Journal of Automated Reasoning (2026) 70:11
https://doi.org/10.1007/s10817-026-09759-8

Probabilistic Automaton Classifier Applied to Examples Related to the Andrews-Curtis Conjecture

Michael Fairbank[1] · **Alexei Lisitsa**[2] · **Alexei Vernitski**[1]

Received: 18 July 2025 / Accepted: 12 June 2026

Abstract
The unsolved Andrews-Curtis conjecture in group theory gives rise to many famously challenging theorem-proving problems. Automated reasoning is one of the best state-of-the-art approaches to solving them. We have developed a new classifier inspired by group theory and a new search algorithm orientated at using for the Andrews-Curtis conjecture. We combine them with automated reasoning and compare their performance with automated reasoning.

Keywords Automated theorem proving · Supervised learning · Group theory · Andrews–Curtis conjecture · Finite automaton

1 Introduction

This paper applies automated reasoning to experimental mathematics; namely, it focuses on applying automated theorem proving to a long-standing open problem in combinatorial group theory, the Andrews–Curtis conjecture (ACC) [2]. Sects. 1, 2 provide the context of this work in the study of ACC; then our approach is described in detail in Sects. 3, 4, 4.2, and experiments are presented in Sects. 5, 6, 7.

Broadly speaking (for precise details, see the following section), ACC asserts that any balanced presentation of the trivial group can be simplified (or AC-simplified), that is, transformed into the trivial presentation by applying a finite sequence of elementary transformations. The conjecture is widely believed to be false, but no counterexamples have been found yet. Many presentations remain unsimplified despite considerable effort, and some of them are considered potential counterexamples.

Despite its simple formulation, the problem is highly nontrivial. Whereas verifying that a given presentation satisfies the conjecture is, in theory, decidable via exhaustive application of all transformation sequences, in practice such brute-force enumeration is infeasible: the

✉ Alexei Lisitsa
a.lisitsa@liverpool.ac.uk

✉ Alexei Vernitski
asvern@essex.ac.uk

Michael Fairbank
m.fairbank@essex.ac.uk

[1] University of Essex, Colchester, United Kingdom

[2] University of Liverpool, Liverpool, United Kingdom

Published online: 08 July 2026

search space grows exponentially with the length of presentation, and known lower bounds on simplification lengths are superexponential [5, 15].

Over the past three decades, a range of computational strategies have been developed to explore potential counterexamples to the ACC. Early heuristic methods based on genetic algorithms [23] managed to simplify some presentations which were previously considered difficult. These techniques were refined and extended in subsequent work [14, 33], simplifying broader families of presentations.

At the same time, a more exhaustive approach based on breadth-first search through the tree of AC-equivalent presentations was developed in [9]. This enabled systematic classification of presentations up to bounded length and showed, for example, that each balanced presentation of the trivial group with two generators and relator length up to 13 is AC-equivalent either to the trivial presentation or to the presentation

$$\langle a, b \mid a^3 b^{-4}, abab^{-1}a^{-1}b^{-1} \rangle,$$

known as AK_3, which remains one of the most prominent potential counterexamples.

Subsequent work addressed practical implementation of these strategies at scale. Disk-based hash tables and parallel computing were employed in [21] to improve tractability, while alternative approaches based on coset enumeration were explored in [8]. Other developments include transformation rules inspired by the conjugacy search problem [26] and methods based on discrete Morse theory [7], combining topological and algorithmic techniques.

Reinforcement learning has also emerged as a promising direction. In a large-scale empirical investigation, Shehper et al. [32] applied learning-guided search to explore transformation sequences for presentations of bounded size, leading to numerous new simplifications and demonstrating the potential of adaptive strategies.

A complementary line of work is based on automated reasoning. A logical approach to AC-simplification based on encoding AC-transformations as first-order axioms was introduced in [16, 18, 19]. This method reduces simplification to a theorem-proving task and allows the use of general-purpose automated theorem provers. It has proved highly effective in practice, discovering long and nontrivial simplification sequences — in one case exceeding 8000 transformation steps [20]. Moreover, in all known instances where simplifications have been obtained by other computational methods, corresponding simplifications have also been found using this logical approach, while additional new simplifications have been discovered.

Despite this progress, important challenges remain. The performance of automated reasoning is highly sensitive to the logical encoding and to the internal strategies of the prover, and it is often unclear whether failure to find a simplification reflects an inherent mathematical obstruction or limitations of the method. Furthermore, some simplifications may require extremely long proofs: for example, [20] reports a case where a simplification was found only after approximately 74 days of computation.

In addition, several infinite families of presentations are known whose simplifiability remains open, including the Akbulut–Kirby family AK_n [3], the Miller–Schupp family $MS_n(w)$ [11], and the Gordon family $G_{n,m,p,q}$ [6]. These examples illustrate the persistent difficulty of the conjecture and motivate the development of new computational approaches.

Against this background, it is natural to explore whether alternative paradigms can complement existing methods and help navigate the complex search space of ACC simplification.

In this work we propose and investigate a novel approach based on machine learning, classification, and search. In contrast to approaches that integrate machine learning directly into the internal proof search of automated theorem provers (e.g. overviewed in [4]), our method applies classification externally to guide the selection of equivalent starting presentations. This approach can be used both independently and in combination with automated reasoning.

Our experiments indicate that the search algorithm itself can be effective in finding simplifications, while the benefit of combining the two approaches remains inconclusive: in some cases, combining them provides a clear advantage, while in others automated reasoning alone is already highly effective. This behaviour reflects the complexity of the ACC landscape and suggests that different computational techniques may be better suited to different classes of instances.

2 Theoretical Introduction to ACC

This section introduces the Andrews–Curtis conjecture and outlines how it can be studied using automated reasoning. We first present the group-theoretic formulation of the problem, and then give a high-level overview of its translation into a logical framework used throughout the paper.

2.1 The Andrews-Curtis Conjecture

Within this subsection we assume that the reader is familiar with basic concepts of group theory, including group presentation. A classical introduction in combinatorial group theory can be found in [24]. A group presentation $\langle x_1, \ldots, x_n; r_1, \ldots, r_m \rangle$ is said to be *balanced* if the number of generators $x_1, \ldots, x_n$ equals the number of relators $R_1, \ldots, r_m$, i.e., $n = m$. A balanced group presentation is said to be *trivial* if all its relators are exactly generators: $\langle x_1, \ldots, x_n; x_1, \ldots, x_n \rangle$.

The Andrews–Curtis framework uses the following transformations (called AC-transformations) of balanced presentations.

(AC1) Replace a relator r_i by its inverse r_i^{-1}.
(AC2) Replace r_i by $r_i r_j$, for $j \neq i$.
(AC3) Replace r_i by its conjugate $w r_i w^{-1}$, where w is any word over the generators.
(AC4) Introduce or delete a generator y together with a relator y.

Two presentations are said to be Andrews-Curtis equivalent or AC-equivalent if one can be obtained from the other using a sequence of transformations of types AC1–AC3. If transformation AC4 is also allowed, the presentations are said to be stably AC-equivalent. Given a balanced presentation p, the simplification (stable simplification) is a sequence of transformations showing that p is AC-equivalent (stably AC-equivalent) to a trivial balanced presentation.

Conjecture 1 (Andrews-Curtis [2]) *If $\langle x_1, \ldots, x_n; r_1, \ldots r_n \rangle$ is a balanced presentation of the trivial group then it can be simplified, that is, it is AC-equivalent to the trivial presentation $\langle x_1, \ldots, x_n; x_1, \ldots x_n \rangle$.*

A weaker form of the conjecture states that every balanced presentation of the trivial group can be stably simplified (i.e. transformations AC4 are allowed). Both variants of the conjecture remain open and challenging problems. In what follows, we focus on the strong form of the conjecture and its particular case of dimension 2, that is, the number of generators (and, equivalently, the number of relators) is 2.

It is useful to note that other types of AC-transformations apart from AC1–AC4 also have been considered. For example, one can change the order in which the relators are listed in the presentation; it is straightforward to show that performing such a change does not affect

AC-equivalence as this transformation can be obtained as a composition of applications of AC1–AC3. Another example is a cyclic permutation transformation, as follows.

(CAC) Replace a relator r_i with a cyclic permutation $\tilde{r_i}$ of r_i

In [12] it is shown that the cyclic version of Andrew-Curtis conjecture, in which AC3 is replaced with CAC transformation, is equivalent to the original one. Our random search algorithm used for producing datasets uses the CAC transformation, as we will describe in more detail in 6.1.

