Knuth–Bendix for groups with infinitely many rules

David Epstein   Paul Sanders

April 12, 2024

Keywords: Automatic Groups, Knuth–Bendix Procedure, Finite State Automata, Word Reduction
Mathematics Subject Classification: Primary 20F10, 20–04, 68Q42; Secondary 03D40, 20F32.

Abstract

We introduce a new class of groups with solvable word problem, namely groups specified by a confluent set of short-lex-reducing Knuth–Bendix rules which form a regular language. This simultaneously generalizes short-lex-automatic groups and groups with a finite confluent set of short-lex-reducing rules. We describe a computer program which looks for such a set of rules in an arbitrary finitely presented group. Our main theorem is that our computer program finds the set of rules, if it exists, given enough time and space. (This is an optimistic description of our result — for the more pessimistic details, see the body of the paper.)

The set of rules is embodied in a finite state automaton in two variables. A central feature of our program is an operation, which we call welding, used to combine existing rules with new rules as they are found. Welding can be defined on arbitrary finite state automata, and we investigate this operation in abstract, proving that it can be considered as a process which takes as input one regular language and outputs another regular language.

In our programs we need to convert several non-deterministic finite state automata to deterministic versions accepting the same language. We show how to improve somewhat on the standard subset construction, due to special features in our case. We axiomatize these special

Funded by EPSRC grant no. GR/K 76597

features, in the hope that these improvements can be used in other applications.

The Knuth–Bendix process normally spends most of its time in reduction, so its efficiency depends on doing reduction quickly. Standard data structures for doing this can become very large, ultimately limiting the set of presentations of groups which can be so analyzed. We are able to give a method for rapid reduction using our much smaller two variable automaton, encoding the (usually infinite) regular language of rules found so far. Time taken for reduction in a given group is a small constant times the time taken for reduction in the best schemes known (see [4]), which is not too bad since we are reducing with respect to an infinite set of rules, whereas known schemes use a finite set of rules.

We hope that the method described here might lead to the computation of automatic structures in groups for which this is currently infeasible.

Contents

To help readers find their way around the inevitably complex structure of this paper, we start with a brief description of each section.

1. Introduction. This briefly sets some of the background for the paper and describes the motivation for this work.

2. Our class of groups in context. We define the class of groups to which this paper is devoted and prove various relations with related classes of groups. Groups in our class satisfy our main theorem (6.13Correctness of our Knuth–Bendix Proceduretheorem.6.13), which states that if the set of minimal short-lex reducing rules is regular, then our program succeeds in finding the finite state automaton which accepts these rules.

3. Welding. Here we describe one of the main new ideas in this paper, namely welding. This process can be applied to any finite state automaton. In our case it is the tool which enables us perform the apparently impossible task of generating an infinite set of Knuth–Bendix rules from a finite set. Welding has good properties from the abstract language point of view (see 3.5Welding in our exampletheorem.3.5). Welding has some important features. Firstly, if an automaton starts by accepting only pairs (u, v) such that $u = v$ in G, then the same is true after welding. Secondly, the welded automaton can encode infinitely many distinct equalities, even if the original only encoded a finite number. Thirdly, the welded automaton is usually much smaller than the original automaton. At the end of this section we show that any group determined by a regular set of rules is finitely presented.

4. Standard Knuth–Bendix. In this section, we describe the standard Knuth–Bendix process for string rewriting, in the form in which it is normally used to analyze finitely presented groups and monoids. We need this as a background against which to describe our modifications.
5. Our version of Knuth–Bendix. We give a description of our Knuth–Bendix procedure. We describe critical pair analysis, minimization of a rule and give some brief details of our method of reduction using a two-variable automaton which encodes the rules.
6. Correctness of our Knuth–Bendix Procedure. We prove that our Knuth–Bendix procedure does what we want it to do. The proof is not at all easy. In part the difficulty arises from the fact that we have to not only find new rules, but also delete unwanted rules, the latter in the interests of computational efficiency, or, indeed, computational feasibility. Our main tool is the concept of a Thue path (see 6.3Correctness of our Knuth–Bendix Proceduretheorem.6.3). Although it is hardly possible that this is a new concept, we have not seen elsewhere its systematic use to understand the progress of Knuth–Bendix with time. One hazard in programming Knuth–Bendix is that some clever manoeuvre changes the Thue equivalence relation. The key result here is 6.5Correctness of our Knuth–Bendix Proceduretheorem.6.5, which carefully analyzes the effect of various operations on Thue equivalence. In fact it provides more precise control, enabling other hazards, such as continual deletion and re-insertion of the same rule, to be avoided. It is also the most important step in proving our main result, 6.13Correctness of our Knuth–Bendix Proceduretheorem.6.13. This says that if our program is applied to a group defined by a regular set of minimal short-lex rules, then, given sufficient time and space, a finite state automaton accepting exactly these rules will eventually be constructed by our program, after which it will loop indefinitely, reproducing the same finite state automaton (but requiring a steadily increasing amount of space for redundant information).
7. Fast reduction. We describe a surprisingly pleasant aspect of our data structures and procedures, namely that reduction with respect to our probably infinite set of rules can be carried out very rapidly. Given a reducible word w, we can find a rule $(;)$, such that w contains as a subword, in a time which is linear in the length of w. Fast algorithms in computer science are often achieved by using finite state automata, and the current situation is an example. We explain how to construct the necessary automata and why they work.
8. A modified determinization algorithm. Here we describe a modification of the standard algorithm, to be found in every book about computing algorithms, that determinizes a non-deterministic finite state automaton. Our version saves space as compared with the standard one. It is well suited

to our special situation. We give axioms which enable one to see when this improved algorithm can be used.
9. Miscellaneous details. A number of miscellaneous points are discussed. In particular, we compare our approach to that taken in kbmag (see [4]).

1 Introduction

We give some background to our paper, and describe the class of groups of interest to us here.

A celebrated result of Novikov and Boone asserts that the word problem for finitely presented groups is, in general, unsolvable. This means that a finite presentation of a group is known and has been written down explicitly, with the property that there is no algorithm whose input is a word in the generators, and whose output states whether or not the word is trivial. Given a presentation of a group for which one is unable to solve the word problem, can any help at all be given by a computer?

The answer is that some help can be given with the kind of presentation that arises naturally in the work of many mathematicians, even though one can formally prove that there is no procedure that will always help.

There are two general techniques for trying to determine, with the help of a computer, whether two words in a group are equal or not. One is the Todd–Coxeter coset enumeration process and the other is the Knuth–Bendix process. Todd-Coxeter is more adapted to finite groups which are not too large. In this paper, we are motivated by groups which arise in the study of low dimensional topology. In particular they are usually infinite groups, and the number of words of length n rises exponentially with n. For this reason, Todd–Coxeter is not much use in practice. Well before Todd–Coxeter has had time to work out the structure of a large enough neighbourhood of the identity in the Cayley graph to be helpful, the computer is out of space.

On the other hand, the Knuth–Bendix process is much better adapted to this task, and it has been used quite extensively, particularly by Sims, for example in connection with computer investigations into problems related to the Burnside problem. It has also been used to good effect by Holt and Rees in their automated searching for isomorphisms and homomorphisms between two given finitely presented groups (see [6]). In connection with searching for a short-lex-automatic structure on a group, Holt was the first person to realize that the Knuth–Bendix process might be the right direction to choose (see [3]). Knuth–Bendix will run for ever on even the most innocuous hyperbolic triangle groups, which are perfectly easy to understand. Holt's successful plan was to use Knuth–Bendix for a certain amount of time, decided heuristically, and then to interrupt Knuth–Bendix and make a guess as to the automatic structure. One then uses axiom-checking, a part of automatic group theory (see [2, Chapter 6]), to see whether the guess is correct. If it isn't correct, the checking process will produce suggestions as to how to improve the guess. Thus, using the concept of an automatic group as a mechanism for bringing Knuth–Bendix to a halt has been one of the philosophical bases for the work done at Warwick in this field almost from the beginning. In addition to the works already cited in this paragraph, the reader may wish to look at [6] and [5].

For a short-lex-automatic group, a minimal set of Knuth–Bendix rules may be infinite, but it is always a regular language (see 2.11Recursive sets of rulestheorem.2.11), and therefore can be encoded by a finite state machine. In this paper, we carry this philosophical approach further, attempting to compute this finite state machine directly, and to carry out as much of the Knuth–Bendix process as possible using only approximations to this machine.

Thus, we describe a setup that can handle an infinite regular set of Knuth–Bendix rewrite rules. For our setup to be effective, we need to make several assumptions. Most important is the assumption that we are dealing with a group, rather than with a monoid. Secondly, our procedures are perhaps unlikely to be of much help unless the group actually is short-lex-automatic. Our main theorem — see 6.13Correctness of our Knuth–Bendix Proceduretheorem.6.13 — is that our Knuth–Bendix procedure succeeds in constructing the finite state machine which accepts the (unique) confluent set of short-lex minimal rules describing a group, if and only if this set of rules is a regular language.

Previous computer implementations of the semi-decision procedure to find the short-lex-automatic structure on a group are essentially specializations of the Knuth–Bendix procedure [7] to a string rewriting context together with fast, but space-consuming, automaton-based methods of performing word reduction relative to a finite set of short-lex-reducing rewrite rules. Since short-lex-automaticity of a given finite presentation is, in general, undecidable, space-efficient approaches to the Knuth–Bendix procedure are desirable. Our new algorithm performs a Knuth–Bendix type procedure relative to a possibly infinite regular set of short-lex-reducing rewrite rules, together with a companion word reduction algorithm which has been designed with space considerations in mind.

In standard Knuth–Bendix, there is a tension between time and space when reducing words. Looking for a left-hand side in a word can take a long time, unless the left-hand sides are carefully arranged in a data structure that traditionally takes a lot of space. Our technique can do very rapid reduction without using an inordinate amount of space (although, for other reasons, we have not been able to save as much space as we originally hoped). This is explained in 8A modified determinization algorithmsection.8.

We would like to thank Derek Holt for many conversations about this project, both in general and in detail. His help has, as always, been generous and useful.

2 Our class of groups in context

In this paper we study groups, together with a finite ordered set of monoid generators, with the property that their set of universally minimal short-lex rules is a regular language. In this section, we explain what this rather daunting sentence means, and we set this class of groups in the context of various other related classes, investigating which of these classes is included in which. In the next section, we will prove that groups in this class are finitely presented.

Throughout we will work with a group G generated by a fixed finite set A, and a fixed finite set of defining relations. Formally, we are given a map $A \to G$, but our language will sometimes (falsely) pretend that A is a subset of G. The reader is urged to remain aware of the distinction, remembering that, as a result of the insolubility of the word problem, it is not in general possible to tell whether the given map $A \to G$ is injective. We assume we are given an involution $\iota : A \to A$ such that, for each $x \in A$, $\iota(x)$ represents $x^{-1} \in G$. By A^* we mean the set of words (strings) over A. (Formally a word is a function $\{1, \ldots, n\} \to A$, where $n \geq 0$.) We also write $\iota : A^* \to A^*$ for the formal inverse map defined by $\iota(x_1 \ldots x_p) = \iota(x_p) \ldots \iota(x_1)$.

We assume we are given a fixed total order on A. This allows us to define the short-lex order on A^* as follows. We denote by $|u|$ the length of $u \in A^*$. If $u, v \in A^*$, we say that $u < v$ if either $|u| < |v|$ or u and v have the same length and u comes before v in lexicographical order. The short-lex representative of $g \in G$ is the smallest $u \in A^*$ such that u represents g. This is also called the short-lex normal form of g. If $u \in A^*$, we write $\overline{u} \in G$ for the element of G which it represents. If u is the short-lex representative of $\overline{u}$, we say that u is in short-lex normal form.

Suppose we have $(G; A)$ as above. Then there may or may not be an algorithm that has a word $u \in A^*$ as input and as output the short-lex representative of $\overline{u} \in G$. The existence of such an algorithm is equivalent to the solubility of the word problem for G, since there are only a finite number of words v such that $v < u$.

A natural attempt to construct such an algorithm is to find a set R of replacement rules, also known as Knuth–Bendix rules. In this paper, a replacement rule will be called simply a rule, and we will restrict our attention to rules of a rather special kind. A rule is a pair $(u;v)$ with $u > v$ Given a rule $(u;v)$, u is called the left-hand side and v the right-hand side. The idea of the algorithm is to start with an arbitrary word w over A and to reduce it as follows: we change it to a smaller word by looking in w for some left-hand side u of some rule $(u;v)$ in R. We then replace u by v in w (this is called an elementary reduction) and repeat the operation until no further elementary reductions are possible (the repeated process is called a reduction). Eventually the process must stop with an R-irreducible word, that is a word which contains no subword which is a left-hand side of R.

2.1 Thue equivalence. Given a set of rules R, we write $u \to_R v$ if there is an elementary reduction from u to v, that is, if there are words and over A and a rule $(;) \in R$ such that $u =$ and $v =$. Thue equivalence is the equivalence relation on A generated by elementary reductions.

There is a multiplication in A given by concatenation. This induces a multiplication on the set of Thue equivalence classes. We will work with rules where the set of equivalence classes is isomorphic to the group G.

By no means every set of rules can be used to find the short-lex normal form of a word constructively. We now discuss the various properties that a set of rules should have in order that reduction to an irreducible always gives the short-lex normal form of a word. First we give the assumptions that we will always make about every set of rules we consider. When constructing a new set of rules, we will always ensure that these assumptions are correct for the new set.

2.2 Standard assumptions about rules.

1. [Condition] For each $x \in A$, $x.(x)$ is Thue equivalent to the trivial word . The preceding condition is enough to ensure that the set of Thue equivalence classes is a group. If $r = s$ is a defining relation for G, then r is Thue equivalent to s. This ensures that the group of Thue equivalence classes is a quotient of G.

2. [Condition] If $(u;v)$ is a rule of R, then $u > v$ and $\overline{u} = \overline{v} \in G$. This ensures that the group of Thue equivalence classes is isomorphic to G.

2.3 Confluence. [Condition] This property is one which we certainly desire, but which is hard to achieve. Given w, there may be different ways to reduce w. For example we could look in w for the first subword that is a left-hand side, or for the last subword, or just look for a left-hand side which is some random subword of w. We say that R is confluent if the result of fully reducing w gives an irreducible that is independent of which elementary reductions were used.

2.4 Lemma. [Lemma] If a set R of rules satisfies the conditions of 2.2 and 2.3 then the set of R-irreducibles is mapped bijectively to G and multiplication corresponds to concatenation followed by reduction. Under these assumptions, an R-irreducible is in short-lex normal form, and conversely; moreover, each Thue equivalence class contains a unique irreducible.

Proof: The homomorphism $A \to G$ is surjective and, by 2.2.2Standard assumptions about rulesItem .2, elementary reduction does not change the image in G. It follows that the induced map from the set of irreducibles to G is surjective. Suppose u and v are irreducibles such that $\overline{u} = \overline{v} \in G$. Then $u \cdot (v) = 1_G$. Therefore $u \cdot (v)$ is equal in the free group generated by A (with (x) equated to the formal inverse of x, for each $x \in A$) to a word s which is a product of formal conjugates of the defining relators. Now $u \cdot (v)$ and s reduce to the same word, using only reductions that replace $x \cdot (x)$, where $x \in A$, by the trivial word . By Condition 2.2.1, s can be reduced to . It follows from Condition 2.3 that $u \cdot (v)v$ can be reduced to v. It can also be reduced to u, using Condition 2.2.1 again, and the fact that $: A \to A$ is an involution. It follows from Condition 2.3 that $u = v$, as required.

The description of the multiplication of irreducibles follows from the fact that multiplication in A is given by concatenation and the fact that the map $A \to G$ is a homomorphism of monoids.

Since reduction reduces the short-lex order of a word, a word in short-lex least normal form must be R-irreducible. Conversely, if u is R-irreducible, let v be the short-lex normal form of $\overline{u}$. Then v is also R-irreducible, as we have just pointed out, and u and v represent the same element of G. Since the map from irreducibles to G is injective, we deduce that $u = v$. Therefore u is in short-lex normal form.

To show that each Thue equivalence class contains a unique irreducible, we note that if there is an elementary reduction of u to v, then, in case of confluence, any reduction of u gives the same answer as any reduction of v.

2.5 Recursive sets of rules. [Condition] Another important property (lacked by some of the sets of rules we discuss) is the condition that the set of rules be a recursive set. As opposed to the usual setup when discussing rewrite systems, we do not require R to be a finite set of rules — in fact, in this paper R will normally be infinite. To say that R is recursive means that there exists a Turing machine which can decide whether or not a given pair (u, v) belongs to R.

2.6 Definition. [Definition] We denote by U the set of all rules of the form (u, v), where $u > v$ and $\overline{u} = \overline{v} \in G$. U is called the universal set of rules. Note that a word is U-irreducible if and only if it is in short-lex normal form. $\square$

2.7 Lemma. The existence of a set of rules R satisfying the conditions of 2.2, 2.3 and 2.5 is equivalent to the solubility of the word problem in G and in this case U defined in 2.6 is such a set of rules.

Proof: On the one hand, if we have such a set R, then we can solve the word problem by reduction—according to Lemma 2.4 a word w reduces to the trivial word if and only if $\overline{w} = 1_G$.

On the other hand, if the word problem is solvable, then the set U of Definition 2.6 is recursive. The various conditions on a set of rules follow for U.

U can be difficult to manipulate, even for a very well-behaved group G and a finite ordered set A of generators, and we therefore restrict our attention to a much smaller subset, namely the set of U-minimal rules, which we now define.

2.8 Definition. [Definition] Let R be a set of rules for a group G with generators A. We say that a rule $(u, v) \in R$ is R-minimal if v is R-irreducible and if every proper subword of u is R-irreducible. $\square$

2.9 Proposition. [Proposition]

1. The set of U-minimal rules satisfies the conditions of 2.2 and 2.3. In particular they are confluent.
2. Let (u, v) be a U-minimal rule and let $u = u_1 \ldots u_{n+r}$ and $v = v_1 \ldots v_n$. Then the following must hold: $0 \le r \le 2$; if $n > 0$, $u_1 \neq v_1$; if $n > 0$, then $u_{n+r} \neq v_n$; if $r = 0$ and $n > 0$, then $u_1 > v_1$; if $r = 2$ and $n > 0$, then $u_1 < v_1$ and $u_2 < (u_1)$; if $r = 2$ and $n = 0$, then $u_1 \quad (u_2)$ and $u_2 \quad (u_1)$.
3. The set of U-minimal rules is recursive if and only if G has a solvable word problem.

