Pattern Matching for Noncommutative Gröbner Basis Computations

Benjamin J. Keller
Department of Computer Science, Virginia Tech
Blacksburg, Virginia 24061-0106 USA
keller@cs.vt.edu

June 28, 1996

1 Introduction

We consider pattern matching for a problem from symbolic computation. In particular, we extend the dynamic dictionary matching approach of Amir *et al.* [2] to solve the pattern matching problems involved in the computation of Gröbner bases in noncommutative algebras. The Gröbner basis computation takes a set of polynomials and forms a new set by adding new polynomials. Each new polynomial is the reduced form of a polynomial formed by combining a pair of polynomials whose leading terms has a nontrivial common multiple. Elements of the set may also be deleted when they can be reduced by a new polynomial. The algorithm repeatedly tests whether one polynomial term divides another, and if two terms have a nontrivial common multiple. In the noncommutative case, terms are just words in a free semigroup, and these tests become pattern matching searches with a dictionary of strings. The dictionary D is the set of leading terms of the set of polynomials. The frequency of additions to and deletions from D during the computation requires that these operations take time linear in the size of the input.

A similar situation occurs in the closely related computation of Knuth-Bendix completion for string rewriting (see [4]). There the dictionary is the set of left-hand sides of rewrite rules, but the pattern matching problems are identical. So, in general, we solve the *(string) completion matching* problem. We are given a set D of words that may be changed by adding or deleting words. In addition, D must be such that no element of D is a subword of another element. Given a word w not in D, the following searches are required:

1. *subword search* – find all $d \in D$ such that d is a subword of w;

2. *superword search* – find all $d \in D$ such that w is a subword of d; and

3. *overlap search* – find all overlaps of w with elements of D and with itself.

The string completion problem is an extension of the dynamic dictionary problem considered by Amir *et al.* [2] which only includes the subword search.

Like Amir *et al.*, we use a suffix tree to store our dictionary structure, and so acheive insertion and deletion in time linear in the size of the word being added or deleted [2]. Using slight modifications to the suffix tree (that do not significantly affect insertion or deletion time), all searches take time linear in the sum of the sizes of the input and output. The overlap searches are split into *left*- and *right*-overlap searches in which overlaps of w are found with elements of D on the left and on the right. The right-overlap search also finds overlaps of w with itself and is combined with insertion into D, thus amortizing the search cost; that is, the combination of insertion and the right overlap search is in time linear in the length of the word inserted and the size of the output.

To some extent, our solution could be considered a dynamic implementation of the index automata of Sims [11]. *Index automata* are used for string rewriting to solve similar pattern matching problems, and could be used to solve the string completion problem. However, Sims suggests using the dictionary matching approach of Aho and Corasick [1] to implement index automata. Aho and Corasick use a static data structure to represent the automata, and so insertions and deletions to the dictionary cannot be performed without completely rebuilding the data structure. This makes the index automata unsuitable for use in implementations of completion algorithms. Our approach provides an alternative implementation of index automata for which insertion and deletion take time linear to the size of the word being inserted or deleted rather than to the size of the dictionary. Therefore, our pattern matching approach applies equally well to completion of string rewriting systems. Here, we only consider the application to noncommutative Gröbner bases.

The paper is organized as follows. Section 2 is a brief discussion of noncommutative Gröbner bases and the Gröbner basis algorithm. Section 3 identifies the pattern matching problems involved in the computation of Gröbner bases. Then Section 4 discusses the use of dynamic dictionary matching as a solution to these problems. Section 5 is a brief conclusion.

2 Computing Gröbner Bases

2.1 Gröbner Bases

Gröbner bases are a tool used in computations over polynomial algebras [3]. Here we consider the less typical case where the polynomials are noncommutative. Let $B \subseteq \Sigma^*$ be a monoid over the alphabet Σ, and let K be a field. Then a noncommutative polynomial has the form $\sum_{t \in B} \alpha_t t$ where α_t is a coefficient from the field K, for each $t \in B$, nonzero for only a finite number of t. For example, if $\Sigma = \{a, b, c\}$ and K is the field of rational numbers, then an example polynomial is $babc + 6abb + 12c$. The polynomial ring $K\langle a, b, c\rangle$ is the collection of all polynomials formed in this way. In general, B may be a semigroup, such as in the case of path algebras where B is the set of finite paths in a finite directed graph [6]. The leading term of a polynomial p, denoted $tip(p)$, is determined by an admissible order on B. An *admissible order* on B is a well-order that is monotonic with respect to composition of words.