2.2 Automated Reasoning for ACC Simplification

A powerful approach to AC-simplification based on automated reasoning in first-order logic has been proposed and developed in [16, 17, 19]. In this approach, AC-transformations are translated into logical axioms, combining group-theoretical rewriting with first-order inference, so that simplification can be formulated as a proof search problem for an automated theorem prover.

Several logical encodings of this idea have been explored, including implicational and equational formulations, each in ground and non-ground variants. No systematic advantage of any one of these variants has been established; rather, their effectiveness appears to depend substantially on the specific problem instance and prover behaviour. In the present paper we focus on non-ground formulations and consider two representative encodings: the implicational non-ground encoding (IN) and the equational non-ground encoding (EN), which we describe in turn.

2.2.1 Implicational Non-Ground Encoding IN

Denote by I_{ACT} the first-order theory that includes the following axioms.

(G1)	$(x * y) * z = x * (y * z)$
(G2)	$x * x' = e \ \& \ x' * x = e$
(G3)	$x * e = x \ \& \ e * x = x$
(I-R1L)	$R(x, y) \rightarrow R(x', y)$
(I-R2L)	$R(x, y) \rightarrow R(x * y, y)$
(I-R2R)	$R(x, y) \rightarrow R(x, y * x)$
(I-R3LZ)	$R(x, y) \rightarrow R((z * x) \cdot z', y)$

We briefly clarify the structure of the theory I_{ACT}. The axioms labeled G1, G2, and G3 represent the standard group axioms, associativity, identity, and the inverse, expressed equationally using the function symbols $*$ (group multiplication), e (identity element), and postfix notation x' for inverses. The additional axioms, I-R1L, I-R2L, I-R2R, and I-R3Z, formalize a version of AC-transformations for balanced presentations with 2 generators and 2 relators. The list of generators is the same in all these presentations (to be specific, in our experiments the generators are denoted by a, b, as in examples of presentations in Sect. 1).

This shortened list of transformations omits certain transformations (inversion and conjugation of the second relator), but those transformations can be obtained as compositions of the transformations which are included in the list [16, 19].

To represent transformation sequences within this logical method, we introduce the binary predicate $R(t_1, t_2)$, where t_1 and t_2 are ground terms denoting relators in a presentation. The

intended interpretation of $R(t_1, t_2)$ being true is that the presentation $\langle a, b; t_1, t_2\rangle$ is AC-equivalent to a specified initial presentation. This interpretation is formalized in the following proposition [16, 19, 20].

Proposition 1 *For ground terms t_1, t_2, t_3, t_4 in group theory built of constants e (group unit), a and b (group generators), we have*

$$I_{ACT} \vdash R(t_1, t_2) \rightarrow R(t_3, t_4)$$

if and only if presentations $\langle a, b; t_1, t_2\rangle$ and $\langle a, b; t_3, t_4\rangle$ are AC-equivalent.

2.2.2 Equational Non-Ground Encoding (EN

Denote by E_{ACT} an equational theory $T_G \cup ACT^=$ where T_G includes groups axioms G1, G2, G3 and $ACT^=$ includes the following axioms:

E-R1L $f(x, y) = f(x', y)$
E-R1R $f(x, y) = f(x, y')$
E-R2L $f(x, y) = f(x \cdot y, y)$
E-R2R $f(x, y) = f(x, y \cdot x)$
E-R3L$_i$ $f(x, y) = f((z \cdot x) \cdot z', y)$

We have now an analogue of Proposition 1 for equational encoding [16, 19]:

Proposition 2 *For ground terms t_1, t_2, t_3, t_4 in group theory built of constants e (group unit), a and b (group generators), we have*

$$E_{ACT} \vdash f(t_1, t_2) = f(t_3, t_4)$$

if and only if presentations $\langle a, b; t_1, t_2\rangle$ and $\langle a, b; t_3, t_4\rangle$ are AC-equivalent.

Propositions 1 and 2 allow us to reduce any question concerning the AC-equivalence of specific balanced group presentations of dimension 2 to a derivability problem in first-order logic over the theories I_{ACT}, and E_{ACT}, respectively . These derivability problems are then handed over to automated theorem provers, making use of the capabilities of modern automated deduction systems.

2.2.3 Experimental Settings

For experiments reported in this paper we have used Prover9 theorem prover [22] as a part of ProverX platform [30] running on single Intel(R) Core(TM) i7-6700 CPU @ 3.400GHz 64GB RAM, Rocky Linux 8.10 (green Obsidian), kernel: Linux 4.18.0-533.8.1.el8_10.x86_64.

The choice of Prover9 for these experiments is motivated by prior experience with ACC-related and similar mathematical reasoning tasks, where in our experiments it has proved particularly useful in practice. Although we have also experimented with other first-order provers, including E and Vampire, we did not obtain comparable results on problems of this type. A systematic comparative evaluation of prover performance, however, lies outside the scope of the present work.

Prover9 was run with fixed configurations for each group of experiments reported in this paper. The relevant limits and search parameters are indicated in the description of the corresponding experiments. For reproducibility, the exact Prover9 configurations used for all reported runs are provided in Appendix.

Python code for the algorithms introduced in Sects. 4 and 6 was run on Intel(R) Core(TM) i5-11500 @ 2.70GHz, 16.0 GB RAM, Windows 11 23H2.

3 Overview of our Learning-Guided Search Approach

In this section we outline the main idea of the learning-guided search approach proposed in this paper. The approach combines three components: a search algorithm browsing the space of presentations, a classifier trained to distinguish regions of this space, and, optionally, automated reasoning used to construct simplifications. The key intuition is to guide the search for simplifications by identifying intermediate presentations that are simultaneously close to both the initial and target presentations. The approach can be applied both independently of automated reasoning and in combination with it. We first describe the underlying graph-theoretic perspective and then present the heuristic algorithm based on classification and search.

Many search tasks can be usefully visualized as searching for a path in an infinite graph; this includes the simplification problem described in Sect. 2. Consider an infinite graph Γ in which each vertex is a presentation, and two vertices are connected with an edge if and only if they can be transformed to one another by a single application of one of AC-transformations. Then simplification is a path from the vertex corresponding to the original presentation to the vertex corresponding to the trivial presentation. If we ask, for instance, if the presentation AK_3 can be simplified, the question can be reformulated as finding a path in the graph Γ from a vertex that is the presentation AK_3 to the vertex that is the trivial presentation.

In this graph-theoretical language, the new approach we introduce in this paper is as follows. Consider two vertices p, q in Γ. Suppose we want to find a path from p to q in Γ. One approach is to use the search algorithm; we shall denote this approach by $\mathcal{S}$. Another approach is to use automated reasoning; we shall denote this approach by $\mathcal{AR}$.

Or, to find a path from p to q, we can use the following heuristic approach.

1. By applying random sequences of AC-transformations to p, produce a family of vertices P. Within the graph Γ, vertices in P can be thought of a being close to p (with the word 'close' understood informally as likely to be reachable either by random search or automated reasoning). Likewise, by applying random sequences of AC-transformations to q, produce a family of vertices Q. Within the graph Γ, vertices in Q can be thought of as being close to q.
2. A classifier is an algorithm which can be trained to categorize data into classes. Using the classes P and Q above, train a classifier to distinguish between vertices belonging to P and vertices belonging to Q.
3. Suppose a vertex r lies in P, but the classifier misclassifies r as belonging to Q; this might be a mere mistake, but also this might be interpreted as an indication that r is close to both p and q.
   (a) Then we can begin search again starting from misclassified presentations and repeat steps 1, 2 above several times, until a path from p to q is found. We shall denote this approach, which uses the search and the classifier iteratively, by $\mathcal{S} \leftrightarrows \mathcal{C}$.
   (b) Or, we can expand the approach $\mathcal{S} \leftrightarrows \mathcal{C}$ in (a) above by passing its results (after several iterations) to automated reasoning and using automated reasoning to finish finding a path from p to q. We shall denote this approach, which uses the search and the classifier iteratively and then automated reasoning, by $\mathcal{S} \leftrightarrows \mathcal{C} \rightarrow \mathcal{AR}$.

Speaking of applying automated reasoning to simplification in ACC, it is useful to note that previous experiments show that automated reasoning can find a way to transform one presentation into another more successfully when the latter transformation is the trivial transformation, see e.g. [17]. Therefore, in most of our experiments, when we use AR to find a

Fig. 1 An example of transitions corresponding to letters x and y

path between two AC-equivalent presentations, one of which is the trivial presentation, we start from the non-trivial presentation and transform it to the trivial presentation.