Proof: If w is U-reducible, let u be the shortest prefix of w which is U-reducible. Then every subword of u which does not contain the last letter is U-irreducible. Let v be the shortest suffix of u which is U-reducible. Then every proper subword of v is U-irreducible. Let s be the short-lex normal form for v. Then (v, s) is a U-minimal rule. Replacing v in w by s gives an elementary reduction by a U-minimal rule. It follows that reduction of w using only U-minimal rules eventually gives us a U-irreducible word, and this must be the short-lex normal form of w. Therefore the conditions of 2.2 and 2.3 are satisfied by the set of U-minimal rules.

We now prove 2.9.2. Since $u > v$ in the short-lex order, $|u| \geq |v|$. So $r \geq 0$. If $r > 2$, then $\overline{u} = \overline{v}$ gives rise to $\overline{u_2 \ldots u_{n+r}} = \overline{\iota(u_1) v_1 \ldots v_n}$. Therefore $u_2 \ldots u_{n+r}$ is not in short-lex normal form. It follows that $u_2 \ldots u_{n+r}$ is U-reducible. Therefore (u, v) is not U-minimal. Similar arguments work for the other cases. This completes the proof of 2.9.2.

Clearly U-minimality of a rule can be detected by a Turing machine if the word problem is solvable. Conversely, if the set of U-minimal rules is recursive, then the word problem can be solved by reduction using only U-minimal rules.

Now we have a uniqueness result for the set of minimal rules.

2.10 Lemma. Let R satisfy the conditions of 2.2 and 2.3. Suppose every rule of R is R-minimal. Then R is equal to the set of U-minimal rules.

Proof: By Lemma 2.4, the R-irreducibles are the same as the words in short-lex normal form. Let (u, v) be a rule in R. Then v is R-irreducible and therefore in short-lex normal form. Also every proper subword of u is in short-lex normal form. Therefore (u, v) is in U and is U-minimal.

Conversely, suppose (u, v) is U-minimal. Then v is the short-lex normal form of $\overline{u}$. By Lemma 2.4 for R, u must be R-reducible. Every proper subword of u is already in short-lex normal form. It follows that there is a rule (u, w) in R. Since this rule is R-minimal, w is R-irreducible. Therefore w is the short-lex normal form of $\overline{u}$. It follows that $v = w$. Therefore every U-minimal rule is in R.

We are interested in those pairs (G, A), where G is a group and A is an ordered set of generators, such that the set of U-minimal rules is not only recursive, but is in fact regular. We now explain what we mean by regular in this context.

We recall that a subset of A^* is called regular if it is equal to $L(M)$, the language accepted by some finite state automaton over A. (See Definition 3.2, where finite state automata are discussed.) We need to formalize what it means for an automaton to accept pairs of words over an alphabet A. If the pair of words is (abb, cdc), then we have to pad the shorter of the two words to make them the same length, regarding this pair as the word of length four $(a,c)(b,c)(b,d)(\$,c)$. In general, given an arbitrary pair of words $(u,v) \in A^* \times A^*$, we regard this instead as a word of pairs by adjoining a padding symbol \$ to A and then "padding" the shorter of u and v so that both words have the same length. We obtain a word over $A \cup \{\$\} \times A \cup \{\$\}$. The alphabet $A \cup \{\$\}$ is denoted A^+ and is called the padded extension of A. The result of padding an arbitrary pair (u,v) is denoted $(u,v)^+$. A word $w \in (A^+)^* \times (A^+)^*$ is called padded if there exists $u,v \in A^*$ with $w = (u,v)^+$ (that is, at most one of the two components of w ends with a padding symbol and there are no padding symbols in the middle of a word).

A set R of pairs of words over A is called regular if the corresponding set of padded words is a regular language over the product alphabet $A^+ \times A^+$. We say that R is accepted by a two-variable finite state automaton over A.

2.11 Theorem. Let G be a group and let A be a finite set of generators, closed under taking inverses. If (G, A) is short-lex automatic, then the set of U-minimal rules is regular.

Having a finite confluent set of rules does not imply short-lex automatic. A counter-example is given in [2, page 118]. So the converse of this theorem is not true.

Proof: Since we have a short-lex automatic structure, the set L of short-lex normal forms is a regular language. If $x \in A$, the automatic structure includes the multiplier M_x, which is a two-variable automaton over A. The language $L(M_x)$ is the set of pairs (u,v), such that $u,v \in L$ and $\overline{ux} = \overline{v}$. It is not hard to construct from the union of the M_x an automaton whose language P is the set of (u,v) such that $\overline{u} = \overline{v} \in G$, $u \in L.A$ and $v \in L$.

We know that $(L.A \cap A.L) \cap (A^* \setminus L)$ is a regular language. Clearly, this is the set of left-hand sides of U-minimal rules, since it is the set of U-reducible words such that each proper subword is U-irreducible. The set of pairs $(u,v) \in P$, such that u is a left-hand side of a U-minimal rule is easily seen to be the set of all U-minimal rules.

2.12 Question. Suppose (G, A) has a finite confluent set R of short-lex reducing rules which define G. Then it is easy to construct from this a finite confluent set R' of R'-minimal rules defining G. The method is to use minimization, as described in 5.7. This set of rules is equal to the set of U-minimal rules by 2.10Recursive sets of rulestheorem 2.10.

Suppose now that $(G;A)$ has an infinite confluent set R of short-lex-reducing rules defining G, and this set is regular. Is the set of U-minimal rules also regular? We know that it is confluent and recursive by 2.9 Recursive sets of rules theorem 2.9, since R provides a solution to the word problem.

If R contains all U-minimal rules, then the answer is easily seen to be yes. The answer is not clear to us if R does not contain all minimal rules. There is no loss of generality in making R smaller so that each proper subword of each left-hand side is irreducible. But we see no way of changing R so as to ensure that each right-hand side is irreducible, while maintaining R's property of being regular.

2.13 Objective. In this paper we present a procedure which, given a set of rules satisfying the conditions of 2.2, changes the set of rules so that it becomes "more confluent". More precisely, the set of words for which all reductions give the same irreducible, and this irreducible is in short-lex normal form, increases with time. If we fix attention on a single word this will eventually be included in the set. However, in general, because of the insolubility of the word problem, it is not in general possible to know when that time has been arrived at.

For a group where the set of all U-minimal rules (see Definition 2.6) is the set of all pairs accepted by a two-variable minimal PDFA M (these concepts are defined in 3.2), our procedure gives rise to M after a finite number of steps.

For many undecidable problems, there is a "one-sided" solution. The technical language is that a certain set is recursively enumerable, but not recursive. For example, consider a fixed group for which the word problem is undecidable. Given a word w in the generators, if you are correctly informed that $\overline{w} = 1_G$, then this can be verified by a Turing machine. All that you have to do is to enumerate products of conjugates of the defining relators, reduce them in the free group on the generators, and see if you get w, also reduced in the free group. If w represents the identity then you will prove this sooner or later. If it's not the identity, the process continues for ever.

We know that there is no algorithm which has as input a finite presentation of a group and outputs whether the group is trivial or not (see [9]). It follows easily that there is no algorithm which has as input a finite presentation and outputs either an FSA accepting the set of U-minimal rules or correctly answers There is no such FSA. For, in the case of the trivial group, the set of U-minimal rules is finite—for each element $x \in A$, we have the rule $(x;\varepsilon)$—and so it is certainly regular.

But the situation is even worse than this. We do not even know of a one-sided solution to the problem of whether the set of U-minimal rules is regular. If the set of U-minimal rules is regular, our procedure will eventually produce a candidate with some indication that it is correct, but we will not know for sure whether the answer is correct or incorrect.

What is at issue is whether there is an algorithm which has as its input a regular set of short-lex rules for a group and outputs whether or not the set of rules is confluent. For finite sets of rules the question of confluence is decidable by classical critical pair analysis which we describe in 4Standard Knuth–Bendixsection.4. However, for infinite rewriting systems the confluence question is, in general, undecidable. Examples exhibiting undecidability are given in [8]. They are length-reducing rewriting systems R which are regular in a very strong sense: R contains only a finite number of right-hand sides and for each right-hand side r, the set $\{l : (l,r) \in R\}$ is a regular language. These examples are in the context of rewriting for monoids. As far as we know, there is no known example of undecidability if we add to the hypothesis that the monoid defined by R is in fact a group.

In the special case where (G,A) is short-lex automatic, there is a test for confluence of a set of rules satisfying the conditions of 2.2, namely the axiom-checking procedure described in theory in [2] and carried out in practice in Derek Holt's kbmag programs [4].

3 Welding

[Section]

In this section we start with an example which motivates the operation of welding. We then give a formal definition, and prove that the operation gives rise to a function from the set of regular languages to the set of regular languages. We then define the concept of a rule automaton — this is a finite state automaton in two variables which can recognize when certain words in the generators are equal in the associated group. We show that a welded rule automaton is also a rule automaton.

3.1 A motivating example. We will use the standard generators x, y, and their inverses X and Y for the free abelian group on two generators. We will impose different orderings on this set of four generators, and, as described in 2.13, see what kind of confluent sets of rules emerge.

Consider the alphabet $A = \{x, X, y, Y\}$ with the ordering $x < X < y < Y$, and denote the identity of A^* by ϵ. Let R be the rewriting system on A^* defined by the set of rules

$$\{(xX,\epsilon), (Xx,\epsilon), (yY,\epsilon), (Yy,\epsilon), (yx,xy), (yX,Xy), (Yx,xY), (YX,XY)\}.$$

It is straightforward to see that R is a confluent system.

We now change the ordering of the set of generators to $x < y < X < Y$ and correspondingly interchange the sides of the sixth rule getting (Xy, yX) and an order reducing set of rules. Once again the rules define the free abelian group on two generators. But this time there can be no finite confluent set of rules. To see this, we consider the set of words $\{xy^nX : n \in \mathbb{N}\}$. None of these is in short-lex normal form. By 2.4Confluencetheorem.2.4, each of these words is reducible relative to any confluent set of rules. On the other hand, each proper subword of one of the words xy^nX is clearly in short-lex normal form and is therefore irreducible. It follows that a confluent set of rules must contain each of the words xy^nX as a left-hand side. In this situation, the classical Knuth–Bendix procedure (see 4Standard Knuth–Bendixsection.4) will never terminate, and the same is true for any method of which generates only a finite number of rules at each step.

We will now introduce a new procedure, which we call welding. This can produce an infinite set of rules from a finite set of rules in a finite number of steps. Welding is central to the main procedure of the computer program described in this paper.

First we need to give some standard definitions.

3.2 Definition. [Definition] A finite state automaton (abbreviated FSA) M over a finite alphabet A is a finite graph with directed edges and the following additional properties. Each edge (called an arrow in this context) is either labelled with an element of A or is unlabelled. Unlabelled arrows are sometimes labelled with ε, which stands for the empty word, and are called ε-transitions. The vertices of the graph are called states. Some of the states are labelled as initial states and some as final states. The language $L(M)$ accepted by M is the set of words over A which are traced out by paths of arrows which start at some initial state and end at some final state. An FSA is said to be partially deterministic (abbreviated PDFA) if it has no ε-transitions, if there is exactly one initial state and if, for each state s and each $x \in A$, there is at most one arrow from s with label x. An FSA is said to be trim if, for each state s, there is a path of arrows which starts at an initial state, and ends at a final state, with s lying on the path. The reversal of a finite state automaton is the same graph with the same labelling, but with each arrow reversed, with each initial state changed to be a final state and each final state changed to be an initial state. A non-deterministic automaton NFA is an automaton with ε-transitions and/or some states s having more than one arrow from s having the same label. $\square$

3.3 Definition. An FSA is called welded if it is partially deterministic, trim and has a partially deterministic reversal. These conditions imply that, given $x \in A$ and a state t, there is at most one x-arrow with target t and also that there is exactly one initial state and one final state. $\square$

Given a trim non-empty FSA M, we can form a welded automaton from it as follows. Given any ε-arrow (s, ε, t), we may identify s with t. Given distinct initial states s_1 and s_2, we may identify s_1 with s_2. Given distinct final states t_1 and t_2, we may identify t_1 with t_2. Given distinct arrows (s, x, t_1) and (s, x, t_2), we may identify t_1 with t_2. Given distinct arrows (s_1, x, t) and (s_2, x, t), we may identify s_1 with s_2. Immediately after any identification of two states, we change the set of arrows accordingly, omitting any ε-arrow from a state to itself. Since the number of states continually decreases, this process must come to an end, and at this point the automaton is welded.

3.4 Welding in our example. Let us see how this works on the example given in 3.1. For the moment we won't try to justify the correctness of our procedure, that is, that the new rules that welding produces are valid rules; we will just carry out the procedure to show how it works. Justification comes from the consideration of rule automata — see 3.9Welding in our exampletheorem.3.9.

We consider the rule $r_n = (xy^nX, y^n)$ for some $n \in \mathbb{N}$. The corresponding padded word r_n^+ gives rise to an $(n+3)$-state PDFA $M(r_n)$ whose accepted language consists solely of the rule r_n. For $n > 2$ this PDFA is shown in Figure 1.

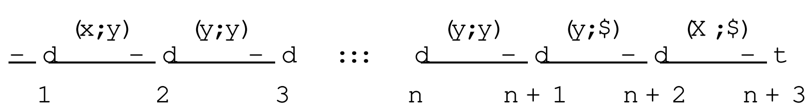

Figure 1. The PDFA $M(r_n)$ for $n > 2$.

Continuing the discussion of the rules for a free abelian group on two generators, we define M_n to be the disjoint union $\bigcup\{M(r_1), \ldots, M(r_n)\}$ of the automata $M(r_1), \ldots, M(r_n)$, with set of initial (final) states equal to the collection of initial (final) states for the various $M(r_i)$. If $n > 1$ then $\mathrm{Weld}(M_n)$ is isomorphic to the PDFA given in Figure 2, and the accepted language of this PDFA is the set of rules $\{r_i : i \in \mathbb{N}\}$. This is independent of n if $n > 1$.

So in this example, after only two steps, the welding procedure provides us with a PDFA whose accepted language consists of an infinite set of identities between words in the free abelian group. Moreover, by using this PDFA to define a suitable reduction procedure, each of the words xy^nX with $n \in \mathbb{N}$ can be reduced to the short-lex normal form.

For this group with the given ordering on the generators, it is not hard to show that by welding the original defining rules for the group together with the 4 rules $\{(xyX, y), (xy^2X, y^2), (yXY, X), (yX^2Y, X^2)\}$, we obtain a PDFA whose accepted language is a confluent set of rules (provided we adjust the automaton to ensure that only padded pairs of words $(u, v)^+$ are accepted, with $u > v$). Any reduction procedure using this infinite set of rules will reduce any word to its short-lex normal form.

The next theorem is a general result about the welding of finite state automata which need have nothing to do with groups. It's a result which is reassuring, but, logically, it is entirely unnecessary for understanding other parts of this paper. Readers pressed for time should skip it.

3.5 Theorem. Given a trim non-empty FSA M, all welded automata obtained from it as above (no matter in what order the states and arrows are identified to each other) are the same, except that the names of the states may be different. The automaton Q thus obtained is a minimal PDFA and Q depends only on the language $L(M)$, up to changing the names of the states. It follows that welding can be regarded as an operation on regular languages, independent of the automaton used to encode them.

Proof: For each $x \in A$, let x^{-1} be its formal inverse and let A^{-1} be the set of these formal inverses. We form from M an automaton over $A \cup A^{-1}$ by adjoining an arrow of the form (t, x^{-1}, s) for each arrow (s, x, t) of M, and adjoining an arrow (t, ε, s) for each arrow (s, ε, t) unless it's already there. We also adjoin (s_1, ε, s_2) if s_1 and s_2 are either both initial states or both final states, unless these arrows are already there. We denote this new automaton by N. N has the same initial and final states as M.

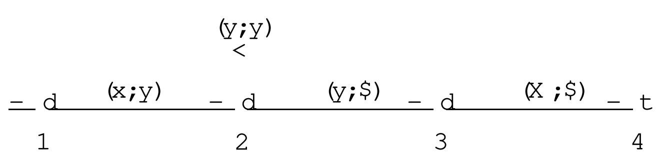

Figure 2. A PDFA isomorphic to $\mathrm{Weld}(M_n)$, $n > 1$.

Let F be the free group generated by A. We define a relation on the set of states of N by $s \sim t$ if there is a path of arrows from s to t in N whose label gives the identity element of F. This is clearly an equivalence relation. Let Q be the automaton defined as follows. Each state of Q is one of the equivalence classes above. The unique initial state of Q is the unique equivalence class containing all initial states of N. The unique final state of Q is the unique equivalence class containing all final states of N. Let S be one equivalence class and T another, and let $x \in A$. We have an arrow $x : S \to T$ in Q if there is an $s \in S$ and a $t \in T$ and an arrow $x : s \to t$ in M. It is easy to see that Q is welded, and it follows that it is a partial deterministic automaton.

If M starts out by being welded, then it is easy to see that $Q = M$, up to the naming of states.

Consider the identifications of states and arrows made during welding (see the passage following 3.3A motivating exampletheorem.3.3). Let $M = M_0, M_1, \ldots, M_k$ be the sequence of automata obtained by identifying at each step only one state with another state or deleting one arrow labelled x from a state s to state t if there are several arrows labelled x from s to t or deleting one ε-arrow from a state to itself. Here M_k, the last automaton in the list, is a welded automaton.

We assign to each state s of M_i the set of all states of the original automaton M which are identified to make s. A state q of $Q(M_i)$ is a set of states of M_i, and this is a set of subsets of the state set of M. By taking the union, we can instead regard q as a set of states of M. This loses some of the structure, but only an irrelevant part.

With this interpretation, we see that the states of $Q(M_i)$ are identical to those of $Q(M_{i+1})$. Moreover, all arrows in $Q(M_i)$ are inherited from M via M_i. It follows that the automaton $Q(M_i)$ is independent of i. So we have $Q = Q(M) = Q(M_k) = M_k$. This shows that Q is independent of the order in which the identifications are carried out. In fact Q can be characterized as the largest welded quotient of M.