An *ideal* in a polynomial ring is a set of polynomials that is both closed under addition and also closed under multiplication (on either side) by any element of the ring [3]. In noncommutative algebras, ideals have "sides" because the variables in the terms do not commute. Therefore, if p is an ideal element and q is any polynomial, then $q \cdot p$ is not the same as $p \cdot q$. The ideals considered here are two-sided, meaning that the result of multiplying an ideal element by another polynomial on either or both sides is also an element of the ideal. Given a set of polynomials F, an ideal $\langle F\rangle$ can be found by including all polynomials formed by either multiplying elements of F on both sides by elements of the ring, or by addition. If the ideal

$\langle F \rangle = I$ for some ideal I, then F is a *generating set* for I.

A Gröbner basis G for an ideal I is a generating set that has the property that for each $p \in I$ there is a $g \in G$ such that the leading term of g is factor of the leading term of p: $tip(p) = u \cdot tip(g) \cdot w$ where $u, w \in B$. Not every generating set satisfies this property. For example, if F contains $p = cc - b$ and $q = cab - ac$ we can form a new polynomial by multiplying p on the right by ab (an element of the ring) and q on the left by $-c$ and adding to yield $p \cdot ab - c \cdot q$, which can be written $bab - cac$. This polynomial $bab - cac$ is an element of $\langle F \rangle$ but its leading term is not divisible by the leading term of any element in F. If G is a Gröbner basis for I, then membership of a polynomial q in I can be decided by finding the remainder of p divided by the elements of G.

The operation of finding the remainder using long-division of polynomials is called *polynomial reduction*. Reduction focuses on the leading term of the dividing polynomial. For example, to reduce $p = babc + 6abb + 12c$ by $q = ab - cc$ we multiply q by b on the left, c on the right, and subtract the result from p. This yields

$$\begin{aligned} p - bqc &= babc - babc + 6abb + 12c + bccc \\ &= bccc + 6abb + 12c. \end{aligned}$$

This reduction is called a *tip-reduction*, since it uses the fact that $tip(p) = b \cdot tip(q) \cdot c$. Tip reduction only reduces the polynomial at the leading term, but general (total) reduction reduces starting at any monomial in the polynomial.

A polynomial p is *reducible* by a polynomial q, if for some monomial m of p, $m = s \cdot tip(q) \cdot t$ for words w and t (if m is the leading term of p, then p is *tip-reducible*). Alternatively, p is *irreducible* (by q) if no monomial of p can be reduced by the leading term of q. If p is reducible by q (meaning $m = s \cdot tip(q) \cdot t$ for a monomial m of p), then the reduction of p by q is $p - sqt$. Reduction by a set of polynomials is the repetition of reductions by the elements of the set until the result is irreducible. The importance of Gröbner bases is that reduction by a generating set reaches a unique irreducible polynomial exactly when the generating set is a Gröbner basis.

The fact that $m = swt$ for two words m and w is written $w|m$ (read w divides m). Given a polynomial h and a set of polynomials P, then h is said to be *tip-reduced* with respect to P if there is no element of P whose leading term divides the leading term of h. A set P is called *tip-reduced* with respect to h if no element of P has a leading term divisible by the leading term of h. Generally, we just say that h is *reduced* when we mean tip-reduced. A polynomial (and a set) can also be *totally reduced*, which means no monomial is reducible.

2.2 Gröbner basis algorithm

A Gröbner basis G has the property that for all elements p of the ideal $\langle G \rangle$, the leading term of p is divisible by the leading term of some g in G. It turns out (not so obviously) that this property is equivalent to a condition on special combinations of pairs of polynomials whose leading terms overlap. Given two polynomials $ab + c$ and $bd + e$ whose leading terms have an overlap of b, the *overlap relation* for these polynomials is $(ab+c)d - a(bd+e) = cd - ae$. (Notice that the overlap relation causes the common multiple abd of the leading terms to be canceled.) The condition that G is a Gröbner basis is equivalent to the condition that all overlap relations of pairs of elements of G (including overlaps of an element with itself) reduce to zero by G. This fact is used in Buchberger's algorithm [5] for computing Gröbner bases.