4 Automaton-Based Classifier and Training

4.1 Automaton Classifiers

In mathematics (more specifically, in group theory and semigroup theory) an old and standard way of classifying words into classes is using finite automata. Note that this classification is done by hand-picking the automata, without using machine learning; typical applications of this approach are presented in [25]. Now we will describe how a finite automaton works. Choose a positive integer s; numbers in the range $0, 1, \ldots, s-1$ will be called states. Accordingly, when we consider matrices of size $s \times s$ or rows of length s, positions in such matrices or rows will be indexed by numbers in the range $0, 1, \ldots, s-1$.

For each i is in the range $0, 1, \ldots, s-1$, by $\vec{e}_i$ we will denote a row of length s in which the entry at the position i is 1 and all others are 0. In particular, $\vec{e}_0$ will be used repeatedly, so let us stress that $\vec{e}_0$ is $(1, 0, \ldots, 0)$. The reader familiar with one-hot encoding can notice that each $\vec{e}_i$ is one-hot encoding of i; therefore, we will refer to rows $\vec{e}_i$ as one-hot vectors.

For each letter x, consider an $s \times s$ matrix T_x, which we will call a *transition matrix*, such that each row of T_x is a one-hot vector. It is traditional to represent T_x by a directed graph whose nodes are the states, and there is an edge (marked with x) from state i to state j if and only if the entry at the position i, j is 1. For example, suppose $T_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $T_y = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$. The corresponding directed graphs are shown in Fig. 1. It is convenient to think of a transition matrix as describing, literally, a transition from one state to another produced by a letter.

Once we have fixed the transition matrices for each letter, for each word we can define the state corresponding to this word; here is how it is done. Fix a word w and denote the i-th letter in w by $w(i)$. Define $\mathcal{T}$ to be the list of all transition matrices, and define

$$f(w, \mathcal{T}) := \vec{e}_0 \prod_i T_{w(i)} \tag{1}$$

Recall that $\vec{e}_0$ has size $1 \times s$ and each $T_{w(i)}$ has size $s \times s$; hence, $\vec{e}_0 \prod_i T_{w(i)}$ has size $1 \times s$, that is, it is a row of length s. It is useful to notice that $f(w, \mathcal{T})$ is not just any row, but a one-hot row vector $\vec{e}_j$.

Instead of matrix notation $\vec{e}_0 \prod_i T_{w(i)}$ used above, if we use graph-theoretical language, j in the previous sentence is the state at which we end up if we start from state 0 and transition to other states as indicated by the directed graphs corresponding to letters, proceeding through all letters of the word one at a time, from the beginning of the word to its end.

To turn a finite automaton into a classifier, we add a final step which maps the one-hot vector outputs produced by (1) into classification categories. Suppose there are c classes $0, 1, \ldots, c-1$. Introduce a vector $\vec{c}$ of length s, in which each entry is in the range $0, 1, \ldots, c-$

1. Define the classification of a word w as:

$$f'(w, \mathcal{T}, \vec{c}) := f(w, \mathcal{T}) \cdot \vec{c} = \left(\vec{e}_0 \prod_i T_{w(i)} \right) \cdot \vec{c} \quad (2)$$

where $\cdot$ denotes the usual inner product. Now f' is the final definition of our finite-automaton classifier.

In simpler terms, the i-th component of the vector $\vec{c}$ indicates which class should be chosen when the i-th entry of $f(w, \mathcal{T})$ is 1. Note that since $f(w, \mathcal{T})$ is a one-hot vector, this results in a unique classification of the input word w as belonging to one of the classes.

To make the classifier work correctly, it needs training, i.e. choosing the appropriate entries for the $\mathcal{T}$ matrices and for the $\vec{c}$ vector. This is discussed in the next section. To train an automaton, we will generalize automata in the following way.

The definition so far has assumed that the transition matrices $\mathcal{T}$ consisted of one-hot rows; we will refer to such transition matrices as *classical* transition matrices. We can extend the definition of transition matrices to allow for non-negative real-valued entries. In these *probabilistic* transition matrices, each row in T_x is not necessarily a one-hot vector, but any probability distribution, that is, any row consisting of non-negative numbers such that their sum is 1 (some readers might be familiar with this kind of matrices under a name of right stochastic matrices). This generalized kind of automata is known as *probabilistic automata* [27].

In Sect. 4.2 we describe an efficient algorithm for training classifiers using probabilistic automata. Classifiers using finite automata are more difficult to train. Therefore, one can be tempted to ask whether it might be better to work only with probabilistic automata classifiers and not with finite automata classifiers. We believe that the advantage of a finite automaton classifier in the context of our research is that its transition matrices have a very simple structure and are, therefore, inspectable; this is useful for double-checking results or when looking for mathematical insights. This is why in our experiments we train probabilistic automata classifiers and after they are trained, we convert them to finite automata classifiers as described in Subsection 4.2.3.

4.2 Training an Automaton

In this section we describe how to train the classifier defined in equation (2). Recall that gradient descent, in machine learning, is a process of iteratively improving parameters in search of an optimal solution. Instead of conducting a brute-force search through the space of possible values for $\mathcal{T}$ and $\vec{c}$, we define a closed-form solution for $\vec{c}$ in terms of $\mathcal{T}$, and then define an objective function on $\mathcal{T}$ on which to do gradient descent, with respect to $\mathcal{T}$. In order to allow gradient descent, which is naturally a continuous-valued process, we use probabilistic transition matrices instead of classical transition matrices, and then convert probabilistic transition matrices to classical transition matrices after training is complete.

4.2.1 Objective Function for Training the Classifier

Recall that we are working in the supervised learning scenario. Consider a training dataset $\mathcal{D}$ consisting of words w_i and their corresponding labels y_i; specifically, $\mathcal{D} =$

$\{(w_0, y_0), (w_1, y_1), \ldots, (w_{N-1}, y_{N-1})\}$. As discussed in Sect. 3, for our purposes it is sufficient to consider 2 classes; following notation used in Sect. 4, we will denote them by 0 and 1. Let $\mathcal{D}_0 = \{w_i : y_i = 0\}$, i.e. the set of words with label 0, and $\mathcal{D}_1 = \{w_i : y_i = 1\}$ be the set of words with label 1.

To obtain a closed-form expression for $\vec{c}$, we first note that the job of $\vec{c}$ is to convert the one-hot output vector of f in equation (1) into a single classification in the range 0, 1. We want to choose $\vec{c}$ that performs best on the training set $\mathcal{D}$, given the current set of tentative values of $\mathcal{T}$. Hence we choose each component of $\vec{c}$ so that it maximises the number of correct classifications given by (2), when counted over the whole training set. This solution is given by:

$$\vec{c} := argmax_{col} \begin{pmatrix} \sum_{w \in \mathcal{D}_0} f(w, \mathcal{T}) \\ \sum_{w \in \mathcal{D}_1} f(w, \mathcal{T}) \end{pmatrix} \tag{3}$$

where $argmax_{col}$ is applied to each column of the $2 \times s$ matrix given in (3). This equation ensures that $\vec{c}$ is a vector of length s, with all of its elements either 0 or 1.

When the definition of $\vec{c}$ given in (3) is used in (2), we can see that the number of correct classifications made by the classifier over the whole dataset will be given by

$$\text{Correct count} = max_{col} \begin{pmatrix} \sum_{w \in \mathcal{D}_0} f(w, \mathcal{T}) \\ \sum_{w \in \mathcal{D}_1} f(w, \mathcal{T}) \end{pmatrix} \cdot \vec{1}, \tag{4}$$

where $\vec{1} := (1, 1, \ldots, 1)$ is a vector of length s comprised of all 1s. (This inner product with $\vec{1}$ is simply designed to sum over all components of the vector produced by the max operator.)

Likewise, we can see that the number of incorrect classifications over the whole dataset is given by

$$\mathcal{L}(\mathcal{D}, \mathcal{T}) := min_{col} \begin{pmatrix} \sum_{w \in \mathcal{D}_0} f(w, \mathcal{T}) \\ \sum_{w \in \mathcal{D}_1} f(w, \mathcal{T}) \end{pmatrix} \cdot \vec{1}. \tag{5}$$

As this is the number of incorrect classifications, which we would ideally like to be zero, this makes a good objective function to minimise during training. We refer to $\mathcal{L}(\mathcal{D}, \mathcal{T})$ as the *loss function*, which we seek to minimize. Note that the loss function does not depend on $\vec{c}$. So it is just necessary to minimize (5) with respect to $\mathcal{T}$, i.e. independently of $\vec{c}$; and then afterwards one can calculate $\vec{c}$ via its definition in ((3)).