We claim that every element of $L(Q)$ arises as follows, and that only elements of $L(Q)$ arise in this way. Let $(w_1, w_2, \ldots, w_{2k+1})$ be a $(2k+1)$-tuple of elements of $L(M)$, where $k \geq 0$. Now consider

$$w_1 w_2^{-1} \ldots w_{2k}^{-1} w_{2k+1} \in F,$$

and write it in reduced form, that is, cancel adjacent formal inverse letters wherever possible. If the result is in A^*, that is, if after cancellation there are no inverse symbols, then it is in $L(Q)$.

To prove this claim, we proceed as follows. For each state s of M, we fix a path of arrows p_s in M from an initial state to s and a path of arrows q_s

from s to a final state. If s is an initial state, we define p_s to be the trivial path. If s is a final state, we define q_s to be the trivial path.

Start with an arbitrary element $w \in L(Q)$. We must show that w can be produced in the way described above. Now w is the label of a path of arrows in Q, starting from the initial state of Q and ending at the final state of Q. Recalling the definition of a state of Q, we can replace this path by a path of arrows in N, which alternately traverses a path of arrows in N labelled by a word over $A \cup A^{-1} \cup \{\varepsilon\}$ which reduces to the identity element in F, and an arrow of N labelled by a letter in w. The path in N starts at an initial state of N and ends at a final state of N. We write the path as a composite of arrows u_i in N.

If $u_i : s \to t$ is an arrow in M, we replace it by $p_s^{-1}(p_s u_i q_t) q_t^{-1}$. Otherwise, if the inverse of $u_i : s \to t$ is an arrow of M, we replace u_i by $q_s \; q_s^{-1} u_i p_t^{-1} \; p_t$. (We consider the inverse of an ε-arrow to be an ε-arrow.) Otherwise s and t are both initial states or both final states and u_i is an ε-arrow and we leave u_i unaltered.

Each expression within parentheses in the preceding paragraph therefore give either some $w_i \in L(M)$ (possibly empty) or the formal inverse of such a word. Outside these parentheses we obtain expressions like ε, $q_s^{-1}q_s$, $p_s p_s^{-1}$, $p_s q_s$ or $q_s^{-1} p_s^{-1}$. In the first three cases, we omit the expressions. In the last two cases, the expression represents either $w_i \in L(M)$, or the formal inverse of such a word. The path starts at an initial state of N and ends at a final state. So, if the set of initial states is disjoint from the set of final states, then the expression of w as a product in the free group F of elements of $L(M)$ and their formal inverses must have an odd number of factors. If the set of initial states meets the set of final states, then the trivial word is an element of $L(M)$, and we can use this to make sure that the number of factors is odd. This completes the claim in one direction.

Conversely, suppose we are given the $w_i \in L(M)$ as in the claim. Then w_i is the label on a path of arrows in M from an initial state to a final state. By inserting ε-arrows in N to join initial states or to join final states, we find that $w_1 w_2^{-1} \dots w_{2k}^{-1} w_{2k+1}$ is the label of a path of arrows in N from an initial state to a final state. An elementary cancellation in F corresponds to the fact that two states of N give rise to the same state of Q. Carrying out all the elementary cancellations possible, if we are left only with a word over A, we have defined a path of arrows in Q from the initial state of Q to the final state of Q. So we have found an element of $L(Q)$, as claimed.

A welded automaton is minimal. For let s and t be distinct states, and let u and v be words over A which lead from s and t respectively to the unique final state. Then u does not lead from t to the final state and v does not lead from s to the final state (otherwise s and t would be equal). It follows that s and t remain distinct in the minimized automaton.

If M is a non-empty trim FSA, we denote by $\mathrm{Weld}(M)$ the PDFA obtained from it by welding. To compute $\mathrm{Weld}(M)$ efficiently, we first add "backward arrows" to M. That is, for each arrow (s,x,t) in M, including ε-arrows, we add the arrow (t,x',s), where x' represents a backwards version of x. We also add ε-arrows to connect the initial states, and ε-arrows to connect the final states. We then make use of a slightly modified version of the coincidence procedure of Sims given in [10, 4.6]. When this stops we have a welded automaton.

In practice, in the automata which we want to weld, backward arrows are needed in any case for some algorithms which we need. The procedure described in the preceding paragraph therefore fits our needs particularly well.

For the welding procedure to be used in a general Knuth–Bendix situation, we need to show that any rules obtained are valid identities in the corresponding monoid. We now show that if the monoid is a group (the situation we are interested in), any rules obtained are valid identities.

3.6 Definition. [Definition] Let A be a finite inverse closed set of monoid generators for a group G and, as before, denote images under the surjection $(A^+)^* \to G$ by overscores. A rule automaton for G is a two-variable FSA $M = (S, A^+ \times A^+, \delta, F, S_0)$ together with a function $\mu_M : S \to G$ satisfying

1. $F, S_0 \neq \emptyset$.
2. If s is an initial or final state then $\mu_M(s) = 1_G$.
3. For any $s,t \in S$ and $(x,y) \in A^+ \times A^+$ with $(s,(x,y),t) \in \delta$ we have $\mu_M(t) = \overline{x}^{-1}\mu_M(s)\overline{y}$.
4. For any $s,t \in S$ with $(s,\varepsilon,t) \in \delta$ we have $\mu_M(s) = \mu_M(t)$. $\square$

3.7 Example. If A is a finite inverse closed set of monoid generators for a group G and $r = (u,v) \in A^* \times A^*$ satisfies $\overline{u} = \overline{v}$ then, as in Figure 1, writing r^+ as a word $(u_1,v_1)\cdots(u_n,v_n) \in (A^+ \times A^+)^*$, we obtain an $(n+1)$-state rule automaton $M(r) = (\{s_0,\ldots,s_n\}, A^+ \times A^+, \delta, \{s_0\}, \{s_n\})$ for G where the arrows are given by

$$\delta(s_i,(u_{i+1},v_{i+1})) = s_{i+1},\ 0 \le i \le n-1.$$

The function $\mu = \mu_{M(r)}$ assigning group elements to states is defined inductively by $\mu(s_0) = 1_G$ and $\mu(s_i) = \overline{u_i}^{-1}\mu(s_{i-1})\overline{v_i}$ for $1 \le i \le n$. As usual, the padding symbol is sent to 1_G. The fact that $\overline{u} = \overline{v}$ ensures that Condition 2 of 3.6Welding in our exampletheorem.3.6 is satisfied. $\square$

3.8 Remark. For a two-variable FSA M which is a rule automaton, the PDFA P obtained by applying the subset construction to the (non-empty) set of initial states of M (and the sets that arise), is also a rule automaton for G, where the map σ_P is induced from σ_M. The fact that this map is well-defined follows from Conditions 2, 3 and 4 of 3.6Welding in our exampletheorem.3.6 and the fact that P is connected (by construction).

The same remark applies to the modified subset construction described in Section 8. □

3.9 Proposition. Let A be a finite inverse closed set of monoid generators for a group G and suppose that M is a rule automaton for G. Then

1. Every pair $(u,v) \in L(M)$ gives a valid identity $\overline{u} = \overline{v}$ in G.
2. $\mathrm{Weld}(M)$ is a rule automaton for G.

Consequently every accepted rule (that is, an accepted pair (u,v) such that $u > v$) of $\mathrm{Weld}(M)$ is a valid identity in G.

Proof: To prove 3.9.1, let $r = (u,v) \in A^* \times A^*$ be an accepted rule of M and write the padded word $(u,v)^+$ as $(u_1,v_1)\cdots(u_n,v_n)$. Then in the PDFA P obtained from M (as in 3.8Welding in our exampletheorem.3.8), there exists a sequence of states $s_0,\ldots,s_n$ of P, such that s_0 is the initial state, s_n a final state, and, for each $i, 1 \le i \le n$, there is a arrow from s_{i-1} to s_i labelled by (u_i,v_i). Hence, from Condition 3 of 3.6Welding in our exampletheorem.3.6, we have

$$\sigma_P(s_i) = \overline{u_1 \cdots u_i}^{\,-1}\,\overline{v_1 \cdots v_i}, \text{ for all } i \text{ with } 0 \le i \le n.$$

Condition 2 of 3.6Welding in our exampletheorem.3.6 tells us that $\sigma_P(s_n) = e$. It follows that $\overline{u_1 \cdots u_n} = \overline{v_1 \cdots v_n}$, and therefore the rule r is valid in G.

To prove 2, we need only show that when any of the operations described just after 3.3A motivating exampletheorem.3.3 is applied to a rule automaton M, we continue to have a rule automaton. This is obvious. The final statement is now immediate.

3.10 Corollary. Let A be a finite inverse closed set of monoid generators for a group G and suppose that $r_1,\ldots,r_m \in A^* \times A^*$ give valid identities in G. Then any rule accepted by $\mathrm{Weld}(M(r_1),\ldots,M(r_m))$ also gives a valid identity in G.

Proof: For $1 \le k \le m$ let $M(r_k)$ be the rule automaton for G as in 3.7 Welding in our example the theorem 3.7. Then the disjoint union $\bigcup \{M(r_1), \ldots, M(r_m)\}$ is also a rule automaton for G and so the result follows by 3.9.

3.11 Remark. Given a rule automaton M for a group G, the map $_M$ may not be injective. In order to think of the matter constructively, we specify the values of $_M$ by representing them as words in the generators. The undecidability of the word problem implies that the injectivity of $_M$ might be impossible to decide, though sometimes we are in a position to know whether $_M$ is injective or not. Even if $_M$ is not injective, the rule automaton M can still be useful for finding equalities in the group G. M may not tell the whole truth, but it does tell nothing but the truth. However, if $_M(s) = {}_M(t)$ and we can somehow determine that this is the case, then we can connect s to t by an -arrow, and we still have a rule automaton. If we then weld, s and t will be identified. In this way, with sufficient investigation, we can hope to make $_M$ injective in particular cases, even though we know that in general this is an impossible task. □

3.12 Theorem. Let G be a group and let A be a finite set of generators, closed under taking inverses. If G is determined by a regular set of short-lex-reducing rules, then G is finitely presented.

Proof: Let M be the finite state automaton accepting the rules in our regular set. Then M can be given the structure of a rule automaton, associating to each state of M a word over A. By 3.6 Welding in our example the theorem 3.6, each arrow $(x; y) : s \to t$ in M gives rise to a relation of the form $_M(t) = \overline{x}^{-1}\,{}_M(s)\overline{y}$. There are only a finite number of these, and they can clearly be combined to prove that $u = v$ for any $(u; v)$ accepted by M. It follows that this finite set of relators is a defining set for G.

4 Standard Knuth–Bendix.

[Section]

We recall the classical Knuth–Bendix procedure. Later we will explain how our procedure differs from it. We continue to restrict to the short-lex case and to groups. Suppose G is a group given by a finite set of generators and relators. We define A to be the set of generators together with their formal inverses. Our initial set of rules consists of all rules of the form $(x\,(x);\ )$ for

$x \in A$, together with all rules of the form $(r;\varepsilon)$, where r varies over the finite set of defining relators for G.

After running the Knuth–Bendix procedure (which we are about to describe) for some time, we will still have a finite set R of rules. As always, we assume that R satisfies Conditions 2.2.

To test for confluence of a finite set of rules, we need only do critical pair analysis, as explained in 4.1, 4.2 and 4.3. The proof of this is as follows.

Suppose R is not confluent. Let w be the short-lex least word over A for which there are two different chains of elementary reductions giving rise to distinct irreducibles. Since w is shortest, it is easy to see that the first elementary reductions in the two chains must overlap.

4.1 Critical pair analysis. A pair of rules $(\lambda_1;\rho_1)$ and $(\lambda_2;\rho_2)$ can overlap in two possible ways. First, a non-empty word z may be a suffix of $\lambda_1 = s_1 z$ and a prefix of $\lambda_2 = z s_2$ (or vice versa). Second, λ_2 may be a subword of λ_1 (or vice versa) and we write $\lambda_1 = s_1 \lambda_2 s_2$.

These cases are not disjoint. In particular, if one of s_1 and s_2 is trivial in the second case, it can equally well be treated under the first case with z equal either to λ_1 or to λ_2.

4.2 First case of critical pair analysis. In the first case, there are two elementary reductions of $u = s_1 z s_2$, namely to $\rho_1 s_2$ and to $s_1 \rho_2$. Further reduction to irreducibles either gives the same irreducible for each of the two computations, or else gives us distinct irreducibles v and w. From Conditions 2.2 we deduce that v and w represent the same element of G. So, if v and w are distinct, we augment R with the rule $(v;w)$ if $w < v$ or with $(w;v)$ if $v < w$. Clearly Conditions 2.2 are maintained.

Note that it is important to allow $(\lambda_1;\rho_1) = (\lambda_2;\rho_2)$ in the case just discussed, provided there is a z which is both a proper suffix and a proper prefix of $\lambda_1 = \lambda_2$.

4.3 Second case of critical pair analysis. In the second case, there are two elementary reductions of $u = \lambda_1 = s_1 \lambda_2 s_2$, namely to ρ_1 and to $s_1 \rho_2 s_2$. If ρ_1 and $s_1 \rho_2 s_2$ reduce to distinct irreducibles v and w, we augment R with either $(v;w)$ or with $(w;v)$, depending on whether $v > w$ or $w > v$.

4.4 Omitting rules. In practice, it is important to remove rules which are redundant, as well as to add rules which are essential. Omitting rules is unnecessary in theory, provided that we have unlimited time and space at our disposal. In practice, if we don't omit rules, we are liable to be overwhelmed by unnecessary computation. Moreover, nearly all programs in computational group theory su er from excessive demands for space. Indeed this is one of the reasons for developing the algorithms and programs discussed in this paper. So it is important to throw away information that is not needed and doesn't help.

For this reason, in Knuth{Bendix programs one looks from time to time at each rule (;) to see if it can be omitted. If a proper subword of the left-hand side can be reduced, then we are in the situation of 4.3. If the two reductions mentioned in 4.3 lead to the same irreducible, we omit (;) from the set of rules. If the two reductions lead to di erent irreducibles, then we augment the set of rules as described in 4.3 and again omit (;). We also investigate whether the right-hand side of a rule (;) is reducible to ^{0}. If so, we can omit (;) from R and replace it with the rule (; ^{0}).

It is easy to see that such omissions do not change the Thue equivalence classes. The process of analyzing critical pairs and augmenting or diminishing the rule set while maintaining the conditions of 2.2 is called the Knuth{Bendix Process.

If the Knuth{Bendix process terminates, every left-hand side having been checked against every left-hand side in critical pair analysis without any new rule being added, we know that we have a nite con uent system of rules. Usually it does not terminate and it produces new rules ad in nitum.

4.5 De nition. [De nition] It is important that the process be fair. By this we mean that if you x your attention on two rules at any one time, then either their left-hand sides must have already been, or must eventually be, checked for overlaps; or one or both of them must eventually be omitted. If the process is not fair, it might concentrate exclusively on one part of the group: for example, in the case of the product of two groups, the process might pay attention only to one of the factors. [2]

4.6 The limit of the process. As the Knuth{Bendix process proceeds, R changes and the set of R-reducibles steadily increases. This is obvious when we add a rule as in 4.2 and 4.3. It is also easy to see when we omit a rule| we need only check that if we omit (;) from R as in 4.4, then remains reducible.

Now let us x a positive integer n. Eventually the set of reducibles of length at most n stops increasing with time, and the set of irreducibles of length at most n stops decreasing. Since the word problem is in general insoluble, we will in general not know for sure at any one time or for any xed n whether the set of reducibles has stopped increasing. It may look as though it has permanently stabilized and then suddenly start increasing again.

Once stabilized, we know by 4.5Omitting rulestheorem.4.5 that any two reductions of a given word of length at most n will give the same irreducible (otherwise a new rule would be added at some time, creating one of more new reducibles of length at most n). It follows that if we take the limit of the set of rules (the set of rules which appear at some time and are never subsequently omitted), then we have a con uent set of rules. We deduce from 2.4Con uencetheorem.2.4 that, after stabilization of the set of reducibles of length at most n, any irreducible of length at most n is in short-lex normal form. In fact, at this point, the set of rules with left-hand side of length at most n coincides with the set of U-minimal rules in U (de ned in 2.6 and 2.8).

4.7 Knuth{Bendix pass. One procedure for carrying out the Knuth{Bendix process is to divide the nite set S of rules found so far into three disjoint subsets. The rst subset, called Considered, is the set of rules whose left-hand sides have been compared with each other and with themselves for overlaps. The second set of rules, called Now, is the set of rules waiting to be compared with those in Considered. The third set, called New, consists of those rules most recently found. Here we only sketch the process. Fuller details of our more elaborate form of Knuth{Bendix are provided in 5Our version of Knuth{Bendixsection.5.

The Knuth{Bendix process proceeds in phases, each of which is called a Knuth{Bendix pass. Each pass starts by looking at each rule in Considered and seeing whether it can be deleted as in 4.4. Consideration of an existing rule in Considered can lead to a new rule, in which case the new rule is added to New.

Next, we look at each rule r in New to see if it is can be omitted or replaced by a better rule, a process which we call minimization. The details of our minimization procedure will be given in 5.7. If the minimization procedure changes a rule, the old rule is either deleted or marked for future deletion. The new rule is added to Now. Eventually New is emptied.

We then look at each rule in Now. Its left-hand side is compared with itself and with all the left-hand sides of rules in Considered, looking for overlaps as in 4.2. Any new rules found are added to New. Then r is moved into Considered. Eventually Now becomes empty.

We then proceed to the next pass.

5 Our version of Knuth{Bendix.

[Section]

In this section we consider a rewriting system which is the accepted language of a rule automaton for some finitely presented group. We call the automaton Rules. We describe a Knuth–Bendix type algorithm for such a system. In light of the undecidability results mentioned in 2.13, our algorithm does not provide a test for confluence. We can however use our procedure together with other procedures which handle short-lex-automatic groups, to prove confluence by an indirect route, provided the group is short-lex-automatic. Details of the theory of how this is done can be found in [2]. The practical details are carried out in programs by Derek Holt— see [4].

We will introduce the concept of Aut-reduction, that is, reduction using a two-variable automaton, which we call Rules, encoding our possibly infinite set of rules. We prove some results about how reducibility may change with time.

5.1 **Properties of the rule automaton.** The most important data structure is a small two-variable PDFA which we call Rules. Roughly speaking, this accepts all the rules found so far. It has the following properties.

