problems of types (i) and (ii). In Table 9.2, we see why: ORBITAL typically requires no search for these problems. LEON used a total of 65,834 nodes to solve all problems in Problem (i), and 37,882,616 nodes for Problem (ii), while ORBITAL required 567 for Problem (i) and 1073 for Problem (ii). The same numbers of nodes were also required for both STRONG and FULL, since there is no possible improvement.

In Table 9.3 for Problem (iii), however, we clearly see the benefits of our new techniques. Partition backtrack – LEON and ORBITAL – takes an increasing number of search nodes, with 140,177 nodes required for LEON and 57,120 nodes for ORBITAL to solve all

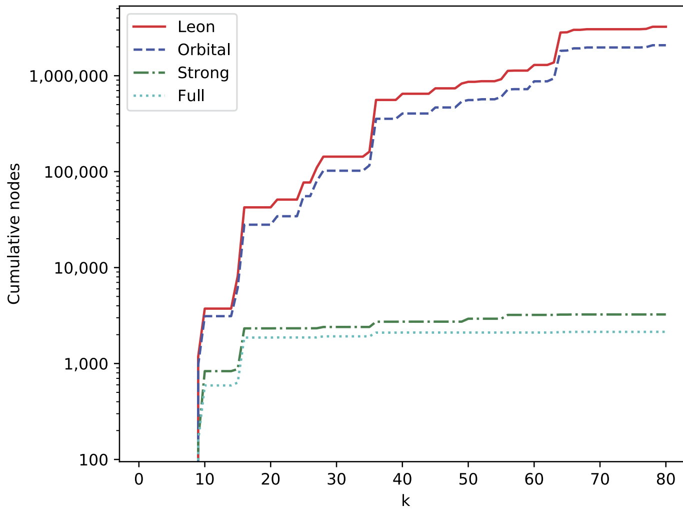

Fig. 9.4. Cumulative nodes required to intersect primitive (but not 2-transitive) groups with wreath products of symmetric groups, for all problems with $n \in \{6, \ldots, k\}$.

instances. But Strong is powerful enough in almost all cases to solve these same problems without search, requiring only 450 nodes to solve all problem instances.

9.2. Intersections of primitive groups with symmetric wreath products

Next, as in [8, Section 6.2], we consider intersections of primitive groups with wreath products of symmetric groups. To construct these problems, we use the primitive groups library, which is included in the PrimGrp [5] package for GAP.

For a given composite $n \in \{6, \ldots, 80\}$, we create the following problems: for each primitive subgroup $G \leq \mathcal{S}_n$ that is neither $\mathcal{S}_n$ nor the natural alternating subgroup of $\mathcal{S}_n$, and for each proper divisor d of n, we construct the wreath product $\mathcal{S}_{n/d} \wr \mathcal{S}_d$ as a subgroup of $\mathcal{S}_n$, which we then conjugate by a randomly chosen element of $\mathcal{S}_n$. Finally, we use each algorithm in turn to compute the intersection of G with the conjugated wreath product. We create 50 such intersection problems for each n, G, and d.

For each $k \in \{6, \ldots, 80\}$, we record the cumulative number of search nodes that each technique needs to solve all of the intersection problems for all composite $n \in \{6, \ldots, k\}$. We show these cumulative totals in Figs. 9.4 and 9.5, separating the groups that are 2-transitive from those that are primitive but not 2-transitive, as in [8, Section 6.2]. Note that the number of problems increases with the numbers of divisors of n and primitive groups of degree n; this explains the step-like structure in these figures.

For the primitive but not 2-transitive groups, the total number of search nodes required by the Leon algorithm is 3,239,403. The Orbital algorithm reduces this total search size to 2,079,356, and the cumulative search sizes for Strong (with 3,248 nodes) and Full (with 2,140 nodes) are even smaller.