4.2.2 Training Procedure by Gradient Descent

To train the classifier function $f(w, \mathcal{T})$, we would like to use gradient descent on $\mathcal{L}(\mathcal{D}, \mathcal{T})$ with respect to the $\mathcal{T}$ matrices; however this aim is complicated by the fact that the classical transition matrices $\mathcal{T}$ consist of one-hot rows and cannot have arbitrary entries. Hence we relax this constraint so that we will now consider probabilistic transition matrices. To achieve this, we replace each transition matrix $T \in \mathcal{T}$ by an entirely unconstrained (but identically shaped) floating-point valued matrix T'. (These T' matrices can include both positive and negative values; and their rows do not necessarily sum to 1.)

In order to recover valid probabilistic matrices, i.e. with non-negative rows that sum to one, we apply a softmax function to each row of each transition matrix, and thus do gradient descent on $\mathcal{L}(D, softmax(\mathcal{T}'))$. The softmax function is applied to each row of each T' matrix. We define the softmax function for an arbitrary row vector $\vec{a}$ with components a^i,

by:

$$(softmax(\vec{a}))^i := \frac{\exp(a^i)}{\sum_k \exp(a^k)}, \qquad (\text{for } i = 0, 1, \ldots, s-1) \tag{6}$$

Initially, we start with completely random values for the $\mathcal{T}'$ matrices. To perform each step of gradient descent, we apply the update equation:

$$\Delta\mathcal{T}' = -\eta \frac{\partial \mathcal{L}(\mathcal{D}, softmax(\mathcal{T}'))}{\partial \mathcal{T}'} \tag{7}$$

where $\eta > 0$ is a small learning rate. To compute the gradients in the right-hand side of (7) efficiently, we can use the back-propagation algorithm [29, 35], or equivalently automatic differentiation packages provided by a neural-network software library [1, 28, 37].

It is necessary to apply multiple iterations of (7) to form the full training process. It is also possible to switch the simple fixed-size gradient descent step given by (7) into something more elaborate, which speeds up and slows down more dynamically, in a high-dimensional loss-function surface. For this we use the Adam optimiser [13] in our experiments, but also many other possibilities are available from the machine-learning literature.

We acknowledge that there could be other methods to search through the parameter space than gradient descent on unconstrained $\mathcal{T}'$ parameters. For example, we could search directly on the constrained stochastic parameters $\mathcal{T}$ if we used instead a constrained optimisation search method, or genetic algorithms, or similar. However we find that the unconstrained gradient descent method works well enough for our purposes in this paper and leave those alternative search possibilities to future research.

4.2.3 Converting to Finite Automata After Training

Whereas it is perfectly possible to use probabilistic automata for classification, using formula (3), in our research we prefer a simpler construction of finite automata, for the reasons explained in the end of Sect. 4.

Gradient descent described in Subsection 4.2.2 gradually improves the value of the loss function and accuracy of classification performed by a probabilistic automaton. As we observed in all our experiments, this process simultaneously also improves accuracy of classification performed by a finite automaton whose transition matrices are formed of one-hot rows approximating the distributions in the rows of the probabilistic transition matrices; see details below.

After training the $\mathcal{T}'$ matrices to optimise the function $\mathcal{L}(D, softmax(\mathcal{T}'))$, even after going through the softmax function, the values of $softmax(T')$ will generally not be just 0s and 1s. We can convert probabilistic transition matrices to their maximum-likelihood discrete-valued equivalent matrices by finding the largest entry in each row, making it 1, and making all other entries 0, by:

$$T \leftarrow \text{onehot}(\text{argmax}_{row}(T')), \tag{8}$$

where argmax_{row} is applied to each row of each T' matrix; and onehot converts an integer back into a one-hot encoded vector of length s.

After applying this update, the $\mathcal{T}$ matrices will be comprised entirely of 0s or 1s, and their rows will sum to one.

We then set the $\vec{c}$ vector as defined by (3). This gives us our final trained classifier, $f'(w, \mathcal{T}, \vec{c})$, given by (2).

Given the training set $\mathcal{D}$, or even an unseen test set $\mathcal{D}'$, we can define the accuracy as the fraction of correct classifications (i.e. the fraction of classifications $f'(w_i, \mathcal{T}, \vec{c})$ that are equal to their true label y_i).

4.2.4 Comparison with Recurrent Neural Networks

The central dynamical equation of the automaton, given by (1), is very closely related to the dynamics of a recurrent neural network (RNN) [31, 36]. In a RNN, at each discrete time step t, we have a row-vector state, s_t, and a row-vector input, x_t. At each time step, the RNN's state vector evolves according to:

$$s_{t+1} = g(s_t \cdot W1 + x_t \cdot W2 + b), \tag{9}$$

where $\cdot$ denotes matrix multiplication, g is a non-linear "activation" function (e.g. tanh) and $W1$, $W2$ are appropriately-dimensioned learnable weight matrices, and b is the neural network's learnable bias vector.

For illustration of how similar this is to the automaton, we could re-write the automaton equation (1) as:

$$s_{t+1} = s_t \cdot T_{w(t)}, \tag{10}$$

with $s_0 = \vec{e}_0$ and $x_t = w(t)$.

Key differences between the automaton classifier and the RNN equation are that there is no activation function g or bias vector required in the automaton; This is possible partly because the transition matrices in the automaton are stochastic, i.e. their rows sum to 1. So the state vector's magnitude cannot grow indefinitely through each multiplication of a new transition matrix in (10). Also, in the automaton, non-linearity at each time step is provided by choosing which transition matrix to use at each time step, i.e. via the subscript of $w(t-1)$ in (10). This choice of transition matrix acts as an "if statement", which provides a form of non-linearity, and allows the automaton to be flexible and powerful like a RNN. Having non-linearity is a key component of neural networks which allows them to be universal function approximators [34].

Another key difference between a RNN and the automaton is that the input vectors arrive as different x_t vectors at each time step in (9), but in the automaton, the inputs arrive as different letters $w(t)$ at each time step, and these affect the choice of transition matrices at each time step t, in (10).

We trained a simple RNN, and also a long-short term memory (LSTM) RNN [10], on the S_3 group-theory classification task described in Sect. 5.1. Results are shown in Fig. 2. Despite the similarity between the automaton and recurrent neural networks, the RNN experimental results were much worse than those of our classifier. In each RNN experiment, the RNN/LSTM layer consisted of 25 hidden nodes with a tanh activation function, followed by a fully connected layer with one sigmoid output node. Training used the Adam optimiser with leaning rate 0.01, and cross-entropy loss function [13]. We attempted other learning rates (0.001, 0.1), and also different context sizes (64 and 25), but found they did not make any improvement for the RNN performances.

A hypothesis for why the RNN experiments did not perform as well as the automaton classifier is that the automaton can switch the entire weight matrix being used in the recurrent layer; dependent on a single input letter x_t at each time step t (by (10)). In contrast, with a RNN, changing the value of a single one-hot encoded input letter x_t merely generates a rank-1 adjustment to the output of the matrix product $x_t.W2$ in (9); so this does not generate as much flexibility as achieved by (10).

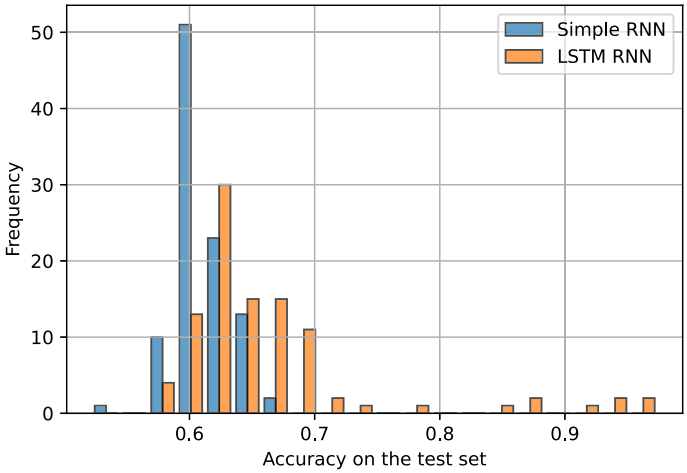


Fig. 2 Performance of a Simple RNN and a LSTM RNN on the S_3 group-theory classification task. Results over 100 trials using each classifier type

5 Our Classifier Learns Algebra

Before applying the proposed approach to ACC simplification tasks, we examine the behavior of the classifier on simpler and controlled examples. The aim of this section is to assess whether the classifier is able to learn group-theoretical facts and capture meaningful structural distinctions in the space of presentations. These experiments serve as a preliminary validation of the classifier component and help to understand its strengths.