1. Rules is a trim rule automaton.
2. Rules has one initial state and one final state and they are equal.
3. Rules and its reversal Rev(Rules) are both partially deterministic.
4. Any arrow labelled (x,x), with either source or target the initial state, has source equal to target. has source the initial state. If this condition is not fulfilled, we can identify the source and target of the appropriate (x,x)-arrows, and then weld. We will still have a rule automaton. Later on (see Lemmas 7.2 and 7.3) we will show that (after any necessary identifications and welding) we can omit such arrows without loss, and, in fact, with a gain given by improved computational efficiency. Apart from the passages proving these lemmas, we will assume from now on that there are no arrows labelled (x,x) with source or target the initial state of Rules.

The first three conditions imply that Rules is welded. Since Rules is a rule automaton, Proposition 3.9 shows that each accepted pair $(u,v) \in L(\text{Rules})$ gives a valid identity $u = v$ in G.

5.2 **The automaton SL2.** The automaton Rules may accept pairs (u,v) such that u is shorter than v. We cannot consider such a pair as a rule and so we want to exclude it. To this end we introduce the automaton SL2.

This is a five state automaton, depicted in Figure 3, which accepts pairs $(u,v) \in A^* \times A^*$, such that u and v have no common prefix, u is short-lex-greater than v and $|v| \le |u| \le |v|+2$. By combining SL2 with Rules, we obtain a regular set of rules Set(Rules), which is possibly infinite, namely $L(\text{Rules}) \cap L(\text{SL2})$. An automaton accepting this set can be constructed as follows. Its states are pairs (s,t), where s is a state of Rules and t is a state of SL2. Its unique initial state is the pair of initial states in Rules and SL2. A final state is any state (s,t) such that both s and t are final states. Its arrows are labelled by (x,y), where $x \in A$ and $y \in A^+$. Such an arrow corresponds to a pair of arrows, each labelled with (x,y), the first from Rules and the second from SL2.

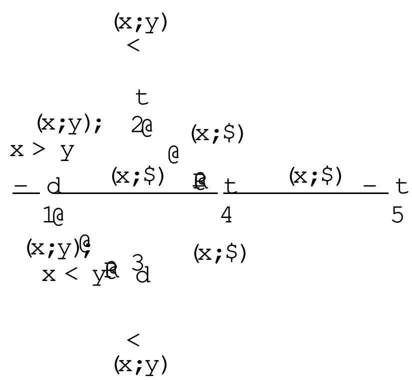

Figure 3. The automaton SL2. Solid dots represent final states. Roman letters represent arbitrary letters from the alphabet A and the labels on the arrows indicate multiple arrows. For example, from state 2 to itself there is one arrow for each pair in $A \times A$.

5.3 Restrictions on relative lengths. The following discussion is closely connected with 2.9Recursive sets of rulestheorem 2.9. The restriction $|u| \le |v|+2$ needs some explanation. The point is that if we have a rule with $|u| > |v|+2$, then we have an equality $u = v$ in G. We write $u = u'x$, where $x \in A$. The formal inverse X of x is also an element of A. We therefore have a pair of words (u', vX) which represent equal elements in G. If our set of rules were to contain such a rule, then $u = u'x$ would reduce to vXx, and this reduces to v, making the rule (u,v) redundant. This leads to an obvious technique for transforming any rule we find into a new and better rule with $|v| \le |u| \le |v|+2$. Since we take this into account when constructing the automaton Rules, we are justified in making the restriction.

This analysis can be carried further. Let $u = u_1 \cdots u_{r+2} = u'u_{r+2} = u_1u''$

and let $v = v_1 \quad {}_r$. Vf $u_1 > v_1$, then the rule $(u;v)$ can be replaced by the better rule $(u^0; vu_{r+2}^{\,1})$. If $u_2 > u_1^{\,1}$, then $(u;v)$ can be replaced by $(u^{00}; u_1^{\,1}v)$. We do in fact carry out these steps when installing new rules. The extra information could have been included in the FSA SL2. However, it seems that this would involve more complicated coding at various points, probably without any gain in efficiency.

We could consider the steps just described as an attempt to force our structures to define a set of rules which conforms to known properties (see 2.9Recursive sets of rulestheorem .2.9) of the set of U-minimal rules (see 2.6 for the definition of U). The most important reason for insisting on these additional restrictions on our rules is to keep down the size of our data structures.

5.4 The basic structures. The basic structures used in our procedure are:

1. A two-variable automaton Rules satisfying the conditions laid down in 5.1. When we want to specify that we are working with the Rules automaton during the nth Knuth{Bendix pass (see 4.7 for the definition of a Knuth{Bendix pass), we will use the notation Rules[n]. We extract explicit rules from Rules[n] by taking elements of the intersection Set(Rules[n]) = L(Rules[n]) \ L(SL2). The two-variable automaton SL2 was defined in Section 5.2 and is depicted in Figure 3.

2. A finite set S of rules, which is the disjoint union of several subsets of rules : Considered, Now, New and Delete. One point of the separate subsets is to avoid constantly doing the same critical pair analyses. Another point is to ensure that our Knuth{Bendix process is fair (see 4.5Omitting rulestheorem .4.5). The reason for holding some rules in a Delete list, rather than delete them immediately, is to make reduction more efficient. This will be explained further in 5.8.3.

   S will continually change, while Rules is constant during a Knuth{Bendix pass. We change Rules at the end of each Knuth{Bendix pass. We will perform the Knuth{Bendix process, using the rules in S for critical pair analysis, as described in 4.1.

3. Considered is a subset of S such that each rule has already been compared with each other rule in Considered, including with itself, to see whether left-hand sides overlap. The consequent critical pair analysis has also been carried out for pairs of rules in Considered. Such rules do not need to be compared with each other again.

4. Now is a subset of S (empty at the beginning of each Knuth–Bendix pass) containing rules which we plan to use during this pass to compare for overlaps with the rules in Considered, as in 4.2. These rules are minimal for the current pass (see 5.7) and so should not be minimized again.

5. New is a subset of S containing new rules which have been found during the current pass, other than those which are output by the minimization routine (see 5.7 for the meaning of "minimization"). Rules which are output by the minimization routine are added to Now.

6. Delete is a subset of S containing rules which are to be deleted at the end of this pass.

7. The two-variable automaton WDiff contains all the states and arrows of Rules[n], and possibly other states and arrows. It satisfies the conditions of 5.1. This automaton is used to accumulate appropriate new rules which are output by the minimization routine. As rules are considered during the Knuth–Bendix pass, states and arrows of WDiff are marked as needed. At the end of the pass, other states and arrows are removed, and WDiff becomes the new Rules automaton Rules[n + 1].

8. A PDFA P(Rules) formed from Rules by a certain subset construction. This automaton accepts words which are Aut-reducible, that is, words which contain a left-hand side of a rule in Set(Rules). The automaton is used as part of our rapid reduction procedure (see 7Fast reductionsection.7). More details of P(Rules) are provided in 7.5.

9. A PDFA Q(Rules) which accepts the reversals of left-hand sides of rules in Set(Rules). This is also formed from Rules by a subset construction and is also used for rapid reduction. More details of Q(Rules) are provided in 7.9.

5.5 Initial arrangements. Before describing the main Knuth–Bendix process, we explain how the data structures are initially set up. Let R be the original set of defining relations together with special rules of the form (x: (x);) which make the formal inverse (x) into the actual inverse of x.

We rewrite each relation of R in the form of a relator, which we cyclically reduce in the free group. We assume that each relator has the form l: (r), where l and r are elements of A and (l;r) is accepted by SL2.

For each rule (l;r), including the special rules (x: (x);), we form a rule automaton, as explained in 3.7. These automata are then welded together to form the two-variable rule automaton WDiff satisfying the conditions of 5.1. Each state and arrow of WDiff is marked as needed. Each of these rules is inserted into New. Considered, Now and Delete are initially empty. Set Rules[1] = WDiff.

5.6 The main loop — a Knuth–Bendix pass. We now describe the procedure followed during the course of a single Knuth–Bendix pass.

A significant proportion of the time in a Knuth–Bendix pass is spent in applying a procedure which we term *minimization*. Each rule encountered during the pass is input (often after a delay) to this procedure and the output is called a *minimal rule*. The details of this process are given in sections 5.7 and 5.8.

1. At the beginning of a Knuth–Bendix pass, Now is empty. If $n > 0$, save space by deleting previously defined automata P(Rules[n]), Q(Rules[n]) and Rules[n]. Increment n. The integer n records which Knuth–Bendix pass we are currently working on.

2. [Step] For each rule $(\ ; \ )$ in Considered, minimize $(\ ; \ )$ as in 5.7 and handle the output rule $({}_1; {}_1)$ as in 5.8. This may affect S and WDiff.

3. [Step] For each rule $(\ ; \ )$ in New, minimize $(\ ; \ )$ as in 5.7 and handle the output as in 5.8. This may affect S and WDiff.

   Since rules added to New during minimization are always strictly smaller than the rule being minimized (see 5.10), it follows that the process of examining rules in New does not continue indefinitely. As a result, we can be sure that our process is fair (see 4.5).

4. For each rule $(\ ; \ )$ in Now:

   (a) Delete the rule from Now and add it to Considered.

   (b) [Step] For each rule $({}_1; {}_1)$ in Considered:

   Look for overlaps between and ${}_1$. That is we have to find each suffix of which is a prefix of ${}_1$ and each suffix of ${}_1$ which is a prefix of . Then Aut-reduce in two different ways as in 4.2, obtaining a pair of words $(u; v)$ with $u \ v$. (Roughly speaking, Aut-reduction means the use of rules in Set(Rules). More precision is provided in 5.10.) If $u > v$, $(u; v)$ is inserted into New, unless it is already in S.

Note that we may have to allow $=\ _1$ in order to deal with the case where two di erent rules have the same left-hand side. In this case, both the pre x and su x of both left-hand sides is equal to $=\ _1$.

5. WDi was possibly a ected in 5.6.2The main loop| a Knuth{Bendix passItem .26 and 5.6.3The main loop| a Knuth{Bendix passItem .27. With WDi in its present form, delete from WDi all arrows and states which are not marked as needed. Copy WDi into Rules[n + 1] and mark all arrows and states of WDi as not needed.

6. Delete the rules in Delete.

7. This ends the description of a Knuth{Bendix pass. Now we decide whether to terminate the Knuth{Bendix process. Since we know of no procedure to decide con uence of an in nite system of rules (indeed, it is probably undecidable), this decision is taken on heuristic grounds. In our context, a decision to terminate could be taken simply on the grounds that WDi and Rules[n] have the same states and arrows. In other words, no new word-di erences or arrows between word-di erences have been found or deleted during this pass. If the Knuth{Bendix process is not terminated, go to 5.6.1.

5.7 De nition. [De nition]We now provide the details of the minimization routine. This processes a rule so as to create from it a minimal rule (see 2.8Recursive sets of rulestheorem .2.8), where, roughly speaking, minimality is de ned using the current set of rules. Since the set of rules is changing, this is a bit di cult to pin down. So instead we make the following de nition, which is more precise, though the underlying concept is the same. Let $(u;v)$ 2 A A and let $u = u_1 \quad _p$ and $v = v_1 \quad _q$, where $u_i;v_j$ 2 A. We say that $(u;v)$ is a minimal rule if $u \neq v$, $u = v$ in G and the following procedure does not change $(u;v)$. The procedure is called the minimization routine. We always start the minimization routine with $u > v$, though this condition is not necessarily maintained as u and v change during the routine. Here the meaning of a \minimal rule" changes with time: a rule may be minimal at one time and no longer minimal at a later time.

1. Aut-reduce (that is, reduce using the rules of Rules) the maximal proper pre x $u_1 \quad _p\ _1$ of u obtaining u^0. Reduction may result in rules being added to New as described in 7.14.5. If $u \neq u^0u_p$, change u to u^0u_p and go to Step 5.7.3.

2. Aut-reduce the maximal proper suffix $u_2 \cdots u_p$ of u obtaining u^{00}. Reduction may result in new rules being added to New. Replace u by $u_1 u^{00}$.

3. If u has changed since the original input to the minimization routine, then Aut-reduce u as explained in 7.14. This may result in rules being added to New as described in 7.14.5.

4. [Step] [Step] Aut-reduce v.

5. If $v > u$, interchange u and v.

6. If (a) $p > q + 2$ or (b) if $p = q + 2$, $q > 0$ and $u_1 > v_1$ or (c) if $p = 2$, $q = 0$ and $u_1 > (u_2)$, replace $(u;v)$ by $(u_1 \cdots_p u; v_1 \cdots_q v(u_p))$ and repeat this step until we can go no further.

7. If $p = q + 2$ and $u_2 > (u_1)$, replace $(u;v)$ by $(u_2 \cdots_p; u(u_1)v_1 \cdots_q)$.v

8. If $q > 0$ and $u_1 = v_1$, cancel the first letter from u and from v and repeat this step.

9. If $q > 0$ and $u_p = v_q$, cancel the last letter from u and from v and repeat this step.

10. If $(u;v)$ has changed since the last time Step 5.7.4 was executed, go to Step 5.7.4.

11. Output $(u;v)$ and stop. 2

Note that the output could be $(\, ; \,)$, which means that the rule is redundant. Otherwise we have output $(u;v)$ with $u > v$. Note that the minimization procedure keeps on decreasing $(u;v)$ in the ordering given by using first the short-lex-ordering on u and then, in case of a tie, the short-lex-ordering on v. Since this is a well-ordering, the minimization procedure has to stop.

5.8 Handling minimization output. Suppose the input to minimization is $(\, ; \,)$ and its output is $({}_1; {}_1)$.

1. If $({}_1; {}_1) \neq (\, ; \,)$, incorporate (by welding) $({}_1; {}_1)$ into the language accepted by WDi . Insert $({}_1; {}_1)$ into Now if it was not already in Now or Considered. Remove it from New, if it was there previously.

2. If some proper subword of is Aut-reducible, then this will be discovered during the first few steps of minimization. $(({}_1; {}_1) = (\, ; \,)$

turns out to be a special case of this, as we will see in 5.11.1.) In this case, delete (;) from S immediately the minimization procedure is otherwise complete.

3. If, at the time of minimization, all proper subwords of were Aut-irreducible and if (;) was not minimal, move (;) to the Delete list. The reason for this possibly surprising policy of not deleting immediately is that further reduction during this pass may once again produce as a left-hand side by the methods of 7 and 7.6. We want to avoid the work involved in finding the right-hand side by the method which will be explained in 7.13. For this, we need to have a rule in S with left-hand side equal to | see 7.14.5.

5.9 Details on the structure of WDiff. At the beginning of Step 5.6.5, each state s of WDiff is associated to a word w_s 2 A which is irreducible with respect to Set(Rules[n]). WDiff is a rule automaton: the rule automaton structure is given by associating the element $\overline{w_s}$ 2 G to the state s. Whenever a minimal rule r is encountered during the nth pass, it is adjoined to the accepted language of WDiff by welding and the corresponding states and arrows are marked as needed. State labels are calculated as and when new states and arrows are added to WDiff.

At the end of the nth Knuth{Bendix pass, WDiff is an automaton which represents the word-differences and arrows between them encountered during that pass. At this stage the word attached to each state is irreducible with respect to the rules in Set(Rules[n]) but not necessarily with respect to the rules implicitly contained in WDiff. Before starting the next pass, we Aut-reduce the state labels of WDiff with respect to Set(WDiff). If WDiff now contains distinct states labelled by the same word we connect them by epsilon arrows and replace WDiff by Weld(WDiff). We then repeat this procedure until all states are labelled by distinct words which are irreducible with respect to Set(WDiff). If during this procedure a state or arrow marked as needed is identified with another which may or may not be marked as needed, the resulting state or arrow is marked as needed.

5.10 Aut-reduction and inserting rules. Given a word w, we look for an Aut-reducible subword such that all proper subwords of are Aut-irreducible, by looking in Set(Rules). Later (7Fast reductionsection.7) we will describe how to do this quickly, but, at the moment, the reader can just think of a non-deterministic search in the automaton giving the short-lex rules recognized by Rules. Having found a reducible subword of w, with no reducible subword, we do not automatically use the corresponding right-hand side , found from the exploration of Rules, because this naive approach is computationally inefficient. Instead we look in S to see if there is a rule (;). If there is such a rule, then we can find it quickly given , and we proceed with our reduction, replacing the subword in w with .

It may however turn out that we can find an Aut-reducible subword of w, with no Aut-reducible subwords, and yet there is no rule of the form (;) in S. In this case, we have to spend time finding such a rule in Set(Rules). Once found, we immediately insert it into S, otherwise the logic of the Knuth–Bendix procedure can go wrong.

In this way, reduction of a single word can result in the insertion of several new rules into S.

It follows from the above description that the Aut-reducibility of a word w depends only on Rules. Since Rules does not change during a Knuth–Bendix pass, exactly the same subset of A will be Aut-reducible throughout such a pass. However, because we may use rules in the changing set S, the result of Aut-reduction may change during a pass.

Another, more conventional, source of rules to insert into S come from critical pair analysis in 5.6.4bThe main loop— a Knuth–Bendix passItem .30.

Minimization also results in rules being added to S, both directly, as the output of the minimization procedure, but also indirectly because minimization uses reduction, and, as we will see in 7.13. reduction can add rules to S. It is important to note that any rules added to S during the minimization of a rule (;) are strictly smaller than (;), if we order such pairs by using first and then in case of a tie. We used this fact when discussing 5.6.3The main loop— a Knuth–Bendix passItem .27.

5.11 Deleting rules. Deletion of rules happens only at the end of each minimization step, and at the end of each pass, when rules marked for deletion are actually deleted. During a Knuth–Bendix pass, deletion does not occur after the beginning of Step 5.6.4. Suppose that the output from minimization of (;) $\in$ S is ($_1$; $_1$).

1. [Case] If every proper subword of is Aut-irreducible, then $_1$ is a non-trivial subword of . This follows by going through the successive steps of minimization (5.7The main loop— a Knuth–Bendix passtheorem .5.7). These change and , while maintaining the inequality > . In particular $_1$ > $_1$, so that $_1$ $\neq$. If ($_1$; $_1$) $\neq$ (;), then we delete (;) after a delay. The mechanism is to mark it for deletion by moving it to the Delete list and actually delete it only at the end of the current Knuth–Bendix pass (Step 5.6.6).

2. [Case] If some proper subword of is reducible, then (;) is immediately deleted from S at Step 5.8.2 at the end of the minimization procedure. (Aut-reducibility of some proper subword of is discovered at Step 5.7.1 or 5.7.2.)