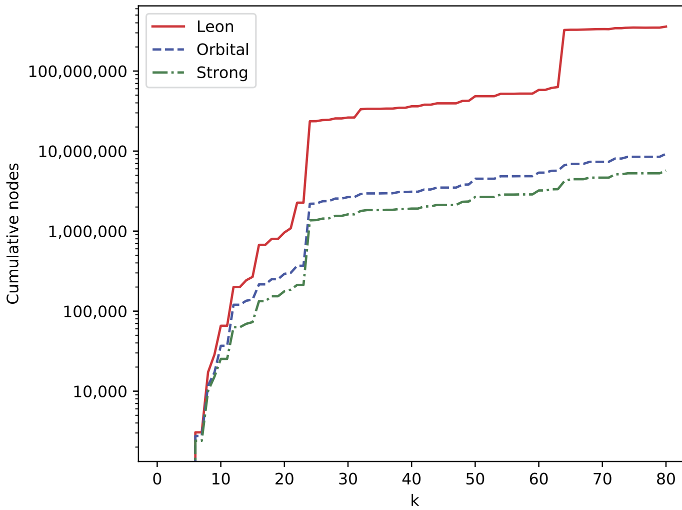

Fig. 9.5. Cumulative nodes required to intersect 2-transitive groups with wreath products of symmetric groups, for all problems with $n \in \{6, \ldots, k\}$. The line for FULL is omitted, since at this scale, it is indistinguishable from the line for STRONG.

This huge reduction happens because the STRONG and FULL algorithms solve almost every problem without search. Out of 40,150 experiments, the STRONG algorithm required search for only 703, and the FULL algorithm required search for just 654. On the other hand, the LEON and ORBITAL algorithms required search for every problem.

For the intersection problems involving groups that are at least 2-transitive, the improvement of the new techniques over the partition backtrack algorithms is much smaller, and all of the algorithms required a non-zero search size to solve every problem. This was to be expected: a 2-transitive group has a unique orbital graph, which is a complete digraph.

9.3. Intersections of cosets of intransitive groups

In this section, we go beyond the experiments of [8, Section 6], with a problem that we expect to be difficult for all search techniques: intersecting cosets of intransitive groups that have identical orbits, and where all orbits have the same size.

More precisely, we intersect right cosets of subdirect products of transitive groups of equal degree. Given $k, n \in \mathbb{N}$, we randomly choose k transitive subgroups of $\mathcal{S}_n$ from the transitive groups library TRANSGRP [4], each of which we conjugate by a random element of $\mathcal{S}_n$, and we create their direct product, G, which we regard as a subgroup of $\mathcal{S}_{kn}$. Then, we randomly sample elements of G until the subgroup that they generate is a subdirect product of G. If this subdirect product is equal to G, then we abandon the process and start again. Otherwise, the result is a generating set for what we call a *proper* (k, n)*-subdirect product.*

Table 9.6
Search sizes for (k, n)-subdirect product coset intersection problems, where for each n, we ran 50 experiments for each $k \in \{2, \ldots, 10\}$.

n	Leon		Orbital			Strong		
	Mean	Median	Mean	Median	Zero%	Mean	Median	Zero%
2	3	2	2	2	14	2	2	14
3	1418	7	19	0	58	19	0	59
4	1250	12	71	0	69	62	0	70
5	37924	30	15576	10	14	8803	0	54
6	584	12	254	6	36	139	0	86
7	53612	28	43555	14	0	8982	0	70
8	1142	8	997	8	15	4	0	98
9	6547	9	5562	9	2	7	0	95
10	8350	10	6959	10	1	7	0	97

Table 9.7
Search sizes for 50 (k, n)-subdirect product coset intersection problems.

k	n	Leon		Orbital			Strong		
		Mean	Median	Mean	Median	Zero%	Mean	Median	Zero%
4	5	13683	30	6356	11	8	6176	5	40
4	6	376	18	335	6	8	87	0	76
4	7	8612	49	7065	43	0	6494	0	54
4	8	1133	8	365	8	14	0	0	100
4	9	1947	9	621	9	0	0	0	96
4	10	458	10	410	10	2	0	0	98
8	5	119561	130	42885	30	17	36888	0	58
8	6	70	12	25	0	56	67	0	98
8	7	19731	49	11154	43	0	167	0	86
8	8	209	8	58	8	12	0	0	100
8	9	152	9	144	9	2	0	0	100
8	10	138	10	64	10	2	0	0	100

In our experiments, for various $k, n \in \mathbb{N}$, we explore the search space required to determine whether or not the intersections of pairs of right cosets of different (k, n)-subdirect products are empty. To make the problems as hard as possible, we choose coset representatives that preserve the orbit structure of the (k, n)-subdirect product.