5.1 The Word Problem of Group S_3

Using an automaton classifier (instead of an off-the-shelf classifier) has conceptual advantages. As we have said, the automaton produced by training has a clear structure and few parameters. Hence, it can be inspected, for example, to look for mathematical insights.

In addition to this, one can conjecture that since finite automata originate from group theory (and its generalization semigroup theory), an automaton classifier might be expected to perform well in applications to group theory. We have conducted several small-scale experiments applying the automaton classifier to group theory; we will describe one. A group known as S_3 has a presentation $\langle a, b | aa, bb, ababab \rangle$. We considered words over the alphabet a, b of length up to 20, and trained classifiers to recognize the words that are equal to the identity element of S_3. To produce the histogram below, we repeated the experiment 100 times. In each experiment, we produced a dataset consisting of two classes. For one class, we randomly chose 500 words equal to the identity element. For the other class, we randomly chose 500 words not equal to the identity element. Then we split each of the two classes equally (into 250 and 250 words) to form a training set and a test set[1]. We compared the performance of our classifier (with 25 states, trained for 500 iterations) with that of decision trees and gradient boosting trees (as implemented in the sklearn library). Note that decision

[1] Data related to experiments described in this paper can be found at https://doi.org/10.5281/zenodo.16088111. As to this experiment, one of the datasets is included in the data provided with this paper, as an example.

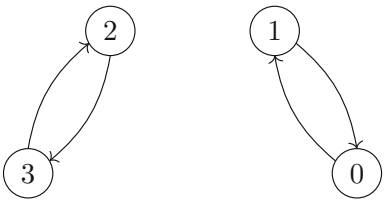


Fig. 3 Transitions of letter a

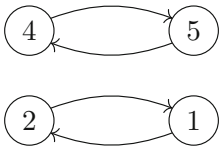
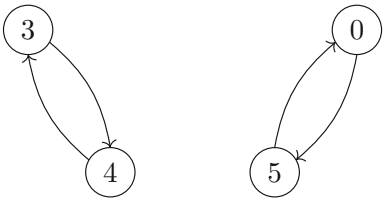


Fig. 4 Transitions of letter b



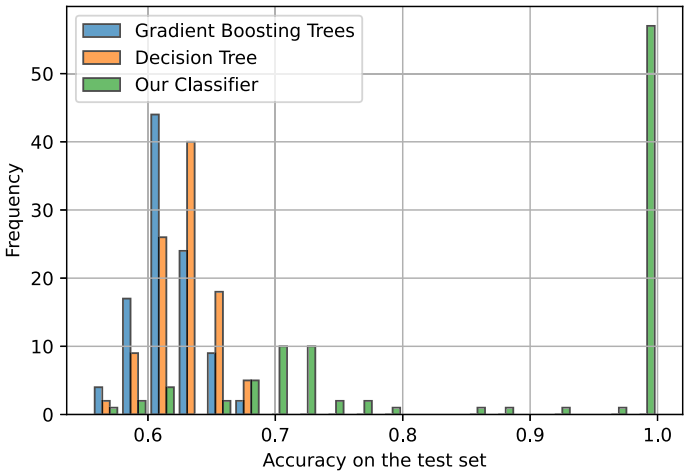


Fig. 5 Gradient boosting trees, decision trees, and our classifier learning on group-theory data. Results over 100 trials using each classifier type

trees are a type of finite automata, so one could expect that their performance would be relatively close to that of an automaton classifier.

A histogram of the accuracy of the three classifier types on test sets is presented in Fig. 5. As you can see, on average, the decision trees and gradient boosting trees have $60\% - 65\%$ accuracy on the test sets. Our classifier had higher accuracy on test sets. Frequently, our

classifier's accuracy on test sets was 100%; this means that our training process successfully managed to construct an automaton that included a version of the automaton shown in Figs. 3, 4 (Figs. 3, 4 present the smallest automaton that can solve this classification problem correctly). In cases when our classifier failed to achieve 100% accuracy on test sets, it was, on average, about 70%.

To attempt to improve the performance of the decision (and gradient-boosting) tree classifiers, we varied the tree depths, learning rate, and number of trees used in the ensembles. However none of these variations yielded any noticeable improvement. We assume these classifiers do not perform well on this task because the task is a time-sequence classification task, comprised of up to 20 time steps. In this situation, decision-tree classifiers treat each of those 20 input time steps as independent features; whereas our classifier treats all time-step transitions identically. This means that once our classifier learns the physics of how the system gets from one given time step to the next, it will automatically apply the same physics at all later time steps. In contrast, the decision-tree based classifiers must learn the given rule independently for all 20 time steps. For this reason, it might be better to compare our classifier against classifiers more suited to time-sequences– which we do in the next subsection.

5.2 Learning the Difference Between Group Presentations

The first question to ask is whether the classifier is just brute memorizing the dataset or is actually learning something mathematically significant about the problem. The answer is that if we split the dataset into a training set and a test set, there is good learning both on the training set and on the test set. We have conducted such experiments repeatedly; here are some specific examples.

Recall that in each of our experiments we consider two presentations p and q. By using AC-transformations at random starting from p, we produce a family of presentations P. Likewise, by using AC-transformations at random starting from q, we produce a family of presentations Q. To make the datasets more homogeneous, in these experiments we only used presentations whose total length is within a reasonable range, namely, between 10 and 30 letters (this measure has ensured, most importantly, that the classifier is not tempted to classify shorter presentations as produced from the trivial presentation).

As to the choice of the original presentations p and q, we will now concentrate on 3 examples. (We use these examples because they are the same examples that we use in more complicated experiments in the following subsections, so it is good to get familiar with them.)

- The trivial presentation with relators a, b; we will refer to the family of presentations produced from the trivial presentation as $\langle \text{Tr} \rangle$.
- The presentation with relators $a^{-1}b^3ab^{-4}$, $a^{-1}b^{-1}ab^{-3}a^{-1}b^{-1}$ introduced in Theorem 10 in [32]; we will refer to this presentation as Th10-1 and the family of presentations produced from this presentation as $\langle \text{Th10-1} \rangle$.
- The presentation with relators $a^{-1}b^4ab^{-5}$, $a^{-1}bab^{-1}a^{-1}b^{-3}$, also introduced in Theorem 10 in [32]; we will refer to this presentation as Th10-2 and to the family of presentations produced from this presentation as $\langle \text{Th10-2} \rangle$.

When one inspects presentations in $\langle \text{Tr} \rangle$, $\langle \text{Th10-1} \rangle$ and $\langle \text{Th10-2} \rangle$ visually, it is impossible to spot any obvious differences between these families of presentations, they look similar[2].

For these experiments, the size of each family of presentations is 500, and each family of presentations is split at random into 250 presentations in the training set and 250 in the

[2] Classes $\langle \text{Tr} \rangle$, $\langle \text{Th10-1} \rangle$, $\langle \text{Th10-2} \rangle$ are included in the data provided with this paper.

test set. The classifier using a 50-state automaton is trained for 1000 iterations. The alphabet of the automaton of the classifier consists of 5 letters, standing for a, a^{-1}, b, b^{-1} and the separator symbol used to separate the two words in a presentation.

When we train the classifier to distinguish between $\langle \text{Tr} \rangle$ and $\langle \text{Th10-1} \rangle$, accuracy on the training set is about 89%, and accuracy on the test set is about 81%.

When we train the classifier to distinguish between $\langle \text{Tr} \rangle$ and $\langle \text{Th10-2} \rangle$, accuracy on the training set is about 91%, and accuracy on the test set is about 83%.

When we train the classifier to distinguish between $\langle \text{Th10-1} \rangle$ and $\langle \text{Th10-2} \rangle$, accuracy on the training set is about 89%, and accuracy on the test set is about 83%.

The fact that the classifier demonstrates good learning both on the training set and the testing set demonstrates that the classifier is not overfitting on the training set but learns something relevant about the two classes in the dataset.

6 A Search Algorithm and Experiments With It

We now introduce the search procedure used in the proposed approach and apply it to simplification tasks in the Andrews–Curtis setting. The procedure explores the space of presentations by generating candidates through sequences of AC-transformations and attempts to identify simplification paths through this exploration. We first describe the algorithm and then evaluate its ability to find simplifications on representative instances.

6.1 Search Algorithm

This subsection describes the search algorithm that we used to produce multiple presentations at random, starting from a given presentation (such as $\langle \text{Tr} \rangle$, $\langle \text{Th10-1} \rangle$ and $\langle \text{Th10-2} \rangle$ used in Subsection 5.2). We designed it to be used as step 1 of the multi-step approach introduced in Sect. 3; to remind you, this search algorithm is needed to produce a training set for the classifier to work on. As noted in Subsection 2.1, although the transformations of words used by our search algorithm might look different from those in Subsection 2.1, they are equivalent to them.