5.12 Lemma. Suppose that, for some n 2 N, there is a rule (;) 2 S during the n-th Knuth{Bendix pass, before the beginning of Step 5.6.4. Then there is a non-trivial subword of such that some rule (;) is output from some instance of the minimization procedure during the n-th pass. If = , then . The rule (;) is a rule in S at the beginning of the (n + 1)-st pass and is accepted by Rules[n + 1].

Proof: By examining 5.6, we see that (;) must be the input to the minimization routine at some time during the n-th pass. (We check the four possibilities, namely that it is in Considered, Now, New or Delete, one by one. If it is in Delete, it must have been the input to the minimization procedure at some earlier stage during the n-th pass.)

We rst deal with the case where some proper subword of is Aut-reducible during the n-th pass. During the rst three steps of minimization (5.7The main loop| a Knuth{Bendix passtheorem.5.7), an Aut-reducible subword of is found, with the property that all the proper subwords of are Aut-irreducible. Minimization then either nds a rule of the form (;) already in S, or such a rule is added to New by the reduction process| see 7.14.5. In any case, it will either be minimized during this pass, or it has already been minimized (and possibly moved to the Delete list.

At the moment when (;) is minimized during the n-th pass, we must be in Case 5.11.1. So the output ($_1$; $_1$) from the minimization procedure with input (;) gives the required rule. $_1$ is a subword of and is a proper subword of .

Alternatively, all proper subwords of are Aut-irreducible during the n-th pass, in which case we set (;) to be the output from minimization of (;). By 5.11.1, is a non-trivial subword of . If = , then .

5.13 Lemma. Suppose that, for some n 2 N, there is a rule (;) 2 S during the n-th Knuth{Bendix pass, after the beginning of Step 5.6.4. Then there is a non-trivial subword of such that some rule (;) is output from some instance of the minimization procedure during the (n + 1)-st pass. If = , then .

Proof: If (;) is in the Delete list, then it must have been input to the minimization procedure at some earlier time during the n-th pass. By 5.11.2Deleting rulesItem .49, every proper subword of must have been found to be Aut-irreducible during the n-th pass. Let (^{0}; ^{0}) be the output from minimization. By 5.11.1Deleting rulesItem .48, ^{0} is a non-trivial subword of , and, if ^{0} = , then ^{0} < . Now (^{0}; ^{0}) is in S at the beginning of the (n + 1)-st pass. We apply 5.12Deleting rulestheorem .5.12 to (^{0}; ^{0}) at the (n + 1)-st pass.

If (;) is not on the Delete list, then it must be in S at the beginning of the (n+1)-st pass. Once again, we can apply 5.12Deleting rulestheorem .5.12.

The following result is often applied with w = .

5.14 Proposition. Let w 2 A be a word which contains the left-hand side of a rule (;) input to the minimization routine during the n-th Knuth{Bendix pass. Then, for m n, w contains the left-hand side of a rule which is input to the minimization procedure during the m-th Knuth{Bendix pass. Moreover w is Aut-reducible for m > n.

Proof: We assume inductively that if m > n then w contains a subword , such that a rule of the form (;) is input to the minimization procedure during the (m 1)-st pass. Since minimization happens only before the beginning of Step 5.6.4, 5.12Deleting rulestheorem .5.12 gives a rule (;), such that is a non-trivial subword of . Moreover, (;) is minimal during the (m 1)-st pass and is contained in S at the beginning of the m-th pass. Therefore (;) is input to the minimization procedure during the m-th pass, as required.

The rule (;) is welded into WDi during the (m 1)-st pass and is therefore accepted by Rules[m]. It follows that w is Aut-reducible during the m-th pass. Inductively this is true for all m > n.

6 Correctness of our Knuth{Bendix Procedure

In this section we will prove that the procedure set out in Section 5 does what we expect it to do. One hazard in programming Knuth{Bendix is that some seemingly clever manoeuvre changes the Thue equivalence relation. The key result here is 6.5Correctness of our Knuth{Bendix Proceduretheorem .6.5, which carefully analyzes the effect of our various operations on Thue equivalence. In fact it provides more precise control, enabling other hazards, such as continual deletion and re-insertion of the same rule, to be avoided. It is also the most important step in proving our main result, 6.13Correctness of our Knuth–Bendix Proceduretheorem.6.13. This says that if our program is applied to a group defined by a regular set of minimal rules, then, given sufficient time and space, a finite state automaton accepting exactly these rules will eventually be constructed by our program, after which the program will loop indefinitely, repeatedly reproducing the same finite state automaton (but requiring a steadily increasing amount of space for redundant information).

6.1 Definition. [Definition] For a discrete time t, we denote by $S(t)$ the rules in S at time t in our Knuth—Bendix procedure. We take t to be the number of elementary steps since the start of the program, assuming the program is expressed in some sort of pseudocode. Any other similar measure of time would do equally well. $\square$

6.2 Definition. A quintuple $(t; s_1; s_2; \lambda; \rho)$, where t is a time, and s_1, s_2, λ and ρ are elements of A^*, is called an elementary $S(t)$-reduction $u \to_{S(t)} v$ from u to v if $(\lambda; \rho)$ is a rule in $S(t)$, $u = s_1 \lambda s_2$ and $v = s_1 \rho s_2$. We call $(\lambda; \rho)$ the rule associated to the elementary reduction. $\square$

We now define the main technical tool that we will use in this section.

6.3 Definition. Let $t \geq 0$. By a time-t Thue path between two words w_1 and w_2, we mean a finite sequence of elementary $S(t)$-reductions and inverses of elementary $S(t)$-reductions connecting w_1 to w_2, such that none of the rules associated to the elementary reductions is in Delete at time t. We talk of the words which are the source or target of these elementary reductions as nodes. The path is considered as having a direction from w_1 to w_2. The elementary reductions in our path will be consistent with this direction and will be called rightward elementary reductions. The inverses of elementary reductions in our path will be in the opposite direction and will be called leftward elementary reductions. $\square$

All our insertions and deletions of rules have been organized so that the following result holds.

6.4 Proposition. Let $\langle A/R \rangle$ be the finite presentation of a group G at the start of the Knuth–Bendix process. Then the group defined by subjecting the free group generated by A to all relations of the form $\lambda = \rho$ as $(\lambda; \rho)$ varies over $S(t)$ is at all times t isomorphic to G with the isomorphism being induced by the unchanging map $A \to G$.

6.5 **Proposition.** Let $t \geq 0$ and suppose that we have a Thue path from u to v in $S(t)$ with maximum node w. Then for any time $s \geq t$, there exists a time-s Thue path from u to v with each node less than or equal to w.

Proof: Note that, given a Thue path, we may assume, if we wish, that no node is repeated, because we could shorten the path to avoid repetition. We show by induction on s that, if at some time $t \leq s$ there is a Thue path between words u and v with all nodes no bigger than $\max(u,v)$, then there is also such a Thue path at time s. So suppose that we have proved this statement for all times $s' < s$.

We first consider the special case where $r_0 = (u,v)$ is a rule being input to the minimization routine (see Definition 5.7) at time t, and s is the time at the end of the subsequent invocation of the minimization handling routine 5.8. There is a Thue path (of length one) from u to v at time t. By induction we are assuming that at time $s-1$ there is a Thue path from u to v with maximum node u. We must show that there is such a Thue path at time s.

One possibility is that r_0 is already minimal, in which case there is a Thue path of length one from u to v, both at the beginning and at the end of minimization. So we assume that r_0 is not minimal. Then the last step in 5.8 is that either r_0 is placed in the Delete list or else r_0 is simply deleted immediately.

What we need to show therefore is that the Thue path p from u to v, which exists at time $s-1$, does not use an elementary reduction coming from r_0. It is part of our inductive hypothesis that the largest node occurring on p is u, and we have already pointed out that we can assume there is no repetition of nodes along p.

Each step of minimization takes an input pair of words and outputs a possibly different pair of words which is used as the input to the next step. The initial input is $r_0 = (u,v)$ and the final output is either $r_n = (\varepsilon,\varepsilon)$ or a minimal rule $r_n = (u',v')$. Let $r_0, r_1, r_2, \ldots, r_n$ be the sequence of such inputs and outputs in the minimization of (u,v). By considering each step of minimization in turn, we will show that for each i, $1 \leq i \leq n$, if there is a time-s Thue path between the two sides of r_i with maximum node no bigger than either side of r_i, then there is a time-s Thue path between the two sides of r_{i-1} with maximum node no bigger than either side of r_{i-1}. We then obtain the desired time-s Thue path between u and v by using descending induction on i. This is a subsidiary induction to our main induction on s. The base case $i = n$ is true, since at time s the rule r_n has been installed in S.

To make the task of checking the proof easier, we use the same numbering and notation here as in Definition 5.7.

1. At the end of the current step, there is a sequence of elementary reductions from $u_1 \ldots u_{p-1}$ to u', but this may not constitute a Thue path since some of the associated rules may be in Delete. However, any such rule $(\; ; \;)$ in Delete will, at some time $s' < s$, have been in S but not in Delete. Therefore, by our induction on s, at time $s-1$ there is a Thue path p from to with maximum node . Now $u_1 \ldots u_{p-1} < u$ and so is smaller than the left-hand side of r_0. Therefore r_0 cannot be used in p. So p continues to be a Thue path at time s. This completes the downward induction step on i in this case.

2. This step is analogous to the previous step.

3. The sequence of Aut-reductions of u to the current left-hand side does not use the rule r_0 and so the required Thue path exists by induction on s.

4. Let v' be the Aut-reduction of v. Immediately after this step there is a Thue path from v to v' with maximum node v which does not use r_0. By the induction hypothesis on s, there is such a Thue path at time $s-1$. Since it does not use r_0, it continues to be a Thue path at time s. Hence a time-s Thue path from u to v' with maximum node either u or v' yields a time-s Thue path from u to v with maximum node u or v. (Recall that, because of previous steps which may shorten u, u may be smaller than v at this point.) This completes the downward induction step on i in this case.

5. If there is a Thue path from u to v with maximum node either u or v, then the reverse of this path is a Thue path from v to u.

6. Suppose that the input to this step is $(u'x; v)$. Then the output is either the same as the input or is equal to $(u'; v\colon (x))$, with $u' > v\colon (x)$. In the first case there is nothing to prove. In the latter case, we have by our downward induction on i a time-s Thue path from u' to $v\colon (x)$ with maximum node u'. This will give a time-s Thue path from $u'x$ to $v\colon (x)x$ with maximum node $u'x$. Furthermore, at the beginning of the Knuth–Bendix process, there was a Thue path of length one from $(x)x$ to with maximum node equal to $(x)x$. Therefore, by our induction hypothesis, there is such a path at time $s-1$, just before possible deletion of r_0. Now $u'x > v\colon (x)x \quad (x)x$. So the time-$(s-1)$ Thue path from $(x)x$ to cannot use r_0, and it remains a Thue path at time s. It follows that there is a Thue path from $u'x$ to v with maximum node $u'x$ at time s.

7. This step is analogous to the previous step.

8. If the input to this step is $(xu^0;xv^0)$ then the output is $(u^0;v^0)$. A time-s Thue path from u^0 to v^0 with maximum node u^0 yields a time-s Thue path from xu^0 to xv^0 with maximum node xu^0.

9. This step is analogous to the previous step.

This completes the induction on s for the special case where $r_0 = (u;v)$ is a rule being input to the minimization routine (see De nition 5.7) at time t, and s is the time at the end of the subsequent invocation of the minimization handling routine 5.8. Now consider the general case, again assuming the induction statement true at time s 1. The only reason why a Thue path at time s 1 between u and v will not work at time s is if some elementary reduction used in this path has an associated rule (;) in S(s 1) which is deleted at time s. Since deletion only takes place as a result of minimization, we know that what must be happening is that we are right at the end of minimizing (;), with minimization completing exactly at time s. But the special case already proved shows that there is a time-s Thue path between and with no node bigger than . Therefore the time-(s 1) Thue path can always be replaced by a time-s Thue path without increasing the maximum node.

6.6 Lemma. If a word is S(t)-reducible, it is S(s)-reducible for all $s > t$.

Proof: If u is S(t)-reducible, there is an elementary S(t)-reduction $u\ !_{S(t)}\ v$. This means that $v < u$. By Proposition 6.5, for each time $s > t$, there is a Thue path from u to v with maximum node u. The rst elementary reduction in this path has the form u ! w at time s. This proves the result.

6.7 Lemma. At any time t, S(t) is a list of rules which contains no duplicates. If a rule is deleted from S, it will never be re-inserted. (Here we mean actual deletion, not just placing the rule on the Delete list for future deletion.)

Proof: The rst statement follows by looking through 5.6 and checking where insertions of rules take place. We always take care not to insert a rule a second time if it is already present.

Let (;) be a rule which is deleted at time s. We assume by contradiction that it is re-inserted at a later time t. We choose m and n so that time s occurs during the m-th Knuth–Bendix pass and time t during the n-th. Then m n.

We note that all proper subwords of are Aut-irreducible during the m-th pass. For otherwise 5.14 Deleting rules theorem 5.14 shows that is Aut-reducible during the n-th pass. But no rule with left-hand side could then be introduced during the n-th pass, a contradiction.

It follows that we are in Case 5.11.1. Therefore $(;)$ was input to the minimization procedure during the m-th pass and was then moved to Delete. The actual deletion took place at the end of the m-th pass. It follows that $n > m$. The output from the minimization procedure was a rule $(;)$, where is a subword of . The rule $(;)$ is welded into WDi and is accepted by Rules$[m + 1]$. As in the preceding paragraph, we see that cannot be a proper subword of , and so $=$ and $<$. We write $_{m\ 1} =$ and $_{m} =$.

Proceeding in this way, we see that between times s and t, rules of the form $(; _{i\ 1})$ $(m \quad i \quad n)$ are input to the minimization procedure during the i-th Knuth–Bendix pass, with output $(; _{i})$ where $_{i} \quad _{i\ 1}$ and $_{m} <$ $_{m\ 1}$. The rule $(; _{i})$ is produced during the i-th Knuth–Bendix pass and is accepted by Rules$[i + 1]$ for $m \quad i \quad n$.

It follows that is Aut-reducible during the n-th pass. Therefore no rule with left-hand side could be introduced into S as a result of critical pair analysis. We see from 5.10 that any rule with left-hand side equal to which is introduced into S as a result of Aut-reduction during the n-th pass must be of the form $(;)$, where $\quad _{n} <$. This completes the proof of the contradiction.

6.8 Definition. We say that a word u is permanently irreducible if there are arbitrarily large times t for which u is $S(t)$-irreducible. By Lemma 6.6 this is equivalent to saying that u is $S(t)$-irreducible at all times $t \quad 0$. A rule $(;)$ in S is said to be permanent if and every proper subword of is permanently irreducible. 2

6.9 Lemma. A permanently irreducible word is permanently Aut-irreducible. A permanent rule of S is never deleted. A permanent rule is accepted by Rules$[n + 1]$ provided it is present in S when the n-th Knuth–Bendix pass begins; it is then accepted by Rules$[m]$ for all $m > n$.

Proof: Let u be permanently irreducible. Aut-reduction of u can only take place if, immediately after the Aut-reduction, u is S-reducible, conceivably as a result of some rule being added to S during the Aut-reduction. But this is impossible by hypothesis.

A rule (;) is deleted only as a result of being the input to the minimization procedure. By Lemma 6.5, there would have to be a Thue path from to with largest node . The rst elementary reduction must therefore be rightward (see De nition 6.3) $!_{S(t)}$. We are assuming that (;) is a permanent rule of S. Since every proper subword of is permanently irreducible, it is permanently Aut-irreducible, as we have just seen. So this rst elementary reduction must be associated to a rule (;).

Either = , in which case the rule (;) has not been deleted, or else, when (;) was input to the minimization routine, was Aut-reducible. However, it is permanently Aut-irreducible which is a contradiction.

It follows that if (;) is present in S at the start of the n-th Knuth–Bendix pass, it will be sewn into WDi at some point during the n-th Knuth–Bendix pass and accepted by Rules[n + 1]. Since (;) is a permanent rule, it will subsequently remain in S and will be presented for minimization during each pass. The same rule will be output and used to mark states and arrows of WDi as needed. Therefore, (;) is accepted by Rules[m] for each m n.

6.10 Lemma. Let u be a xed word. Then there is a t_0 depending on u, such that, for all t t_0, each elementary S(t)-reduction of u is associated to a permanent rule. If all proper subwords of u are permanently irreducible, then, for t t_0, there is at most one elementary reduction of u, and this is associated to a permanent rule (u;w).

Proof: There are only nitely many subwords of u. So we need only prove that, given any word v, there is a t_0 such that for all t t_0, each rule in S(t) with left-hand side v is permanent. If there is a proper subword of v which is not permanently irreducible, then at some time s_0 it becomes $S(s_0)$-reducible. By Lemma 6.6, it is S(s)-reducible for s s_0. By Lemma 5.14, it becomes Aut-reducible at the beginning of the next Knuth–Bendix pass after s_0. During this pass all rules with left-hand side v will be deleted. Also, since this proper subword of v is now permanently Aut-reducible, no rule with left-hand side equal to v will ever be inserted subsequently. In this case, the result claimed about v is vacuously true.

So we assume that each proper subword of v is permanently irreducible, and that v itself is S-reducible at some time t. A rule (v;w) will be permanent if w is permanently irreducible. Otherwise it will disappear as a result of minimization and, by Lemma 6.7, never reappear. There cannot be two permanent rules $(v;w_1)$ and $(v;w_2)$ with $w_1 > w_2$. For critical pair analysis would produce a new rule $(w_1;w_2)$ during the next Knuth{Bendix pass, and so w_1 would not be permanently irreducible.

6.11 Theorem. Let u be a fixed word in A and let v be the smallest element in its Thue congruence class. Then, for large enough times, there is a chain of elementary reductions from u to v each associated to a permanent rule. After enough time has elapsed, Aut-reduction of u always gives v. (Recall that v is the short-lex representative of $\overline{u}$.)

Proof: We start by proving the first assertion. By hypothesis, we have, for each time t, a time-t Thue path p_t from u to v, and we can suppose that p_t contains no repeated nodes by cutting out part of the path if necessary. The only reason why we couldn't take p_{t+1} to be p_t is if some rule $(\ ; \ )$, used along the Thue path p_t, is deleted at time t. By Lemma 6.5 we can, however, assume that each node of p_{t+1} is either already a node of p_t or is smaller than some node of p_t.