We performed 50 random instances for each pair (k, n), for all $k, n \in \{2, \ldots, 10\}$, and we show a representative sample of this data in Tables 9.6 and 9.7 and Fig. 9.8. Table 9.6 shows results for each n for all k combined, and Table 9.7 gives a more in-depth view for two values of k. The tables omit data for the Full algorithm, because it was mostly identical to the data for the Strong algorithm.

The Strong algorithm solved a large proportion of problems with zero search. As n and k increase, we find that Strong is also able to solve almost all problems without search, and the remaining problems with very little search. The only problems where Strong does not perform significantly better are those involving orbits of size 2 ($n = 2$). This is not surprising as there are very few possible orbital graphs for such groups. We note that the problems with $n = 5$ and 7 seem particularly difficult. This is because transitive groups of prime degree are primitive, and sometimes even 2-transitive, in which case they do not have useful orbital graphs.

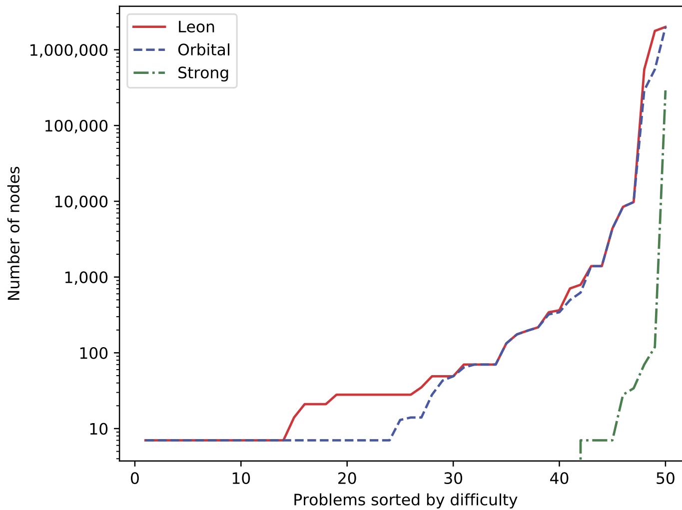

Fig. 9.8. Search sizes for 50 (7,7)-subdirect product coset intersection problem instances. The data for Full is almost identical to the data for Strong, and is omitted.

On the other hand, Orbital solved a lot fewer problems without search, and Leon solved none in this way. Although the relatively low medians show that all of the algorithms performed quite small searches for many of the problems, we see a much starker difference in the mean search sizes. These means are typically dominated by a few problems; see Fig. 9.8.

To give a more complete picture of how the algorithms perform, Fig. 9.8 shows the search sizes for all 50 intersections problem that we considered for $n = k = 7$, sorted by difficulty. The data that we collected in this case was fairly typical. Fig. 9.8 shows that Strong solves almost all problems with very little or no search, and it only requires more than 50 search nodes for the three hardest problems. On the other hand, Leon and Orbital need more than 50 nodes for the 18 hardest problems. All algorithms found around 30% of the (randomly generated) problems easy to solve.

10. Conclusions and directions for further work

We have discussed a new search technique for a large range of group and coset problems in $\mathrm{Sym}(\Omega)$, building on the partition backtrack framework of Leon [11,12], but using stacks of labelled digraphs instead of ordered partitions.