Our search algorithm performs a random search and is not designed to simplify presentations, but in our experiments, frequently it is unexpectedly successful at finding the trivial presentation. These results are described in the following subsections of this section.

Below are the steps of our random search algorithm. In this description we refer to group presentations as pairs of words in the group alphabet.

A1 We begin from a pair of words u, v which is chosen at random from a list including the initial presentation and, if this algorithm has already successfully produced some other presentations, those presentations, too.

A2 Repeat the following steps for the chosen number of iterations (in our experiments, 500, but this number is of little significance due to another, more frequently used termination condition below).

A3 split the word u into two words $u = u_1u_2$ at a random position and produce a word $u' = u_2u_1$.

A4 Split the word v into two words $v = v_1v_2$ at a random position and produce a word $v' = v_2v_1$.

A5 Produce the word v'' at random as either v' or $(v')^{-1}$.

A6 Produce the word w as $v''u'$.

A7 Produce the word w' by reducing the word w, that is, we delete all consecutive occurrences of a letter and its inverse, including at the ends of w.
A8 Choose the new pair of words at random as either w', u or w', v.
A9 If the total length of the new pair is longer than the maximal chosen length of an intermediate pair (in our experiments, usually 55 letters), stop. This is done to prevent having to work with unwieldy long presentations.

We only use presentations in our dataset whose length is locally minimal. This is done to prevent having presentations in the dataset which are too close to one other, and to encourage finding presentations which are sufficiently far from one another. Thus, after a pair has been produced by the steps above, the following exhaustive search is performed. To keep notation simple, we assume that these steps start from a pair of words u, v.

B1 Split the word u into two words $u = u_1u_2$ in all possible ways, producing a word $u' = u_2u_1$.
B2 Split the word v into two words $v = v_1v_2$ in all possible ways, producing a word $v' = v_2v_1$.
B3 Produce the word v'' in both ways, as v' and $(v')^{-1}$.
B4 Produce the word w as $v''u'$.
B5 Produce the word w' by reducing the word w.
B6 Choose the new pair of words in both ways, as w', u and w', v.
B7 If none of the pairs produced is shorter than the pair u, v, keep the pair u, v; otherwise, as soon as a shorter pair has been found, repeat these steps beginning from that shorter pair.

After a pair whose length is locally minimal has been found by the steps above, we finalise the selection process by the following steps.

C1 The two words in the presentation are swapped with one another, and the cyclic shifts and the group inversion are applied to the two words in all possible combinations, and among these presentations, the one which is smallest in the lexicographical order is selected. This is done to prevent having presentations in the dataset which are essentially the same and only differ in notation.
C2 If the presentation is longer than the chosen maximal length (in our experiments, 30 letters), it is discarded. This is done to prevent having to work with unwieldy long presentations. Also, together with the next condition, this prevents the classifier learning to classify by length, instead of learning more pertinent mathematical features.
C3 If the presentation is shorter than the chosen minimal length (in our experiments, 10 letters), it is discarded.
C4 After that, if the presentation has not been produced before, it is added to the dataset.

The steps above are repeated until a desired number of presentations has been found (in our experiments, usually 50).

6.2 Simplifying Presentations Th10-1 and Th10-2

Recall presentations Th10-1 and Th10-2 used in Subsection 5.2. It was recently proved (by computer, using approach known as reinforcement learning) [32] that presentation Th10-2 can be simplified.

In our experiments, the search algorithm described above simplifies both Th10-1 and Th10-2 within approximately one minute.

For comparison, automated reasoning requires substantially longer runtimes on these instances. Using the implicational non-ground (IN) encoding, simplifications were obtained in approximately 29,000s and 380,000s, respectively.[3] The equational non-ground (EN) encoding reduces these times to approximately 11,400s and 35,000s, respectively; however, these runtimes remain significantly higher than those achieved by the proposed search-based method.

The two proofs produced by our search algorithm are so short that we present their summaries below.

First, below is a summary of how Th10-1 is simplified.

- Begin with $ababbbAb, abbbbABBB$. (Note that step C1 is applied before we begin, to have all presentations in a canonical form.)
- After 6 moves (steps A2–A9) it becomes $aabbbAb, aaBAbbbb$
- After 5 moves it becomes $aaBabbbbAAb, aaBAbbbb$
- After 8 moves it becomes $aaaaBAAbAAB, aaBAbbbb$
- After 5 moves it becomes $aaaBAAbAAB, aaBAbababab$
- After 4 moves it becomes $aaaaBBAb, abbABBBBAbb$
- After 5 moves it becomes $aaaabbaBBAB, aaaaBBAb$
- After 6 moves it becomes $aaaabbaBBAB, ababbAbbaBBBBAB$
- After 3 moves it becomes $aabbAbbaBBBBAB, abaBBABaBaBaB$
- After 4 moves it becomes $aaaaBBAb, aaabaBB$
- After 5 moves it becomes $aaabaBB, abABBAb$
- After 5 moves it becomes $aaabaBAB, aaBAbaB$
- After 5 moves it becomes $aaabaBAAB, abAbAB$
- After 6 moves it becomes $aaBABBB, abbAB$
- After 6 moves it becomes a, b

Second, below is a summary of how Th10-2 is simplified.

- Begin with $abABabbb, abbbbbABBBB$.
- After 4 moves it becomes $ababAB, abbbbbABBBB$
- After 4 moves it becomes $aabAB, aaBBBBAbbb$
- After 5 moves it becomes $aaaaBBBAbb, aabAB$
- After 5 moves it becomes $aaaaaaaBBab, aabAB$
- After 5 moves it becomes $aabAB, aaBaBaBabab$
- After 4 moves it becomes $aaaBBBabb, aabAB$
- After 6 moves it becomes $aabAB, aaBaBBabb$
- After 5 moves it becomes $aaBabbbab, ababAB$
- After 6 moves it becomes $aaabbbbb, aaaBAAbaBABBAAb$
- After 4 moves it becomes $aaababAAABaabbabABAAB, aaabbbbb$
- After 3 moves it becomes $aaabbAAABaabAbbABAB, aabAbAbAbAb$
- After 4 moves it becomes $aaaaaBBB, aaBBAbbbaBaBBBAbbAbaB$
- After 5 moves it becomes $aaaaaBBB, aaBBAbbbabbABBBaBAbaB$
- After 5 moves it becomes $aaaaaBBB, aaBBabbbAbbABBBaBAbaB$
- After 5 moves it becomes $aabaBabbbAbABBBaBAb, abababababBB$
- After 4 moves it becomes $aaaaaBBB, aaBBabbbABBAbaB$
- After 4 moves it becomes $aaaaaBBB, aaBBabbabABBBabbAbABBAbaB$

[3] These results were obtained using configuration C1; both IN and EN experiments reported here use this configuration (see Appendix).

- After 4 moves it becomes $aaaaaBBB, aaBBabbabABBBabbAbAbABBaB$
- After 4 moves it becomes $aaaaaBBB, aaBBaBabbbbABBBabbAbAbABBaB$
- After 2 moves it becomes $aaaBBBAAbABAAbb, abababababBB$
- After 4 moves it becomes $aaaaaBBB, aaBBaBabbbAbAbABBaB$
- After 5 moves it becomes $aaaaaBBB, aaBBabbbAbABBaB$
- After 4 moves it becomes $aabaBabbbAABB, abababababBB$
- After 4 moves it becomes $abaBAb, abbbbaBaB$
- After 5 moves it becomes $aaabbbbb, abABBAB$
- After 5 moves it becomes $aabAABaBBBB, abAbbAB$
- After 5 moves it becomes $aabbaBBaB, aaBAb$
- After 8 moves it becomes $aaabAbbaB, abaBAb$
- After 5 moves it becomes $aaaabbAB, aabaBAb$
- After 5 moves it becomes $aaababbAB, abaBAb$
- After 5 moves it becomes $aabbaBBB, aaBAb$
- After 6 moves it becomes $aaaabaBB, aaBAb$
- After 5 moves it becomes $ababbbAb, abABB$
- After 7 moves it becomes $aabbbAb, abABB$
- After 5 moves it becomes $aabbAABaB, abABB$
- After 4 moves it becomes $ababbAb, abABB$
- After 7 moves it becomes $aabbAb, abABB$
- After 10 moves it becomes a, b

7 Comparing $\mathcal{AR}$ and $\mathcal{S} \leftrightarrows \mathcal{C}$ and $\mathcal{S} \leftrightarrows \mathcal{C} \rightarrow \mathcal{AR}$

In this section we investigate the interaction between the search-based method, the classifier, and automated reasoning across several classes of ACC instances. The aim is to understand how the effectiveness of these approaches varies depending on the problem and the logical encoding. We present three representative experiments: instances derived from Th10-1, the family $MS_n(w_*)$, and instances from the datasets in [26].