Let h_0 be the largest node on p_0, and suppose that we have already proved the theorem for all pairs u and v which are connected by a Thue path with largest node smaller than h_0. By induction on t, using 6.5Correctness of our Knuth{Bendix Proceduretheorem.6.5, we can assume that h_0 is the largest node on p_t for all time t. If $v = h_0$ then since v is the smallest element in its congruence class, there are no elementary reductions starting from v, and we must have $u = v$ in this case.

By Lemma 6.10, we may assume that t_0 has been chosen with the property that, for all words w h_0 and for all t t_0, all elementary $S(t)$-reductions of w are associated to permanent rules which are accepted by Rules[n] provided n is sufficiently large.

Let $h_0 = {}_t\,{}_t\,{}_t \to_{S(t)} {}_t\,{}_t\,{}_t$ be the rightward elementary reduction of h_0 at time t. Our construction of p_{t+1} from p_t, as in 6.5Correctness of our Knuth{Bendix Proceduretheorem.6.5, makes ${}_{t+1}$ a subword of ${}_t$. The construction also ensures that, if ${}_{t+1} = {}_t$, then ${}_{t+1}$ ${}_t$. The rule $({}_t; {}_t)$ is therefore independent of t for large values of t. Then $({}_t; {}_t)$ is permanent and ${}_t$ is Aut-reducible for large enough t. If $u \neq h_0$, the same argument applies to the unique elementary leftward reduction with source h_0 at time t.

If $h_0 = u$, let $u \to_{S(t)} w$ be the first rightward elementary reduction for large values of t. By our induction hypothesis, there is a Thue path of elementary reductions from w to v, each associated to a permanent rule, and with no node larger than w, and so we have the required Thue path from u to v.

Suppose now that $h_0 \neq u$, so that we get two permanent rules, associated to the leftward and rightward elementary reductions of h_0. If the two elementary reductions are identical, that is, if the two permanent rules are equal and if their left-hand sides occur in the same position in h_0, then p_t contains a repeated node which we are assuming not to be the case. So the two elementary reductions occur in different positions in h_0. Now choose t to be large enough so that the two rules concerned have already been compared in a critical pair analysis in Step 5.6.4.b during some previous Knuth–Bendix pass.

If these two rules have left-hand sides which are disjoint subwords of h_0, then we can interchange their order so as to obtain a Thue path from u to v where all nodes are strictly smaller than h_0 — see Figure 4. The first assertion of the theorem then follows by the induction hypotheses in this particular case.

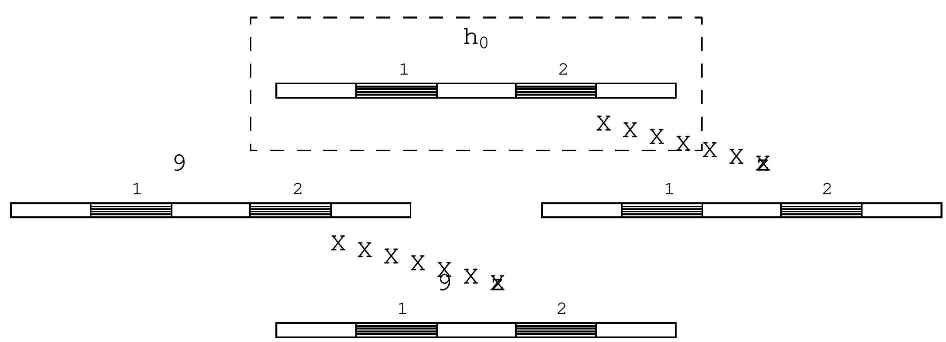

Figure 4. Removing the node h_0 when the leftward and rightward reductions are obtained from rules having disjoint left-hand sides.

If the two left-hand sides do not correspond to disjoint subwords of h_0 then, by assumption, there is some time $t' < t$, such that a critical pair $(u'; v'; w')$ was considered. Here $u' \to_{S(t')} v'$ and $u' \to_{S(t')} w'$ are elementary $S(t')$-reductions given by the two rules, and u' is a subword of h_0. After the critical pair analysis, at time $t'' \leq t$, the Thue paths illustrated in Figure 5 are possible. As a consequence of 6.5Correctness of our Knuth–Bendix Proceduretheorem.6.5, it is straightforward to see that for all times $s \geq t''$, v' and w' can be connected by a time-s Thue path in which all nodes are no larger than the largest of v' and w'. In particular, this applies at time t so that the targets of the two elementary S(t)-reductions from h_0 can be connected by a time-t Thue path in which all nodes are strictly smaller than h_0. This completes the inductive proof of the first assertion of the theorem.

We have arranged that t is large enough so that, for all $w \leq u$, all elementary S(t)-reductions of w are associated to permanent rules, and such a w can be permanently Aut-reduced to the least element in its Thue congruence class. It follows that such a w is Aut-irreducible if and only if it is minimal in its Thue class. In particular Aut-reduction of u must give v.

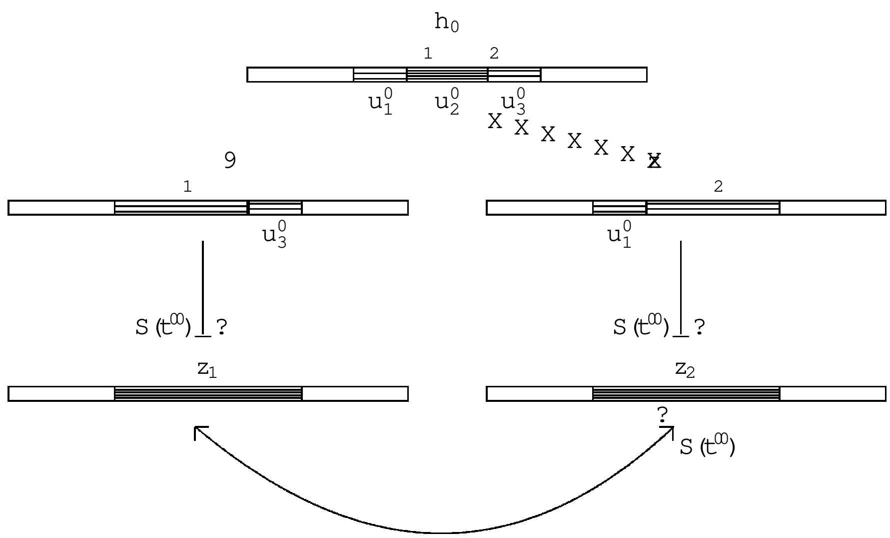

Figure 5. When the leftward and rightward reductions from h_0 are obtained from rules $(\ _1;\ _1)$ and $(\ _2;\ _2)$ having overlapping left-hand sides, this diagram shows the time-t'' Thue paths that exist after the resulting critical pair analysis.

6.12 Corollary. (i) The set of permanent rules in Aut is confluent. (ii) The set of such rules is equal to $P = \bigcap_t \bigcup_{s \geq t} S(s)$. (iii) A word u is smallest in its Thue congruence class if and only if it is permanently irreducible and this is equivalent to being in short-lex normal form. (iv) Each permanent rule is a U-minimal rule and each U-minimal rule is accepted by Rules[n] for n sufficiently large.

Proof: The first and third statements are obvious from Theorem 6.11. For the second statement, each permanent rule is contained in P by Lemma 6.9.

Conversely, if we have a rule r in S which is not permanent, then for all su ciently large times s either its right-hand side or a proper subword of its left-hand side is S(s)-reducible. Theorem 6.11 ensures that this reducible word is Aut-reducible for all su ciently large times s. Therefore r will be minimized and deleted from S. Hence from Lemma 6.7 we see that r is not contained in P.

To prove the fourth statement, suppose (;) is U-minimal. By 6.11Correctness of our Knuth{Bendix Proceduretheorem.6.11, a Thue path from to will eventually be generated by our Knuth{Bendix procedure and each elementary reduction in the path will be rightward and associated to a permanent rule. The rst elementary reduction must have the form (; ^{0}), because each proper subword of is permanently irreducible. But then ^{0} = , for otherwise ^{0} > and 6.11Correctness of our Knuth{Bendix Proceduretheorem.6.11 applies to show that ^{0} is not permanently irreducible. But then (; ^{0}) would not have been a permanent rule. Therefore (;) is a permanent rule.

Conversely, suppose that (;) is a permanent rule. This means that and every proper subword of is permanently irreducible. By 6.11Correctness of our Knuth{Bendix Proceduretheorem.6.11, this mens that and every proper subword of are in short-lex normal form. It follows that (;) is U-minimal.

The next result is the main theorem of this paper.

6.13 Theorem. [Theorem] Let G be a group with a given nite presentation and a given ordering of the generators and their inverses. Suppose that the set of U-minimal rules is regular (for example if (G;A) is short-lex-automatic). Then the procedure given in 5.6 will stabilize at some n_0 with Rules[n + 1] = Rules[n] if n n_0. P (de ned in 6.12Correctness of our Knuth{Bendix Proceduretheorem.6.12) is then the language of a certain two-variable nite state automaton and the automaton can be explicitly constructed. (Unfortunately we do not have a method of knowing when or whether we have reached n_0.)

Proof: By hypothesis there is a two-variable automaton accepting the set of all U-minimal rules. By welding, we obtain a two-variable rule automaton M . By amalgamating states, we may assume that each state of M corresponds to a di erent word-di erence.

Given any arrow in M , there is a U-minimal rule (;) which is accepted by M and which uses that arrow. By 6.12Correctness of our Knuth{Bendix

Proceduretheorem .6.12. (;) is a permanent rule which is eventually generated by our Knuth{Bendix procedure. By 6.9Correctness of our Knuth{Bendix Proceduretheorem .6.9, such a rule is never deleted. Since there are only a nite number of arrows in M , we see that, for large enough n, each (;) in this nite set of rules may be traced out in Rules[n]. We record the states and arrows reached as being required by this nite set of rules.

We may also assume that the states in Rules[n] which have been recorded as just explained, are all associated to di erent word-di erences. To see this, rst note that any equality of word-di erences between di erent states is eventually discovered according to 6.11Correctness of our Knuth{Bendix Proceduretheorem .6.11. Then, as in 5.9, the corresponding states are amalgamated. It follows that, for n large enough, there is a copy of M inside Rules[n].

Subsequently, arrows and states lying outside M will not be used in Autreduction. They will not be marked as needed and will be deleted. It follows that Rules[n] = M for n su ciently large.

Finally, knowing M , we can easily change it to a nite state automaton accepting exactly the minimal rules— this involves making sure that if $(u;v)$ is accepted, then $u > v$, v is irreducible and every proper subword of u is irreducible.

7 Fast reduction

[Section]

In this section, we show how to rapidly reduce an arbitrary word, using the rules in Set(Rules) together with the rules in S. We assume the properties made explicit in 5.1. The time taken to carry out the rst reduction is bounded by a small constant times the length of the word. This e ciency is possible because of the use of nite state automata to do the reduction.

7.1 **Rules for which no pre x or su x is a rule.** At the moment, it is possible for an element $(u;v)^+$ of Set(Rules) to have a pre x or su x which is also a rule. This is undesirable because it makes the computations we will have to do bigger and longer without any compensating gain.

Recall that the automaton recognizing Set(Rules) is the product of Rules with SL2, the initial state being the product of initial states and the set of nal states being any product of nal states. By 5.1, there is only one initial and one nal state of Rules; these are equal and the state is denoted by s_0.

We remove from Rules any arrow labelled $(x;x)$ from the initial state to itself. We then form the product automaton, as described above, with two restrictions. Firstly, we omit any arrow whose source is a product of final states. Secondly, we omit the state with first component equal to s_0, the initial state of Rules, and second component equal to state 3 of SL2 (see Figure 3) and any arrow whose source or target is this omitted state. We call the resulting automaton Rules$'$.

7.2 Lemma. The language accepted by Rules$'$ is the set of labels of accepted paths in the product automaton, starting from the product of initial states and ending at a product of final states, such that the only states along the path with first component equal to s_0 are at the beginning and end of the path.

Proof: First consider an accepted path in Rules$'$. The only arrows in Rules$'$ with source having first component s_0 are those with source the product of initial states. In SL2 it is not possible to return to the initial state. It follows that has the required form.

Conversely any such path in the product automaton also lies in Rules$'$ because it avoids all omitted arrows.

7.3 Lemma. The language accepted by Rules$'$ is the subset of Set(Rules) which has no proper suffix or proper prefix in Set(Rules).

Proof: If is an accepted path in Rules$'$, then it is clearly in Set(Rules). Moreover if it had a proper suffix or proper prefix which was in Set(Rules), there would be a state in the middle of with first component s_0. We have seen that this is impossible in Lemma 7.2.

Conversely, we must show that if is an accepted path in the product automaton such that no proper prefix and no proper suffix of would be accepted by the product automaton, then no state met by , apart from its two ends, has s_0 as a first component. Let $= ((s_0,1),(u_1,v_1),q_1,\dots,(u_n,v_n),q_n)$,

First suppose $u_1 < v_1$. Since is accepted by SL2, $|u| > |v|$ and we must have $v_n = \$$. Let $r < n$ be chosen as large as possible so that the first component of q_r is s_0. Then $(u_{r+1},v_{r+1})\dots(u_n,v_n)$ will be accepted by Rules and will be accepted by SL2 because $v_n = \$$. Since this cannot be a proper suffix of by assumption, we must have $r = 0$. Hence q_i has a first component equal to s_0 if and only if $i = 0$ or $i = n$.

Next note that we cannot have $u_1 = v_1$. This is because there is no arrow labelled (u_1,u_1) in SL2 with source the initial state, so would not be accepted by the product automaton.

Now suppose that $u_1 > v_1$ and let $r > 0$ be chosen as small as possible so that the first component of q_r is s_0. Since $u_1 > v_1$, the second component of

q_r will be a final state (see Figure 3). Since σ has no accepted proper prefix, we must have $r = n$. Hence q_i has a first component equal to s_0 if and only if $i = 0$ or $i = n$.

So we have proved the required result for each of the three possibilities.

Reduction with respect to Set(Rules) is done in a number of steps. First we find the shortest reducible prefix of w, if this exists. Then we find the shortest suffix of that which is reducible. This is a left-hand side of some rule in Set(Rules). Then we find the corresponding right-hand side and substitute this for the left-hand side which we have found in w. This reduces w in the short-lex-order. We then repeat the operation until we obtain an irreducible word. The process is explained in more detail in 7.14.

Our first objective is to find the shortest reducible prefix of w, if this exists. To achieve this, we must determine whether w contains a subword which is the left-hand side of rule belonging to Set(Rules).

Let Rules'' be the automaton obtained from Rules' (see Lemmas 7.2 and 7.3) by adding arrows labelled $(x;x)$ from the initial state to the initial state.

We construct an FSA $\mathrm{Rble}_N(\mathrm{Rules})$ in one variable by replacing each label of the form $(x;y)$ on an arrow of Rules'' by x. Here $x \in A$ and $y \in A^+$. The name of the automaton $\mathrm{Rble}_N(\mathrm{Rules})$ refers to the fact that the automaton accepts reducible words, and does so non-deterministically. We obtain an FSA with no ε-arrows. However there may be many arrows labelled x with a given source. Let LHS(Rules) be the regular language of left-hand sides of rules in Set(Rules) such that no proper prefix or proper suffix of the rule is itself a rule.

7.4 Lemma. $A^*\mathrm{LHS}(\mathrm{Rules}) = L(\mathrm{Rble}_N(\mathrm{Rules}))$.

Proof: Because of the extra arrows labelled $(x;x)$ from initial state to initial state, inserted into Rules'', the inclusion $A^*\mathrm{LHS}(\mathrm{Rules}) \subseteq L(\mathrm{Rble}_N(\mathrm{Rules}))$ is clear.

Conversely, if u is accepted by $\mathrm{Rble}_N(\mathrm{Rules})$, there is a corresponding pair $(u;v)$ accepted by Rules''. We find a maximal common prefix p of u and v, so that $u = pu'$ and $v = pv'$. Rules'' remains in the initial state while reading $(p;p)$. Since the initial state of SL2 is not a final state, $(u';v')$ must be non-empty. Since there is no way of returning to the initial state of SL2, once Rules'' starts reading $(u';v')$, it can never return to the initial state, and therefore $(u';v')$ must be accepted by Rules'. Therefore $u' \in \mathrm{LHS}(\mathrm{Rules})$, as claimed.

7.5 The automaton P. To find the shortest reducible prefix of a given word w we could feed w into the FSA $\mathrm{Rble}_N(\mathrm{Rules})$. However, reading a word with a non-deterministic automaton is very time-consuming, as all possible alternative paths need to be followed.

For this reason, it may at first sight seem sensible to determinize the automaton. However, determinizing a non-deterministic automaton potentially leads to an exponential increase in size. The states of the determinized automaton are subsets of the non-deterministic automaton, and there are potentially 2^n of them if there were n states in the non-deterministic automaton.

For this reason, we use a lazy state-evaluation form of the subset construction. The lazy evaluation strategy (common in compiler design—see for example [1]) calculates the arrows and subsets as and when they are needed, so that a gradually increasing portion $P(\mathrm{Rules})$ of a determinized version $\mathrm{Rble}_D(\mathrm{Rules})$ of $\mathrm{Rble}_N(\mathrm{Rules})$ is all that exists at any particular time.

Lazy evaluation is not automatically an advantage. For example, if in the end one has to construct virtually the whole determinized automaton $\mathrm{Rble}_D(\mathrm{Rules})$ in any case, then nothing would be lost by doing this immediately. In our special situation, lazy evaluation is an advantage for two reasons. First, during a single pass of the Knuth–Bendix process (see 4.7), only a comparatively small part of the determinized one-variable automaton $\mathrm{Rble}_D(\mathrm{Rules})$ needs to be constructed. In practice, this phenomenon is particularly marked in the early stages of the computation, when the automata are far from being the "right" ones. Second, this approach gives us the opportunity to abort a pass of Knuth–Bendix, recalculate on the basis of what has been discovered so far in this pass, and then restart the pass. If an abort seems advantageous early in the pass, very little work will have been done in making the structure of a determinized version of $\mathrm{Rble}_D(\mathrm{Rules})$ explicit.

At the start of a Knuth–Bendix pass we let $P(\mathrm{Rules})$ be the one-variable automaton containing only one state and no arrows. The state is an initial state of $P(\mathrm{Rules})$ which is a singleton set whose only element is the ordered pair of initial states of Rules and $SL2$. At a subsequent time during the pass, $P(\mathrm{Rules})$ may have increased, but it will always be a portion of $\mathrm{Rble}_D(\mathrm{Rules})$. Each state of $P(\mathrm{Rules})$ is a set of pairs (s,t), where s is a state of Rules and t is a state of $SL2$.