We illustrate the noncommutative version of Buchberger's algorithm by an example. The algorithm takes as input a generating set $F = \{cab - ac, cc - b, baba - bc\}$, and maintains a generating set G, and the set of overlaps of elements of G. Initially $G = F$ and the set of overlaps is that of the elements of F. The algorithm iterates the following steps: choose an overlap, form the overlap relation, reduce the overlap relation, and if the result is nonzero then add it to G and find the new overlaps. (The algorithm for noncommutative Gröbner bases is presented by Keller [7] and Mora [10]).

Denote the overlap c of cc with cab as $c\boxed{c}ab$. Then the initial set of overlaps, formed from the elements of F, are

$$\{c\boxed{c}c, cc\boxed{c}ab, ca\boxed{b}aba, ba\boxed{ba}ba\}.$$

In our example, we could first choose the overlap $c\boxed{c}c$. For this overlap, the overlap relation h is $(cc - b)c - c(cc - b)$ or $bc - cb$, which is irreducible. When h is added to G, overlaps of

bc are found with the other elements of G (and with itself, if any self overlaps exist). The new overlaps $b\boxed{c}c$ and $b\boxed{c}ab$ are added to the overlap set and the computation continues.

Next we choose the overlap $cc\boxed{c}ab$, which gives the overlap relation h as $(cc-b)ab-c(cab-ac)$ or $bab-cac$. This polynomial is reduced with respect to G, but notice that G is not reduced with respect to h. In particular, $baba-bc$ is reducible by $bab-cac$.

Therefore, before adding the new element, $baba-bc$ is removed from G (and all overlaps involving $baba$ deleted). Then $bab-cac$ is added to G and new overlaps involving bab with the elements of G (including itself) are formed. The reducible element $baba-bc$ is reduced by G, yielding $baba-bc-(bab-cac)a = caca-bc$, and the result added back into G. If any element of G is reducible by $caca-bc$, then the process is repeated. This process of *set reduction* ensures that the elements of G are reduced with respect to each other.

After set reduction G is $\{cab-ac, cc-b, caca-bc, bab-cac\}$, and the set of overlaps is

$$\{ba\boxed{b}ab, ca\boxed{b}ab, c\boxed{c}aca, ca\boxed{ca}b\}.$$

Continuing these steps gives the Gröbner basis

$$\{cab-ab, cc-b, bab-ab, bc-cb, cac-ab, aba-cb, bac-ab, acb-ab, bb-cb, aab-cb, cba-ab\}.$$

2.3 Pattern Matching in the Gröbner Basis Computation

The Gröbner basis computation is dominated by operations that use pattern matching. These operations are computing overlaps for forming triples, polynomial reduction, and set reduction. Both reduction and overlap computation occur at every iteration of the algorithm, and it is important to make sure that these operations are as fast as possible. These operations are defined as follows. (The leading term of a polynomial p is denoted $tip(p)$, and the set of leading terms of a set of polynomials P is denoted $Tip(P)$.)

Computing overlaps. Overlaps are computed when a new nonzero, reduced overlap relation h is added to G. An overlap of a word p with a word q is a suffix of p that is also a prefix of q.

The overlaps to be computed are all those of $tip(h)$ with $Tip(G) \cup \{tip(h)\}$.

Polynomial reduction. Polynomial reduction of a polynomial q by set of polynomials P requires all $p \in P$ such that $tip(p)|m$ for some monomial m of q. For tip-reduction, $m = tip(q)$.

Set reduction. Set reduction is performed when a new nonzero reduced overlap relation h is added to G. Since set reduction removes all elements of G that are tip-reducible by h, we want to find all $g \in G$ such that $tip(h)|tip(g)$.

The tip set $Tip(G)$ of G can be viewed as a dictionary of words on which searches must be performed. Only three types of pattern matching searches are needed for the above operations. Overlap searches are (obviously) used to compute overlaps, subword searches are used for polynomial reduction, and superword searches are used for set reduction. In addition to fast algorithms for these searches, a pattern matching mechanism must allow fast modifications to the dictionary. Our extension of dynamic dictionary matching satisfies these requirements.