7.1 $\mathcal{AR}$ Applied to ⟨Th10-1⟩ and Compared with $\mathcal{C}$

Recall from Subsection 5.2 that ⟨Th10-1⟩ is a large family of presentations produced by our search algorithm $\mathcal{S}$ from presentation Th10-1. In [32], it was established (using reinforcement learning) that Th10-1 can be simplified. With automated reasoning, simplifications of Th10-1 and presentations in ⟨Th10-1⟩ typically found within several hours. Our search algorithm $\mathcal{S}$ simplifies Th10-1 within seconds; in other words, the trivial presentation is contained in ⟨Th10-1⟩. The purpose of the current experiment is, on the one hand, to apply $\mathcal{AR}$ to the non-trivial presentations in ⟨Th10-1⟩ and, on the other hand, apply the classifier $\mathcal{C}$ to the same presentations. We compare how quickly $\mathcal{AR}$ simplifies these presentations with how likely these presentations are to be misclassified by $\mathcal{C}$.

We consider the family of 500 presentations ⟨Th10-1⟩ produced from Th10-1 by our search algorithm (excluding one presentation which is the trivial presentation). For each presentation, we recorded:

Table 1 Results for Th10-1, implicational non-ground encoding (IN)

	classified incorrectly	classified correctly
simplified within 30m	14	22
not simplified within 30 m	39	424

Table 2 Results for Th10-1, equational non-ground encoding (EN)

	classified incorrectly	classified correctly
simplified within 30m	27	262
not simplified within 30 m	26	164

- whether it can be simplified by automated reasoning within 30 minutes,[4]
- whether it is classified correctly as belonging to $\langle$Th10-1$\rangle$ or misclassified as belonging to $\langle$Tr$\rangle$.

If classifier-guided search is useful for supporting automated reasoning, one would expect that presentations that simplify quickly are more likely to be misclassified. On this assumption, those presentations that are misclassified by $\mathcal{C}$ are good candidates to pass to $\mathcal{AR}$ to attempt to simplify.

The results for the implicational non-ground encoding (IN)[5] are shown in Table 1. As you can see, indeed, incorrectly classified presentations are 36% likely to be simplified within 30 minutes, compared with correctly classified presentations, which are only 5% likely to be simplified within 30 minutes.

We repeated the same experiment using the equational non-ground encoding (EN in automated reasoning. The results are shown in Table 2. As you can see, in this experiment, being misclassified by $\mathcal{C}$ is not a good predictor for being simplified quickly by $\mathcal{AR}$.

As discussed earlier, all presentations considered here are equivalent variants of a presentation known to admit a simplification. For this class of instances, the EN encoding appears to be substantially more effective for automated reasoning than the IN encoding. Consequently, while Table 1 shows a clear concentration of successful simplifications among misclassified presentations, Table 2 exhibits a much more uniform distribution.

Thus, the benefit of combining classifier-guided search with automated reasoning seems to be a mixed picture. For the IN encoding, where automated reasoning alone encounters greater difficulty, selecting promising starting points via the classifier significantly improves performance. In contrast, for the EN encoding, where automated reasoning is relatively efficient, the additional guidance from the classifier does not provide a noticeable advantage. This illustrates that the interaction between search-based methods and automated reasoning depends on the encoding and the structure of the instance, reflecting the complexity of the ACC landscape.

7.2 The Family of Presentations $MS_n(w_*)$

We now consider the family of presentations $MS_n(w_*)$, which has been studied extensively in the literature. The Miller–Schupp family of balanced presentations of the trivial group was

[4] Proofs and outputs are included in the data accompanying this paper.

[5] for all experimnents reported in this section we used configutaion C2 of Prover9 (see Appendix).

introduced in [11] and is defined by

$$MS_n(w) = \langle a, b \mid a^{-1}b^n ab^{-(n+1)},\ a^{-1}w \rangle,$$

where $n \geq 1$ and w is a word with exponent sum 0 on a. A particular subfamily $MS_n(w_*)$, obtained by fixing $w_* = b^{-1}aba^{-1}$ for $n \geq 1$, has been investigated in a number of works [8, 19, 23, 32]. The approach based on reinforcement learning and then human generalization of experimental results allowed authors of [32] to prove that $MS_n(w_*)$ presentations are AC-simplifiable for all $n \geq 2$

Automated reasoning methods have proved highly effective on this family: using both generic and specialized encodings, presentations for $n = 2, \ldots, 9$ have been successfully simplified, see [19, 20].

We evaluated the proposed search-based method on this family in order to assess its effectiveness both as a standalone approach $\mathcal{S} \leftrightarrows \mathcal{C}$ and in combination with automated reasoning $\mathcal{S} \leftrightarrows \mathcal{C} \rightarrow \mathcal{AR}$.

As a standalone method, $\mathcal{S} \leftrightarrows \mathcal{C}$ is able to simplify instances up to $n = 4$ within a short time (within 5 minutes). For comparison, automated reasoning solves these instances almost instantaneously (within seconds). Thus, while the search-based approach is effective on small instances, it is less efficient than automated reasoning on these examples.

For larger instances, the search-based approach $\mathcal{S} \leftrightarrows \mathcal{C}$ is unable to produce simplifications. For $n = 5$, the approach $\mathcal{S} \leftrightarrows \mathcal{C}$ was able to demonstrate (within 5 minutes) that this presentation is AC-equivalent to that corresponding to $n = 4$, and the latter can be successfully simplified by $\mathcal{S} \leftrightarrows \mathcal{C}$.

We also investigated whether the search-based approach and automated reasoning can assist one another in the combined approach $\mathcal{S} \leftrightarrows \mathcal{C} \rightarrow \mathcal{AR}$. In the family of presentations $MS_n(w_*)$, this combined approach was not beneficial. For $n = 6$, we identified an equivalent presentation that led to a marginal improvement in proof search time (less than 3% in our experiments, with proof times on the order of 10^4 seconds). For other tested instances ($n = 5$ and $n = 6$), no consistent improvement was observed [6].

To summarise, these results indicate that on the family of presentations $MS_n(w_*)$, automated reasoning is very effective, whereas $\mathcal{S} \leftrightarrows \mathcal{C}$ is much less so, and combining these approaches $\mathcal{S} \leftrightarrows \mathcal{C} \rightarrow \mathcal{AR}$ does not seem to produce any benefits.

7.3 Presentations from [26]

We now consider instances from [26], which provide a wide range of test cases for evaluating different approaches to AC-simplification.

Instances known to be simplifiable. The instances listed in Table 4 of [26] are known to admit Andrews–Curtis simplification. For all 12 such instances, both automated reasoning (see [16]) and the proposed search-based method are able to find simplifications quickly (within seconds in our experiments). This confirms that both approaches are effective on relatively accessible cases.

Open instances. Table 2 of [26] lists instances for which simplification remains an open problem. In our experiments, neither automated reasoning nor the search-based method was able to resolve these cases, however much time we spent on this task. This is consistent with

[6] EN encoding and C2 configuration for Prover9 were used for experiments reported in this section (see Appendix).

the current state of knowledge and indicates that these instances remain challenging for both approaches.

Automorphic reductions. The most interesting cases are those listed in Table 3 of [26]. Table 3 in [26] gives 16 examples of presentations which nobody has managed to simplify yet. However, it is reported in [26] that each of the presentations in Table 3 satisfies a weaker property, namely, it is AC-equivalent to the presentation produced from it by replacing every a by b and vice versa, although explicit reductions were not provided. (Slightly more generally, for these instances, it was reported in [26] that they are AC-equivalent to their automorphic images, meaning automorphisms of F_2.)

Previous attempts to obtain such AC-equivalences using automated reasoning were unsuccessful (see [17]).

When we used $\mathcal{S} \leftrightarrows \mathcal{C}$, that is, our search algorithm and classifier together, applying them in turns, we were partially successful; namely, for presentations 1, 2, 7, 8, 12, 16 in the table we were able to prove that these presentations are AC-equivalent to those produced by applying the automorphism $a \mapsto b, b \mapsto a$. For each presentation, it took up to 10 minutes to complete the proof. For the other presentations in the table, we could not find a proof even when running search for much longer time.