The transition with source s, a state in $P(\mathrm{Rules})$, and label $x \in A$ may or may not already be defined. If it is defined, we denote by $\delta(s,x)$ the target of this arrow.

Suppose now that we wish to find the shortest prefix of the word $w = x_1 \cdots x_n \in A^*$ which is Set(Rules)-reducible. Suppose that $s_0, s_1, \ldots, s_k$ are states of $P(\mathrm{Rules})$, where $0 \le k \le n-1$, that s_0 is the start state of

$P(\text{Rules})$, and that, for each i with $1 \le i \le k$, the arrow with source s_{i-1} and label x_i has been constructed, with target $(s_{i-1};x_i) = s_i$. Suppose that the target of the arrow with source s_k and label x_{k+1} has not yet been defined.

The conventional subset construction applied to the state s_k of $P(\text{Rules})$ under the alphabet symbol x_{k+1} yields a set, which we denote by $_1(s_k;x_{k+1})$. This is how $_1(s_k;x_{k+1})$ is defined. For each $(s^0;t^0) \in s_k$, we look for all arrows in $\text{Rble}_N(\text{Rules})$ labelled x_{k+1} with source $(s^0;t^0)$. If $(s;t)$ is the target of such an arrow, then $(s;t)$ is an element of $_1(s_k;x_{k+1})$. Note that this subset is always non-empty, because the initial state of $\text{Rble}_N(\text{Rules})$ is an element of each s_i.

In the standard determinization procedure one would now look to see whether there is already a state s_{k+1} of $P(\text{Rules})$ which is equal to $_1(s_k;x_{k+1})$. If not, one would create such a state s_{k+1}. One would then insert an arrow labelled x_{i+1} from s_k to s_{k+1}, if there wasn't already such an arrow. A new state is defined to be a final state of $P(\text{Rules})$ if and only if the subset contains a final state of $\text{Rble}_N(\text{Rules})$. Of course, one does not need to determine the subset $_1(s_k;x_{k+1})$ if there is already an arrow in $P(\text{Rules})$ labelled x_{k+1} with source s_k, because in that case the subset is already computed and stored.

In our procedure we improve on the procedure just described. The point is that $_1(s_k;x_{k+1})$ may contain pairs which are not needed and can be removed. From a practical point of view this has the advantage of saving space and reducing the amount of computation involved when calculating subsequent arrows. Specifically, we remove a pair $(p;q^0)$ from $_1(s_k;x_{k+1})$ if q^0 is state 3 of $SL2$ (see Figure 3) and $_1(s_k;x_{k+1})$ also contains the pair $(p;q)$ where q is state 2 of $SL2$ (same p as in $(p;q^0)$) Removing all such pairs $(p;q^0)$ yields the set $_P(s_k;x_{k+1})$ and we add the corresponding arrow and state to $P(\text{Rules})$, creating a new state if necessary. We make the state a final state if the subset contains a final state of $\text{Rble}_N(\text{Rules})$. The validity of this modification follows from Theorem 8.2, and we see that some prefix of w arrives at a final state of $P(\text{Rules})$ if and only if w is $\text{Set}(\text{Rules})$-reducible.

When finding the corresponding left-hand side of a rule inside w, we need never compute beyond a final state of $P(\text{Rules})$. As a space-saving and time-saving measure our implementation therefore replaces each final state of $P(\text{Rules})$, as soon as it is found, by the empty set of states. As remarked above, the standard determinization of $\text{Rble}_N(\text{Rules})$ never produces an empty set of states, so there is no possibility of confusion.

Reading w can be quite slow if many states need to be added to $P(\text{Rules})$ while it is being read. However, reading w is fast when no states need to be built. In practice, fairly soon after a Knuth–Bendix pass starts, reading becomes rapid, that is, linear with a very small constant.

7.6 Finding the left-hand side in a word. We retain the hypotheses of Section 7. Namely, we have a two-variable automaton Rules satisfying the conditions of Paragraph 5.1. We are given a word $w = x_1 \cdots x_n$, and we wish to reduce it. In the previous section we showed how to find the minimal reducible prefix $w' = x_1 \cdots x_m$ of w with respect to the rules implicitly specified by Rules. We now wish to find the minimal suffix of w' which is a left-hand side of some rule in Set(Rules). The procedure is quite similar to that of the previous section.

We will now give the basic construction. However, the details will later need to be modified so as to achieve greater computational efficiency in finding the associated right-hand side, if this is necessary. Our reason for including the simpler version is to lead the reader more gently and with more understanding to the actual more complex version.

We form the two-variable automaton Rev(Rules), which we combine with Rev(SL2). The first automaton is, by hypothesis, partially deterministic. If we determinize the second automaton, we obtain another PDFA. Figure 6 shows the determinization of Rev(SL2), where the subsets of states of SL2 are explicitly recorded.

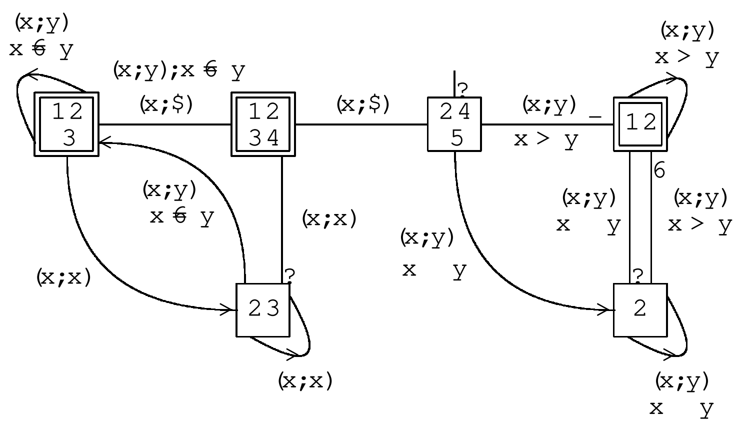

Figure 6. This PDFA arises by applying the accessible subset construction to Rev(SL2) in the case where the base alphabet has more than one element. Each state is a subset of the state set of Rev(SL2) and final states have a double border. This PDFA, when reading a pair (u;v) from right to left, keeps track of whether u is longer than v or not, which it discovers immediately since padding symbols if any must occur at the right-hand end of v. Note that this automaton is minimized.

We take the product of the two automata Rev(Rules) and Rev(SL2). A new state is a pair of old states. An arrow is a pair of arrows with the same label (x,y). The initial state in the product is the unique pair of initial states. A final state in the product is a pair of final states.

To form the one-variable non-deterministic automaton $Rev_N(LHS(Rules))$ without ε-arrows, we use the same states and arrows as in the product automaton, but replace each label of the form (x,y) in the product automaton by the label x. The deterministic one-variable automaton $Rev_D(LHS(Rules))$ can then be constructed using the subset construction.

As we have already warned the reader, we use not the construction just described, but a related construction which we describe below. The point of what we do may not become fully apparent until we get to 7.13.

7.7 Reversing the rules. We first describe a two-variable PDFA M which accepts exactly the reverse of each rule $(\ ,\ )^+$ in Set(Rules) such that no proper suffix and no proper prefix of $(\ ,\ )^+$ is in Set(Rules) (cf. Lemma 7.3). We assume that we have a two-variable automaton Rules satisfying the conditions of Paragraph 5.1.

A state of M is a triple (s,i,j), where s is a state of Rev(Rules), $i \in \{0,1,2\}$ and $j \in \{+,-\}$. The intention is that in a state (s,i,j), i represents the number of padded symbols occurring in any path of arrows from the initial state of M to (s,i,j). By 5.3, the padded symbols must be of the form $(x,\$)$, where $x \in A$. There are zero, one or two padded symbols in any rule, and, if padded symbols appear, they are at the right-hand end of a rule. This means that they are the first symbols read by M. The j component is intended to represent whether an arrow is permitted with source (s,i,j) and label a padded symbol. We take $j = +$ if a padded symbol is permitted, and $j = -$ if a padded symbol is not permitted.

M has a unique initial state $(s_0,0,+)$ where s_0 is the unique initial state of Rev(Rules). In addition, M has three final states $f_0 = (s_0,0,-)$, $f_1 = (s_0,1,-)$ and $f_2 = (s_0,2,-)$. We do not allow states of M of the form (s_0,i,j), except for the initial state and the three final states just mentioned. We will construct the arrows of M to ensure that any path of arrows accepted by M has first component equal to s_0 for its initial state and its final state and for no other states. (Compare this with Lemma 7.2.)

The following conditions determine the arrows in M.

1. Each arrow of M is labelled with some (x,y), where $x \in A$ and $y \in A^+$.
2. $(s,i,j)^{(x,\$)}$ is defined if and only if 1) $t = s^{(x,\$)}$ is defined in Rev(Rules), and 2a) $(s,i,j) = (s_0,0,+)$, the initial state, or 2b) $(i,j) = (1,+)$. In case 2a) the target is $(t,1,+)$, unless t is the final state of Rev(Rules), in which case the target is $f_1 = (s_0,1,-)$. In case 2b), the target is

$(t,2, \ )$, which may possibly be equal to f_2. The final state f_1 arises in case 2a) when we have a rule $(x, \ )$, which means that the generator x of our group represents the trivial element. The final state f_2 arises in case 2b) when we have a rule $(x_1x_2, \ )$. This kind of rule arises when x_1 and x_2 are inverse to each other, usually formal inverses.

3. For $i = 0,1,2$, there are no arrows with source f_i.

4. Suppose (s,i,j) is not a final state. Then $(s,i,j)^{(x,y)}$ with $x,y \in A$ is defined if and only if 1) $t = s^{(x,y)}$ is defined in Rev(Rules), and 2) if $t = s_0$ then 2a) $i = 0$ and $x > y$ or 2b) $i > 0$ and $x \neq y$. We then have $(s,i,j)^{(x,y)} = (t,i, \ )$. This condition corresponds to the requirement that (u,v) can only be a rule if a) u and v have the same length and $u_1 > v_1$, where these are the first letters of u and v respectively, or b) if u is longer than v and $u_1 \neq v_1$.

7.8 Lemma. The language accepted by M is the set of reversals of rules $(\ , \ )^+ \in$ Set(Rules) such that no proper suffix and no proper prefix of $(\ , \ )^+$ is in Set(Rules).

The proof of this lemma is much the same as the proofs of Lemmas 7.2 and 7.3. We therefore omit it.

Using the above description of M, we now describe how to obtain a non-deterministic one-variable automaton Rev_N(LHS(Rules)) from M in an analogous manner to that used to obtain Rble_N(Rules) from Rules$^{(2)}$ in Section 7. Rev_N(LHS(Rules)) accepts reversed left-hand sides of rules in Set(Rules) which do not have a proper prefix or a proper suffix which is in Set(Rules). Rev_N(LHS(Rules)) has the same set of states as M and the same set of arrows. However, the label (x,y) with $x \in A$ and $y \in A^+$ of an arrow in M is replaced by the label x in Rev_N(LHS(Rules)) The two automata, M and Rev_N(LHS(Rules)), have the same initial state and the same final states. Hence Rev_N(LHS(Rules)) accepts all reversed left-hand sides ^{R} of rules $(\ , \ )$ whose reversals $((\ , \ )^+)^R$ are accepted by M.

7.9 The automaton Q. The one-variable automaton Q(Rules) is formed from Rev_N(LHS(Rules)) by a modified subset construction, using lazy evaluation. Q(Rules) is part of the one-variable PDFA Rev_D(LHS(Rules)), the determinization of Rev_N(LHS(Rules)). As we shall see, a word is accepted by Q(Rules) only if its reversal is the left-hand side of a rule in Set(Rules) and no proper subword of has this property.

7.10 Note. In order to construct states and arrows in $Q(\mathrm{Rules})$, one only needs to have access to $\mathrm{Rev}(\mathrm{Rules})$, that is, neither M nor $\mathrm{Rev}_N(\mathrm{LHS}(\mathrm{Rules}))$ has to be explicitly constructed. $\square$

7.11 The algorithm for finding the left-hand side. Suppose we have a word $x_1 \cdots x_n \in A^*$ and we know it has a suffix which is the left-hand side of some rule in $\mathrm{Set}(\mathrm{Rules})$. Suppose no proper prefix of $x_1 \cdots x_n$ has this property. We give an algorithm that finds the shortest such suffix.

We read the word from right to left, starting with x_n. We assume that $x_{k+1}x_{k+2} \cdots x_n$ has been read so far and that as a result the current state of $Q(\mathrm{Rules})$ is S_k, where S_k is a state of $Q(\mathrm{Rules})$ (so S_k is a subset of the set of states of $\mathrm{Rev}_N(\mathrm{LHS}(\mathrm{Rules}))$).

We start the algorithm with $k = n$ and the current state of $Q(\mathrm{Rules})$ equal to the singleton $\{(s_0; 0; +)\}$ whose only element is the initial state of M, where s_0 is the initial state of $\mathrm{Rev}(\mathrm{Rules})$. $Q(\mathrm{Rules})$ has three final states, namely the singleton sets $\{f_i\}$ for $i = 0; 1; 2$.

The steps of the algorithm are as follows:

1. Record the current state as the k-th entry in an array of size n, where n is the length of the input word.

2. If the current state is not a final state, go to Step 7.11.3. If the current state is a final state, then stop. Note that the initial state of $Q(\mathrm{Rules})$ is not a final state, so this step does not apply at the beginning of the algorithm. If the current state is a final state, then the shortest suffix of $x_1 \cdots x_n$ which is the left-hand side of a rule in $\mathrm{Set}(\mathrm{Rules})$ can then be proved to be $x_{k+1}x_{k+2} \cdots x_n$.

3. If the arrow labelled x_k with source the current state is already defined, then redefine the current state to be the target of this arrow and decrease k by one.

4. If the preceding step does not apply, we have to compute the target T of the arrow labelled x_k with source the current state S_k. We do this by looking for all arrows labelled x_k in $\mathrm{Rev}_N(\mathrm{LHS}(\mathrm{Rules}))$ with source in S_k. We define T to be the set of all targets of such arrows. Note that this set of targets cannot be empty since we know that some suffix of $x_1 \cdots x_n$ is accepted by $\mathrm{Rev}_N(\mathrm{LHS}(\mathrm{Rules}))$.

5. There are two modifications which we can make to the previous step.

   (a) Firstly, if the set of targets contains some final state f_j, then we look for the largest value of $i = 0; 1; 2$ such that $f_i \in T$ and redefine

T to be $\{f_i\}$. We then insert into $Q(Rules)$ an arrow labelled x_k from S_k to this final state. If we have found that T is a final state, we set S_{k-1} equal to T, decrease k by one, and go to Step 7.11.1.

(b) Secondly, if, while calculating the set T, we find that a state s of $Rev(Rules)$ occurs in more than one triple $(s;i;j)$, then we only include the triple with the largest value of i. For this to be well-defined, we need to know that $(s;i;+)$ and $(s;i;-)$ cannot both come up as potential elements of T — this is addressed in the proof of Theorem 7.12 along with justifications of the other modifications.

6. Having found T, see if it is equal to some state T' of $Q(Rules)$ which has already been constructed. If so, define an arrow labelled x_k from S to T'.

7. If T has not already been constructed, define a new state of $Q(Rules)$ equal to T and define an arrow labelled x_k from S to T.

8. Set the current state equal to T and decrease k by one. Then go to Step 7.11.1.

7.12 Theorem. Suppose $x_1 \cdots x_n$ has a suffix which is the left-hand side of a rule in $Set(Rules)$ and suppose no prefix of $x_1 \cdots x_n$ has this property. Then the above algorithm correctly computes the shortest such suffix.

Proof: We first show that the modification in Step 7.11.5.b is well-defined in the sense that triples $(s;i;+)$ and $(s;i;-)$ cannot both occur while calculating T. The reason for this is that the third component can only be $+$ if either none of $x_1 \cdots x_n$ has been read, in which case the only relevant state is $(s_0;0;+)$, or else only x_n has been read, in which case the possible relevant states are $(f;1;-)$, $(s;1;+)$ with $s \neq f$, and $(s;0;-)$. So a state of the form $(s;i;j)$ with a given s occurs at most once in a fixed subset with the maximum possible value of i.

The effect of Step 7.11.5.a in the above algorithm is to ensure that termination occurs as soon as a final state of $Rev(Rules)$ appears in a calculated triple. Since we know that $x_1 \cdots x_n$ contains a left-hand side of a rule in $Set(Rules)$ as a suffix we need only show that the introduction of Step 7.11.5.b does not affect the accepted language of the constructed automaton. This will be a consequence of Theorem 8.2, as we now proceed to show.

Consider a triple $t = (s;i;j)$ arising during the calculation of a subset T, and suppose that s is a non-final state of Rev(Rules). If $j = +$ then T cannot contain both $(s;0;+)$ and $(s;1;+)$ and so t will not be removed from T as a result of Step 7.11.5.b. Therefore we only need to consider the case $j = -$. For $k = 0;1;2$, let $L_k \subseteq A^* \times A^*$ be the language obtained by making $(s;k;-)$ the only initial state of M, and observe that there can be no padded arrows in any path of arrows from $(s;k;-)$ to a final state of M. Now by considering the definition of the non-padded transitions in M given in 7.7.4, it is straightforward to see that $L_0 \subseteq L_1 = L_2$. Therefore, since Rev_N(LHS(Rules)) has no ε-arrows, we have just shown that the hypotheses of Theorem 8.2 apply to Step 7.11.5.b. Hence the omission in Step 7.11.5.b does not affect the accepted language of Q(Rules).

As with P(Rules), reading a word into Q(Rules) from right to left can be slow in the initial stages of a Knuth–Bendix pass, but soon speeds up to being linear with a small constant.

7.13 Finding the right-hand side of a rule. We retain the hypotheses of Section 5.1. Namely, we have a two-variable rule automaton Rules which is welded and satisfies various other minor conditions. We are given a word $w = x_1 \cdots x_n$, and we wish to reduce it relative to the rules implicitly contained in Rules. So far we have located a left-hand side λ which is a subword of w. In this section we show how to construct the corresponding right-hand side.

We first go into more detail as to how we propose to reduce w. In outline we proceed as follows.

7.14 Outline of the reduction process.

1. Feed w one symbol at a time into the one-variable automaton P(Rules) described in Section 7, storing the history of states reached on a stack.

2. If a final state is reached after some prefix u of w has been read by P(Rules), then u has some suffix which is a left-hand side. Moreover, this procedure finds the shortest such prefix.