3 Pattern Matching Searches

Note that G has the property that no leading term of an element of G is a subword of another leading term. Therefore, we assume that the dictionary of words D has this property. In particular, for all $d \in D$, there is no d' such that $d'|d$.

If w is a string not in D, the subword and superword searches are the following:

1. *Subword search* – find all (d, i) where $d \in D$ and d is a subword of w beginning at the ith symbol of w.

2. *Superword search* – find all (d, i) where $d \in D$ and w is a subword of d beginning at the ith symbol of d.

The overlap searches are split into two kinds. Again, w is a word not in D, and (based on the discussion of the operation of computing overlaps) w is not a subword of any element of d.

1. *Left-overlap search* – find all (d, s) where $d \in D$ and s is a suffix of d and a prefix of w.

2. *Right-overlap search* – find all (d, p) where $d \in D \cup w$ and p is a prefix of d and a suffix of w.

Note that the algorithm we use to insert words into the dictionary makes it convenient to have the search of overlaps of w with itself to be a right-overlap search. If another insertion algorithm is used, then it may be better to have the search for self overlaps be a left-overlap search.

4 Pattern Matching Solution

The dynamic dictionary matching approach of Amir *et al.* [2] provides linear time modifications to the dictionary, but only implements the subword search. In particular, they show how to use a suffix tree to implement the automata used for the subword search (essentially replacing the data structure used by Aho and Corasick [1]). We show how to also use the suffix tree structure for the superword and overlap searches. We first describe the suffix tree we use and then discuss the searches.

4.1 Suffix Trees

A suffix tree is a compacted trie (an edge may be labeled by an arbitrary length word) for the suffixes of a word. Suffix trees can be defined as follows.

Definition 1 *Given a string s of length n terminated by a unique symbol # that appears nowhere else in s, the* suffix tree *for s is a tree satisfying:*

1. *each edge of the tree is labeled by a substring of s, and each node corresponds to a position in s;*

2. *no two sibling edges have labels with a common nonempty prefix; and*

3. *each leaf corresponds to a particular suffix of s given by the concatenation of the labels from the root to the leaf.*

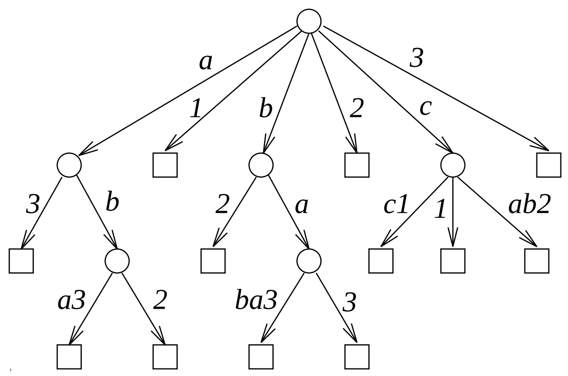

Figure 1: Dictionary Suffix Tree.

Note that each node v in a suffix tree corresponds to a unique string $l(v)$ formed by concatenating the labels of the edges from the root to v. For each v, the label $l(v)$ is a prefix of the suffixes represented by all leaves of the subtree rooted at v. Given a word s such that $s = l(v)$ for some node v of a suffix tree, then v is called the *locus* of s [9]. Note that if there are k leaves of the suffix tree, the size of the tree is bounded by $2k - 1$.

For dictionary matching, the suffix tree holds the suffixes for the individual dictionary words. If the dictionary is $\{w_1, \ldots, w_n\}$, the suffix tree is isomorphic to the suffix tree for the word formed by concatenating the words w_i separated by special symbols outside of the alphabet. For our purposes, numeric digits are sufficient, so the tree for a dictionary of n words corresponds to the tree for the string $w_1 1 w_2 2 \cdots n-1 w_n n$. For example, the tree for the dictionary $\{cc, cab, baba\}$ shown in Figure 1 corresponds to the suffix tree for $cc1cab2baba3$ (where any part of the suffix past a digit has been deleted). The tree is actually built by inserting each dictionary word in sequence.