These results demonstrate that the approach $\mathcal{S} \leftrightarrows \mathcal{C}$ can capture structural transformations that are difficult to obtain using automated reasoning alone. Having said this, the success of $\mathcal{S} \leftrightarrows \mathcal{C}$ with Table 3 is only partial, and further investigation is required to determine the full scope of applicability of $\mathcal{S} \leftrightarrows \mathcal{C}$.

To summarize, the experiments with presentations from [26] provide evidence showing that both approaches $\mathcal{S} \leftrightarrows \mathcal{C}$ and $\mathcal{AR}$ perform well on known solvable cases and struggle on currently open instances; $\mathcal{S} \leftrightarrows \mathcal{C}$ is able to find AC-equivalences in some situations where $\mathcal{AR}$ alone has not been successful.

8 Conclusion

In this paper we present a new way of searching for results to complement automated reasoning. Our combined approach employs a search algorithm and a classifier to find promising search directions, and we investigate whether it is beneficial to use them before automated reasoning is applied or whether they perform as well as automated reasoning. We test effectiveness of the new approach on a range of examples taken from a famously hard mathematical problem, the Andrews-Curtis conjecture. For the classifier part of our work, we introduce a new way to train automaton classifiers, explain their relationship to RNNs, and demonstrate their effectiveness. In future work, we will look at alternative training methods for the classifier, including studying alternative training loss functions, and expanding the search parameters to include $\vec{c}$.

Author Contributions Michael Fairbank wrote Sections 4, 5. Alexei Lisitsa wrote Sections 1, 2, 7. Alexei Vernitski wrote Sections 3, 6, 7. All authors contributed equally to writing code and conducting experiments.

Funding The research was conducted in the University of Essex and the University of Liverpool. There was no external funding supporting this research.

Data Availability Data related to the experiments presented in our paper is published at https://doi.org/10.5281/zenodo.16088111

Declarations

Competing Interests The authors declare no competing interests.

Appendix: Prover9 Configurations Used in Experiments

The experiments reported in this paper were performed using a small number of fixed Prover9 configurations, each tailored to a particular class of instances. For clarity and reproducibility, we list these configurations below. Each configuration is referenced in the main text by its label (C1, C2, etc.).

Configuration C1 (Th10-1 and Th10-2 Experiments)

This configuration was used for experiments on the presentations Th10-1 and Th10-2, including both implicational non-ground (IN) and equational non-ground (EN) encodings. All other parameters were left at their default values.

```
assign(order, kbo).
assign(max_weight,70).
assigne(max_megs,20000).
```

Configuration C2 ($\langle Th10-1\rangle$ and $MS_n(w_*)$ Experiments)

This configuration was used for experiments on the families $\langle Th10-1\rangle$) and $MS_n(w_*)$ All other parameters were left at their default values.

```
assign(order, kbo).
assign(max_weight,120).
assigne(max_megs,20000).
```

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, DG., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng. X.: Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, USENIX Association (2016)
2. Andrews, J., Curtis, M.L.: Free groups and handlebodies. Proc. Amer. Math. Soc. **16**, 192–195 (1965)
3. Akbulut, S., Kirby, R.: A potential smooth counterexample in dimension 4 to the Poincare conjecture, the Schoenflies conjecture, and the Andrews-Curtis conjecture. Topology **24**(4), 375–390 (1985)

4. Blaauwbroek, L., Cerna, D.M., Gauthier, T., Jakubův, J., Kaliszyk, C., Suda, M., Urban, J.: *Learning Guided Automated Reasoning: A Brief Survey*, page 54–83. Springer Nature Switzerland, (2024)
5. Bridson, M.R.: The complexity of balanced presentations and the Andrews-Curtis conjecture, (2015). ArXiv e-prints
6. Brown, R.: Coproducts of crossed p-modules: Applications to second homotopy groups and to the homology of groups. Topology **23**(4), 337–345 (1984)
7. Fernández, X.: Morse theory for group presentations. Trans. Am. Math. Soc. **377**, 2495–2523 (2024)
8. Havas, G., Ramsay, C.: Andrews-Curtis and Todd-Coxeter proof words. Technical report, in Oxford. Vol. I, London Math. Soc. Lecture Note Ser, (2001)
9. Havas, G., Ramsay, C.: Breadth-first search and the Andrews-Curtis conjecture. Internat. J. Algebra Comput. **13**(01), 61–68 (2003)
10. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
11. Miller III, C. F., Schupp, P. E.: *Some presentations of the trivial group*, volume 250 of *Contemp. Math.*, pages 113–115. Amer. Math. Soc., Providence, RI, (1999)
12. Ivanov, S.V.: On Conjectures of Andrews and Curtis. Proc. Amer. Math. Soc. **146**, 2283–2298 (2018)
13. Diederik, P., Kingma, Adam: A method for stochastic optimization, (2014). arXiv preprint arXiv:1412.6980
14. Krawiec, K., Swan, J.: AC-trivialization proofs eliminating some potential counterexamples to the Andrews-Curtis conjecture. Available at www.cs.put.poznan.pl/kkrawiec/wiki/uploads/Site/ACsequences.pdf, (2015)
15. Lishak, B.: Balanced finite presentations of the trivial group, (2015). ArXiv e-prints
16. Lisitsa, A.: The Andrews-Curtis Conjecture, Term Rewriting and First-Order Proofs. In *Mathematical Software - ICMS 2018 - 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings*, pages 343–351, (2018)
17. Lisitsa, A.: Automated reasoning for the Andrews-Curtis conjecture. In *AITP 2019, Fourth Conference on Artificial Intelligence and Theorem Proving, Abstracts of the Talks April 7–12, 2019, Obergurgl, Austria*, pages 82–83, (2019)
18. Lisitsa, A.: Towards computer-assisted proofs of parametric andrews-curtis simplifications. In *AITP 2023, 8th Conference on Artificial Intelligence and Theorem Proving, Abstracts of the Talks September 3–8, 2023, Aussois, France*, (2023)
19. Lisitsa, A.: New Andrews-Curtis trivializations for Miller-Schupp group presentations. Examples and Counterexamples **6**, 100168 (2024)
20. Lisitsa, A.: Automated Theorem Proving Reveals a Lengthy Andrews-Curtis Trivialization for a Miller-Schupp Trivial Group Presentation (2025). preprint available at https://doi.org/10.2139/ssrn.5100345
21. McCaul, S.B., Bowman, R.S.: Fast searching for Andrews-Curtis trivializations. Exp. Math. **2006**, 193–197 (2006)
22. McCune, W.: Prover9 and mace4. `http://www.cs.unm.edu/ mccune/prover9/`, 2005–2010
23. Miasnikov, A.D.: Genetic algorithms and the Andrews-Curtis conjecture. Internat. J. Algebra Comput. **09**(06), 671–686 (1999)
24. Magnus, W., Karrass, A., Solitar, D.: Combinatorial Group Theory: Presentations of Groups in Terms of Generators and Relations. Dover books on mathematics, Dover Publications (2004)
25. Pin, J.E.: Varieties of formal languages, Plenum Publishing Co (1986)
26. Panteleev, D., Ushakov, A.: Conjugacy search problem and the Andrews-Curtis conjecture. *ArXiv e-prints*, page arXiv:1609.00325, September (2016)
27. Rabin, M.O.: Probabilistic automata. Inf. Control **6**(3), 230–245 (1963)
28. Rall, L. B.: Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science*, 120, (1981)
29. Rumelhart, D.E., Hintont, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
30. Robert, I.B.: Proverx: rewriting and extending Prover9. https://api.semanticscholar.org/CorpusID:226185022, (2020)
31. Fathi, M.: Salem: Recurrent Neural Networks , (2022) Springer
32. Shehper, A., Medina-Mardones, A.M., Lewandowski, B., Gruen, A., Kucharski, P., Gukov, S.: What makes math problems hard for reinforcement learning: a case study. arxiv:2408.15332, (2024)
33. Swan, J.: Gabriela Ochoa, Graham Kendall, and Martin Edjvet. Fitness Landscapes and the Andrews-Curtis Conjecture. *IJAC*, 22(2), (2012)
34. Schäfer, A.M., Zimmermann, H.G.: Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 632–640. Springer, (2006)

35. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, Harvard University (1974). (PhD thesis)
36. Paul, J., Werbos: Backpropagation through time: what it does and how to do it. Proc. IEEE **78**(10), 1550–1560 (1990)
37. Werbos. P.J.: Backwards differentiation in AD and neural nets: Past links and new opportunities. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 15–34. Springer, (2005)