3. Feed u from right to left into Q(Rules). A final state is reached as soon as Q(Rules) has read the shortest suffix λ of u such that there is a rule $(\lambda;\rho) \in$ Set(Rules). We now have $u = p\lambda$ and $w = p\lambda q$, where $p;q \in A^*$, every proper prefix of $p\lambda$ and every proper suffix of λ is Set(Rules)-irreducible.

4. Find , the smallest word such that there is a rule (;) in S (see 4.7). If there is no such rule in S, nd by a method to be described in 7.15, such that is the smallest word such that (;) 2 Set(Rules).

5. If (;) is not already in S, insert it into the part of S called New.

6. Replace with in w and pop j j levels o the stack so that the stack represents the history as it was immediately after feeding p into P (Rules).

7. Rede ne w to be p q. Restart at Step 1 as though p has just been read and the next letter to be read is the rst letter of . The history stack enables one to do this.

Note that other strategies might lead to nding rst some left-hand side in w other than . Moreover, there may be several di erent right-hand sides with (;) 2 Set(Rules). A rule (;) in Set(Rules) gives rise to paths in Rules, SL2 and Rev_D (SL2). We will nd the path for which right-hand side is short-lex-least, given that the left-hand side is equal to .

Let = y_1 $_m$ .y Recall that a state of the one-variable automaton Q (Rules) used to nd is a set of states of the form (s;i;j), where s is a state of Rules; i 2 f0;1;2g and j 2 f+ ; g. When nding we kept the history of states of Q (Rules) which were visited — see Step 7.11.1. Let Q_k be the set of triples (s;i;j) comprising the state of Q (Rules) after reading the word y_{k+1} $_m$ yfrom right to left. Q_0 = ff$_i$g = f(s_0;i;)g where s_0 is the unique initial and nal state of Rules, and i is the di erence in length between and the that we are looking for.

7.15 Right-hand side routine. Inductively, after reading y_1 $_k$ wye will have determined z_1 $_k$, the pre x of . Inductively we also have a triple (s_k;i_k;j_k), where s is a state of Rules, i_k is 0 or 1 or 2 and j_k is + or . Note that we always have m k i_k.

1. If m k = i_k, then we have found = z_1 $_k$ and we stop. So from now on we assume that m > i_k + k. This means that the next symbol (y_{k+1};z_{k+1}) of (;) does not have a padding symbol in its right-hand component.

2. We now try to nd z_{k+1} by running through each element z 2 A in increasing order. Set z equal to the least element of A.

3. If k = 0 and i_0 = 0, then and will be of equal length, so the rst symbol of (;) must be (y_1;z_1), where y_1 > z_1. So at this stage we

can prove that we have $y_1 > z$, since we know that there must be some right-hand side corresponding to our given left-hand side.

If $k = 0$ and $i_0 > 0$, then the first symbol of $(\; ; \;)^+$ is $(y_1; z_1)$ with $z_1 \in A$ and $y_1 \neq z_1$. If $k = 0$, $i_0 > 0$ and $y_1 = z$, we increase z to the next element of A.

4. Here we are trying out a particular value of z to see whether it allows us to get further. We look in Rules to see if $s_k^{(y_{k+1};z)} = s_{k+1}$ is defined. If it is not defined, we increase z to the next element of A and go to Step 7.15.3.

5. If s_{k+1} is defined in Step 7.15.4, we look in Q_{k+1} for a triple $(s_{k+1}; i_{k+1}; j_{k+1})$ which is the source of an arrow labelled $(y_{k+1}; z)$ in the automaton M, defined in Section 7.6. Note that, by the proof of 7.12The algorithm for finding the left-hand sidetheorem .7.12, Q_{k+1} contains at most one element whose first coordinate is s_{k+1}. As a result, the search can be quick.

6. If $(s_{k+1}; i_{k+1}; j_{k+1})$ is not found in Step 7.15.5, increase z to the next element of A and go to Step 7.15.3.

7. If $(s_{k+1}; i_{k+1}; j_{k+1})$ is found in Step 7.15.5, set $z_{k+1} = z$, increase k and go to Step 7.15.1.

The above algorithm will not hang, because each triple $(s_k; i_k; j_k)$ that we use does come from a path of arrows in M which starts at the initial state of M and ends at the first possible final state of M. Therefore all possible right-hand sides such that $(\; ; \;) \in \mathrm{Set}(\mathrm{Rules})$, are implicitly computed when we record the states of $Q(\mathrm{Rules})$ (see Step 7.11.1). Since i_k does not vary during our search, we will always find the shortest possible , with $| \; | \; | \; |$ being equal to this constant value of i_k. Since we always look for z in increasing order, we are bound to find the lexicographically least .

8 A modified determinization algorithm

[Section]

In this section we discuss a useful modification to the usual determinization algorithm for turning an NFA into a DFA. Let N be an NFA. The usual proof that N can be determinized, is to form a new automaton M each state of which is a subset of the set $S(N)$ of states of N such that is -closed. That is to say, if $s \in \quad S(N)$, then each -arrow with source s also has target in . The initial state of M is the ε-closure of the set of all initial states in N. The effect of an arrow labelled $x \in A$ on is to take each $s \in$, apply x in all possible ways, and then to take the ε-closure of the subset of $S(N)$ so obtained. A final state of M is any subset of $S(N)$ containing a final state of N.

In practice, to find M, we start with the ε-closure of the set of initial states of N and proceed inductively. If we have found a state s of M as a subset of the set of states of N, we fix some $x \in A$, and apply x in all possible ways to all $t \in s$, where t is a state of N. We then follow with ε-arrows to form an ε-closed subset of states of N. This gives us the result of applying x to s. The modification we wish to make to the usual subset construction is now explained and justified.

We will denote by M' the modified version of M thus obtained. M' is a DFA which accepts the same language as M and N, but the structure of M' might be simpler than that of M.

Suppose p is a state of the NFA N. Let N_p be the same automaton as N, except that the only initial state is p. Suppose p and q are distinct states of N and that $L(N_p) \subseteq L(N_q)$. Suppose also that the ε-closure of q does not include p. Under these circumstances, we can modify the subset construction as follows. As before, we start with the ε-closure of the set of initial states of N. We follow the same procedure for defining the arrows and states of M' as for M, except that, whenever we construct a subset containing both p and q, we change the subset by omitting p.

8.1 Required conditions. The situation can be generalized. We suppose that we have a partial order defined on the set of states of N, such that, if $p < q$, then $L(N_p) \subseteq L(N_q)$. We assume that if $p < q$, $p' < q'$ and p' is contained in the ε-closure of q, then $p' = q$.

We follow the same procedure for defining the arrows and states of M' as for M, except that, whenever we construct a subset containing both p and q with $p < q$, we change the subset by omitting p.

8.2 Theorem. Under the above hypotheses, $L(M') = L(N)$.

Proof: Consider a word $w = x_1 \ldots x_n \in A^*$ which is accepted by N via the path of arrows in N

$$(v_0, \ldots, u_1, x_1, v_1, \ldots, v_{n-1}, \ldots, u_n, x_n, v_n, \ldots, u_{n+1}).$$

This means that, for each i with $0 \le i \le n$, there is an x_i-arrow in N from u_i to v_i and u_{i+1} is in the ε-closure of v_i. Moreover v_0 is an initial state and u_{n+1} is a final state.

Our proof will be by induction on i. The i-th statement in the induction is that we have states $s_0, \ldots, s_i$ of M' such that s_0 is the initial state and, for each j with $0 < j < i$, there is an arrow $x_j : s_{j-1} \to s_j$ in M', so that, after reading $x_1 \cdots x_{i-1}$, M' is in state s_{i-1}. Our induction statement also says that we have a path of arrows in N

$$(u_i^i, x_i, v_i^i, \varepsilon, u_{i+1}^i, \ldots, u_n^i, x_n, v_n^i, \varepsilon, u_{n+1}^i),$$

such that $u_i^i \in s_{i-1}$ and u_{n+1}^i is a final state of N.

The induction starts with $i = 1$ and s_0 the initial state of M'. We form s_0 by taking all initial states of N, and taking their ε-closure. If this subset of states of N contains both p and q with $p < q$, then p is omitted from s_0, the initial state of M'. If $u_1 \notin s_0$, then we must have $u_1 = p$, with $q \in s_0$ and $p < q$. So q must be a maximal element of s_0 with respect to the partial order. Now $w \in L(N_p) \subseteq L(N_q)$. It follows that we can take u_1^1 in the ε-closure of q and then define the rest of the path of arrows for the case $i = 1$. Since $q \in s_0$ and u_1^1 is in the ε-closure of q, it is not the case that there is a q' such that $u_1^1 < q' \in s_0$, according to 8.1. So $u_1^1 \in s_0$ (that is, it is not omitted in our construction) and the induction can start.

Now suppose the induction statement is true for i. We prove it for $i + 1$. we have a path of arrows

$$(u_i^i, x_i, v_i^i, \varepsilon, u_{i+1}^i, \ldots, u_n^i, x_n, v_n^i, \varepsilon, u_{n+1}^i),$$

in N such that $u_i^i \in s_{i-1}$ and u_{n+1}^i is a final state of N. We define s_i from s_{i-1} in the manner described above. First we apply x_i in all possible ways to all states in s_{i-1}, obtaining v_i^i as one of the target states, and then take the ε-closure, obtaining u_{i+1}^i as one of the targets of an ε-arrow. Finally, if s_i contains both p and q, with $p < q$ then p is deleted from s_i before s_i becomes a state of M'.

It now follows that either $u_{i+1}^i \in s_i$, or else, for some $p < q$, $u_{i+1}^i = p$, $q \in s_i$ and $p \notin s_i$. In the first case we define $u_j^{i+1} = u_j^i$ and $v_j^{i+1} = v_j^i$ for $j > i$ and the induction step is complete. In the second case, using the fact that $x_{i+1} \cdots x_n \in L(N_p) \subseteq L(N_q)$, we see that we can take u_{i+1}^{i+1} in the ε-closure of q and then define the rest of the path of arrows. Since $q \in s_i$ and u_{i+1}^{i+1} is in the ε-closure of q, 8.1 shows that it is not possible to have $q' \in s_i$ and $u_{i+1}^{i+1} < q'$. Therefore $u_{i+1}^{i+1} \in s_i$. This completes the induction step.

At the end of the induction, M' has read all of w and is in state s_n. We also have the final state $u_{n+1}^{n+1} \in s_n$, so that w is accepted by M'.

Conversely, suppose w is accepted by M'. It follows easily by induction that if M' is in state s_i after reading the prefix $x_1 \cdots x_i$ of w, then each state $u \in s_i$ can be reached from some initial state of N by a sequence of arrows labelled successively $x_1, \ldots, x_i$, possibly interspersed with ϵ-arrows. Now s_n must contain a final state, and so w is accepted by N.

8.3 **Remark.** The practical usage of this theorem clearly depends on having an efficient way of determining when the condition $L(N_p) \subseteq L(N_q)$ is satisfied. In this paper we have seen several examples of such tests which cost virtually nothing to implement but have the potential to save an appreciable amount of both space and time. □

9 Miscellaneous details

In this section we present a number of points which did not seem to fit elsewhere in this paper.

9.1 **Aborting.** It is possible that we come to a situation where the procedure is not noticing that certain words are reducible, even though the necessary information to show that they are reducible is already in some sense known. It is also possible that reduction is being carried out inefficiently, with several steps being necessary, whereas in some sense the necessary information to do the reduction in one step is already known. An indication that our procedure is not proceeding as well as one hoped might be that WDiff is constantly changing, with states being identified and consequent welding, or with new states or arrows being added. In this case it might be advisable to abort the current Knuth–Bendix pass.

To see if abortion is advisable, we can record statistics about how much WDiff has changed since the beginning of a pass. If the changes seem excessive, then the pass is aborted. A convenient place for the program to decide to do this is just before another rule from New is examined at Step 5.6.3.

If an abort is decided upon then all states and arrows of WDiff are marked as needed. At this point the program jumps to Step 5.6.1.

9.2 **Priority rules.** A well-known phenomenon found when using Knuth–Bendix to look for automatic structures, is that rules associated with finding new word differences or new arrows in WDiff should be used more intensively than other rules. Further aspects of the structure are then found more quickly. This is not a theorem — it is observed behaviour seen on examples which happen to have been investigated.

A new rule associated with new word differences or new arrows in WDiff is marked as a priority rule. When a priority rule is minimized, the output is also marked as a priority rule. If a priority rule is added to one of the lists

Considered, Now or New, it is added to the front of the list, whereas rules are normally added to the end of the list. Just before deciding to add a priority rule to New, we check to see if the rule is minimal. If so, we add it to the front of Now instead of to the front of New.

When a rule is taken from Now at Step 5.6.4 during the main loop, it is normally compared with all rules in Considered, looking for overlaps between left-hand sides. In the case of a priority rule, we compare left-hand sides not only with rules in Considered, but also with all rules in Now. If a normal rule (;) is taken from Now and comparison with a rule in Considered gives rise to a priority rule, then the rule (;) is also marked as a priority rule. It is then compared with all rules in Now, once it has been compared with all rules in Considered.

Treating some rules as priority rules makes little difference unless there is a mechanism in place for aborting a Knuth–Bendix pass when WDiff has sufficiently changed. If there is such a mechanism, it can make a big difference.

9.3 An efficiency consideration. During reduction we often have a state s in a two-variable automaton and an $x \in A$, and we are looking for an arrow labelled (x, y) with certain properties, where $y \in A^+$. It therefore makes a big difference if the arrows with source s are arranged so that we have rapid access to arrows labelled (x, y) once x is given.

9.4 The present. Many of the ideas in this paper have been implemented in C++ by the second author. But some of the ideas in this paper only occurred to us while the paper was being written, and the procedures and algorithms presented in this paper seem to us to be substantial improvements on what has been implemented so far. An unfortunate result of this is that we are unable to present experimental data to back up our ideas, although many of our ideas have been explored in depth with actual code. Our experimental work has been essential in enabling us to come to the better algorithms which are presented here.

9.5 Comparison with kbmag. Here we describe the differences between our ideas and the ideas in Derek Holt's kbmag programs [4]. These programs try to compute the short-lex-automatic structure on a group. Our program is a substitute only for the first program in the kbmag suite of programs.

In kbmag, fast reduction is carried out using an automaton with a state for every prefix of every left-hand side. In our program we also keep every rule. However, the space required by a single character in our program is less by a constant multiple than the space required for a state in a finite state automaton. Moreover, compression techniques could be used in our situation so that less space is used, whereas compression is not available in the situation of kbmag.

The other large objects in our set-up are the automata P(Rules[n]) defined in 7.5 and Q(Rules[n]) defined in 7.9. In kbmag, there has also to be an automaton like P(Rules[n]), and it is possible to arrange that this automaton is only constructed after the Knuth–Bendix process is halted. In kbmag there is no analogue of our Q(Rules[n]). So these are advantages of kbmag.

In kbmag, reduction is carried out extremely rapidly. However, as new rules are found, the automaton in kbmag needs to be updated, and this is quite time-consuming. In our situation, updating the automata is quick, but reduction is slower by a factor of around three, because the word has to be read into two or three different automata. Moreover we sometimes need to use the method of Section 7.13 which is slower (by a constant factor) than simply reading a word into a deterministic finite state automaton.

In kbmag, there is a heuristic, which seems to be inevitably arbitrary, for deciding when to stop the Knuth–Bendix process. In our situation there is a sensible heuristic, namely we stop if we find Rules[n + 1] = Rules[n].

In the case of kbmag, there are occasional cases where the process of finding the set of word differences oscillates indefinitely. This is because redundant rules are sometimes unavoidably introduced into the set of rules, introducing unnecessary word differences. Later redundant rules are eliminated and also the corresponding word differences. This oscillation can continue indefinitely. Holt has tackled this problem in his programs by giving the user interactive modes of running them.

In our case, the results in Section 6 show that, given a short-lex-automatic group, the automaton Rules[n] will eventually stabilize, as proved in 6.13 Correctness of our Knuth–Bendix Procedure theorem 6.13, given enough time and space.

We believe that the main advantage of our approach for computing automatic structures will only become evident (if it exists at all) when looking at very large examples. We plan to carry out a systematic examination of short-lex-automatic groups generated by Jeff Weeks' SnapPea program — see [11] — in order to carry out a systematic comparison.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers, Principles, Techniques, and Tools. Addison-Wesley Publishing Company, 1986.

[2] D.B.A. Epstein, J.W. Cannon, D.F. Holt, S.V.F. Levy, M.S. Paterson, and W.P. Thurston. Word Processing in Groups. A.K. Peters, Natick, Mass, 1992.

[3] D.B.A. Epstein, D.F. Holt, and S.E. Rees. The use of Knuth-Bendix methods to solve the word problem in automatic groups. Journal of Symbolic Computation, 12:397–414, 1991.

[4] D.F. Holt. KBMAG (Knuth-Bendix in Monoids and Groups), Version 2. Software package, 1996. Available by anonymous ftp from ftp.maths.warwick.ac.uk in directory people/dfh/kbmag2.

[5] D.F. Holt. The Warwick Automatic Groups Software. In Geometrical and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994), volume 25 of DIMACS Ser. Discrete Math, Theoret. Comput. Sci., pages 69–82. Amer. Math. Soc., Providence RI, 1996.

[6] D.F. Holt and S.E. Rees. Software for automatic groups, isomorphism testing and finitely presented groups. In Geometric group theory, Vol. 1 (Sussex 1991), volume 181 of Londin Math. Soc. Lecture Note Ser., pages 120–125, Cambridge, 1993. Cambridge Univ. Press.

[7] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, Computational problems in abstract algebras, pages 263–297. Pergamon Press, 1970.

[8] C. O'Dunlaing. Infinite regular Thue systems. Theoret. Comput. Sci., 25:171–192, 1983.

[9] M.O. Rabin. Recursive unsolvability of group theoretic problems. Annals of Mathematics, (2)67:172–194, 1958.

[10] Charles C. Sims. Computation with finitely presented groups. Cambridge University Press, 1994.

[11] J.R. Weeks. SnapPea: a computer program for studying hyperbolic 3-manifolds. Freely available from www.geom.umn.edu.

D.B.A. Epstein
Mathematics Institute, University of Warwick
Coventry CV4 7AL, UK
dbae@maths.warwick.ac.uk

P.J. Sanders
Mathematics Institute, University of Warwick
Coventry CV4 7AL, UK
pjs@maths.warwick.ac.uk