The insertion algorithm and data structure used for the suffix tree is based on that of McCreight [9]. McCreight's algorithm inserts the suffixes from the longest to the shortest, and uses suffix links in the tree so that insertion requires only linear time. A *suffix link* is an extra arrow from a node v to the node holding the suffix for $l(v)$ formed by removing the first symbol. The tree for the dictionary $\{cc, cab, baba\}$ with suffix links added is shown in Figure 2. McCreight's insertion algorithm is also used for insertion of suffixes of dictionary words by Amir

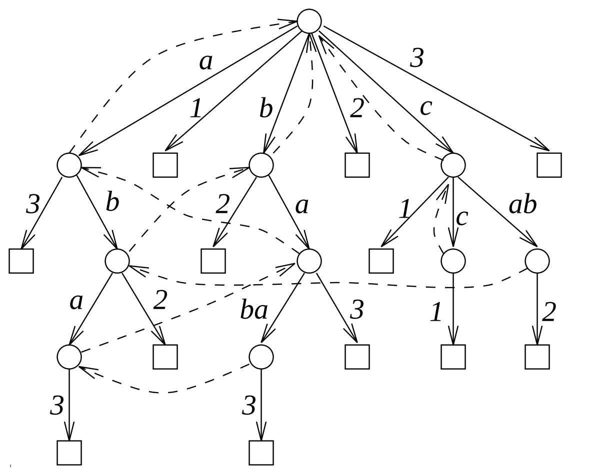

Figure 2: Dictionary Suffix Tree with Suffix Links.

et al, and they use the suffix links to simulate an automata with failure paths (such as those used by Aho and Corasick).

In the general situation considered by Amir *et al.*, one dictionary word may be a subword of another. In this case, the longer word is hidden by the shorter word, and so an additional data structure is required to keep track of the subword relationships. However, this situation does not occur for the Gröbner basis computation and so the suffix tree is all that is needed.

We make two modifications to the suffix tree. The first is to distinguish between nontrivial suffixes and full dictionary words. To do this, define *pattern* leaves to correspond to full dictionary words, and *suffix* leaves to correspond to any other suffix. This modification is mostly conceptual, since leaves are actually labeled by something that identifies the suffix and the word it belongs to.

The second modification to the suffix tree is to the nodes of the tree. A counter is added to each node v that indicates the number of pattern leaves in the subtree rooted at v. The insertion and deletion algorithms must be modified to maintain these counters, but both operations are still accomplished in linear time.

4.2 Searches

The dynamic dictionary matching approach can be easily extended to the pattern matching searches needed for computing Gröbner bases. The approach already solves the subword problem, so we only need to add the superword and overlap searches. Since we can assume that no dictionary word is a subword of another, only the suffix tree is used.

All searches *scan* their input with the suffix tree. Scanning is the operation of comparing the word to the labels of the edges. For example, to scan the word *cab* by the tree in Figure 2, we begin at the root, match c with the arc labeled c and then match ab to the child with the arc labeled ab. Scans may fail if the symbols of the next child to be considered do not match the prefix of the remaining input word. As an example, a scan of the word *bac* by the tree in Figure 2 fails because there is no arc from the locus of ba that is labeled by c. The operation of determining which branch matches an input word is considered to take constant time. Therefore, scanning a word w takes time $O(|w|)$.

The complexity of all searches is of the form $O(n+m)$, where n is the size of the input and m is the size of the output. The input size is $n = |w|$ in all searches, but the size of the output m depends on the search. In particular, the subword search takes time $O(n+m)$ for which n is as above, and $m = \sum_{d \in D} \#(d, w)$ where $\#(d, w)$ is the number of times d occurs in w.

4.2.1 Superword Search

The superword search takes as input a word w that is not an element of D, and returns as output the set of pairs (d, i) where $d \in D$ and w divides d beginning at position i. This problem is equivalent to finding all suffixes of $d \in D$ that have w as a prefix. The length of the suffix indicates the position where w divides d, and therefore determines the pair.

The algorithm then is to scan w with the suffix tree for D to find the locus of w. If the scan fails, then w is not a subword of any dictionary word. Otherwise, the locus v of w exists in the tree. In this case, return all leaves of the subtree rooted at v.

Scanning w takes time $O(|w|)$, and the search of the subtree takes time proportional to the size of the subtree. The size of the subtree is on the order of the number of leaves, which is the size of the output m. In particular, $m = \sum_{d \in D} \#(w, d)$. Therefore, the superword search takes time $O(n + m)$.