Our new algorithms often reduce problems that previously involved searches of hundreds of thousands of nodes into problems that require no search, and can instead be solved by applying strong equitable approximation to a pair of stacks. There already exists a significant body of work on efficiently implementing equitable partitioning and automorphism finding on digraphs [10,13], which we believe can be generalised to work incrementally with labelled digraph stacks that grow in length. We did not yet concern ourselves with the time complexity or speed of our algorithms in this paper, nor did we

discuss their implementation details. However, we do intend for our algorithms to be practical, and we expect that with sufficient further development into their implementations, our algorithms should perform competitively against, and even beat, partition backtrack for many classes of problems. This requires optimising the implementation of the algorithms that we have presented here. In future work, we plan to show how the algorithms described in this paper can be implemented efficiently, and compare the speed of various methods for hard search problems. In particular, we aim for a better understanding of when partition backtrack is already the best method available, and when it is worth using our methods. Further, earlier work which used orbital graphs [8] showed that there are often significant practical benefits to using only some of the possible orbital graphs in a problem, rather than all of them. We will investigate whether a similar effect occurs in our methods.

Another direction of research is the development and analysis of new types of refiners, along with an extension of our methods. For example, we have seen some refiners in our examples that perfectly capture all the information about the set that we search for, and it is worth investigating this more. See [9, Section 5.1] for first steps in this direction. We could also allow more substantial changes to the digraphs, such as adding new vertices outside of Ω. One obvious major area not addressed in this paper is normaliser and group conjugacy problems. These problems, as well as a concept for the quality of refiners are addressed in ongoing work that builds on the present paper.

While the step from ordered partitions to labelled digraphs already adds some difficulty, we still think that it is worth considering even more intricate structures. Why not generalise our ideas to stacks of more general combinatorial structures defined on a set Ω? The definitions of a splitter, of an isomorphism approximator, and of a refiner were essentially independent of the notion of a labelled digraph, and so they – and therefore the algorithms – could work for more general objects around which a search method could be organised.

References

[1] Jan De Beule, Julius Jonušas, James D. Mitchell, Michael Torpey, Maria Tsalakou, Wilf A. Wilson, Digraphs – GAP package, Version 1.4.1, https://digraphs.github.io/Digraphs, 2021.

[2] John D. Dixon, Brian Mortimer, Permutation Groups, Springer-Verlag, New York, 1996.

[3] The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.11.1, 2021.

[4] Alexander Hulpke, TransGrp – GAP package, Version 3.3, http://www.math.colostate.edu/~hulpke/transgrp, 2021.

[5] Alexander Hulpke, Colva M. Roney-Dougal, Christopher Russell, PrimGrp – GAP package, Version 3.4.1, https://gap-packages.github.io/primgrp, 2020.

[6] Christopher Jefferson, Wilf A. Wilson, GraphBacktracking – GAP package, Version 0.4.0, https://github.com/peal/GraphBacktracking, 2021.

[7] Christopher Jefferson, Markus Pfeiffer, Wilf A. Wilson, BacktrackKit – GAP package, Version 0.4.1, https://github.com/peal/BacktrackKit, 2021.

[8] Christopher Jefferson, Markus Pfeiffer, Rebecca Waldecker, New refiners for permutation group search, J. Symb. Comput. 92 (2019) 70–92.

[9] Christopher Jefferson, Markus Pfeiffer, Rebecca Waldecker, Wilf A. Wilson, Permutation group algorithms based on directed graphs (extended version), arXiv:1911.04783.

[10] Tommi Junttila, Petteri Kaski, Engineering an efficient canonical labeling tool for large and sparse graphs, in: 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2007, pp. 135–149.
[11] Jeffrey S. Leon, Partitions, refinements, and permutation group computation, in: Groups and Computation, II, New Brunswick, NJ, 1995, in: DIMACS Ser. Discrete Math. Theoret. Comput. Sci., vol. 28, 1997, pp. 123–158.
[12] Jeffrey S. Leon, Permutation group algorithms based on partitions. I. Theory and algorithms, J. Symb. Comput. 12 (1991) 533–583.
[13] Brendan D. McKay, Adolfo Piperno, Practical graph isomorphism, II, J. Symb. Comput. 60 (2014) 94–112.
[14] Heiko Theißen, Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen, Ph.D. thesis, Lehrstuhl D für Mathematik, RWTH Aachen, 1997.