4.2.2 Left-overlap Search

The left-overlap search takes as input a word w, where w is not in D and is not a subword of any d in D, and returns the set of pairs (d, k) where k is the length of a suffix of d that is a proper prefix of w. Viewed the other way, this problem is to find any proper prefix of w that is also a suffix of some d in D.

The algorithm is to scan w with the suffix tree for D. If during the scan a node is reached that has a suffix leaf as a child, then the corresponding suffix should be returned. Stop if the scan fails, or the $(|w| - 1)$th symbol of w is reached.

Again, scanning w with the tree takes time $O(|w|)$, and we must visit $O(m)$ suffix leaves. In this case,

$$m = \sum_{d \in D} \sum_{k < \min(|d|, |w|)} p(d_k, w),$$

where d_k is the length k suffix of d, and $p(d_k, w)$ is a function which returns one if d_k is a prefix of w and zero otherwise.

4.2.3 Right-overlap Search

The right-overlap search takes as input a word w, where w is not in D and is not a subword of any d in D, and returns the set of pairs (d, k) where k is the length of a prefix of d ($d \in D \cup \{w\}$) that is a proper suffix of w. This search is combined with the insertion algorithm.

We use McCreight's insertion algorithm, which inserts the suffixes of w into the tree from longest to shortest. This means that w is in the tree before any other suffix of w is inserted. Therefore, when a proper suffix w_k of w is inserted into the tree, any d from $D \cup \{w\}$ with w_k as a prefix is a pattern leaf in the subtree rooted at the locus of w_k.

The algorithm uses this fact, and after each insertion of a proper prefix w_k searches the subtree rooted at the locus of w_k for all pattern leaves. Recall that each node is marked with the number of pattern leaves in its subtree. The algorithm checks each child and searches a subtree only if its root has a nonzero number of pattern leaves.

The insertion algorithm alone takes $O(|w|)$ time with each suffix insertion taking amortized constant time [2]. The addition of the subtree search does not affect this time for inserting each suffix. During the search for each insertion, the restriction of the search to subtrees with pattern leaves takes time bounded by $(c-1)|d|$ for each dictionary word d corresponding to a pattern leaf in the subtree where c is the number of symbols in the alphabet. Since c is a constant the total time required for searching is $O(m)$ where m is the sum of the lengths of the dictionary words in the answer.

5 Conclusion

We have described an extension of the dynamic dictionary matching approach of Amir *et al.* to the string completion problem as it arises in the computation of noncommutative Gröbner bases. This pattern matching approach fits well within the algorithm and provides both fast searches (linear in the size of the input and output) and fast modifications to the dictionary. The ability to make fast modifications is crucial to Gröbner basis computations where the dictionary of patterns can change frequently. This data structure is a crucial advantage of our implementation of the noncommutative Gröbner basis algorithm for path algebras, called Opal [7, 8].

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and Systems Sciences*, 49(2), 1994.

[3] T. Becker and V. Weispfenning. *Gröbner Basis: A Computational Approach to Commutative Algebra.* Springer-Verlag, New York, 1993.

[4] R. V. Book and F. Otto. *String-Rewriting Systems.* Springer-Verlag, 1993.

[5] B. Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, Mathematics and its Applications, pages 184–232. D. Reidel Publishing Company, Dordrecht, Holland, 1985.

[6] D. R. Farkas, C. D. Feustel, and E. L. Green. Synergy in the theories of Gröbner bases and path algebras. *Canadian Journal of Mathematics*, 45(4):727–739, 1993.

[7] B. J. Keller. Alternatives in implementing noncommutative Gröbner basis systems. Submitted to the *Proceedings of Conference on Symbolic Rewriting Systems*, Monte Verita, Ticino, Switzerland, May 1–5 1995.

[8] B. J. Keller. *Algorithms and Orders for Finding Noncommutative Gröbner Bases.* PhD thesis, Virginia Polytechnic Institute & State University, (in progress).

[9] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[10] T. Mora. An introduction to commutative and non-commutative Gröbner bases. *Theoretical Computer Science*, 134:131–173, 1994.

[11] C. C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, New York, 1994.