

PhD Seminars 2002/2003: G. Evans Semester 2

Gareth Evans

October 29, 2003

Contents

| | |
|--|-----------|
| 1 Seminar 1: 29th January 2003 | 3 |
| 1.1 Automatic Groups | 3 |
| 1.1.1 Finite State Automata | 3 |
| 1.1.2 The Word Problem | 3 |
| 1.1.3 Automatic Groups: Definition | 4 |
| 1.1.4 Solving the Word Problem in Quadratic Time | 5 |
| 2 Seminar 2: 5th February 2003 | 8 |
| 2.1 Constructing Automatic Structures | 8 |
| 2.1.1 The Knuth-Bendix Procedure | 8 |
| 2.2 Constructing Word-Difference Automata | 9 |
| 2.3 Calculating the Word Acceptor W | 10 |
| 3 Seminar 3: 12th February 2003 | 15 |
| 3.1 Calculating the Multipliers M_x | 15 |
| 3.2 Testing the Automata | 16 |
| 3.2.1 Axiom Checking | 16 |
| 3.2.2 Axiom Checking in <i>kbmag</i> | 17 |

| | | |
|-----|------------------------------------|----|
| 3.3 | Conclusions | 18 |
| 3.4 | Automatic Gröbner Bases? | 18 |

1 Seminar 1: 29th January 2003

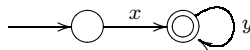
1.1 Automatic Groups

In this series of seminars we shall discuss the definition, uses and algorithmic implementation of Automatic Groups.

1.1.1 Finite State Automata

The definition of an automatic group assumes knowledge of finite state automata. We shall now briefly recall that a (deterministic) finite state automata is represented by $A = (S, \Sigma, s_0, \delta, F)$, where S is a set of states, Σ is an input alphabet, s_0 is the initial state, F is a set of terminal or accepting states, and $\delta : S \times \Sigma \rightarrow S$ is the transition function, where $\delta(s_1, a) = s_2$ (s_1 is the current state and a is the input letter) is a transition. Note that a non-deterministic finite state automata (fsa) does not necessarily have a transition defined for each member of the alphabet at each state and does not necessarily have a unique initial state. Further, a non-deterministic fsa with ϵ -transitions allows special ϵ -transitions between states allowing simpler automata to be constructed. All non-deterministic fsa (with or without ϵ -transitions) may be converted to deterministic fsa, but only in exponential time.

A string made up of letters of the alphabet Σ is said to be an accepted string if the string defines a path in the fsa from an initial state to a terminal state. The set of all accepted strings make up the language of the fsa. In the following example, we see that the set of strings accepted by the fsa is the set $\{xy^i \mid i \geq 0\}$, and so the language of the fsa is infinite.



1.1.2 The Word Problem

An important application of the theory of automatic groups is the notion that if a group is automatic then the word problem may be solved in quadratic time. We will discuss this notion in more detail later on in this seminar.

Definition 1.1 The word problem in a group G consists of finding an algorithm that takes as its input a word w in the generators of G and answers Yes or No, depending on whether or not w represents the identity word in G .

1.1.3 Automatic Groups: Definition

Definition 1.2 Let G be a group. An automatic structure on G consists of a set A of semigroup generators of G , a finite state automata W over A , and finite state automata M_x over (A, A) for $x \in A \cup \{\epsilon\}$, satisfying the following conditions:

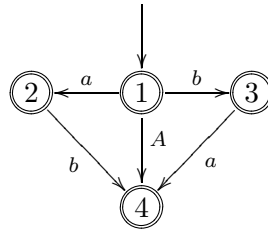
- The map $\pi : L(W) \rightarrow G$ is surjective;
- For $x \in A \cup \{\epsilon\}$, we have $(w_1, w_2) \in L(M_x)$ iff $\overline{w_1 x} = \overline{w_2}$ and both w_1 and w_2 are elements of $L(W)$.

We call W the word acceptor, M_ϵ the equality recogniser, and each M_x , for $x \in A$, a multiplier automaton for the automatic structure. An automatic structure is sometimes called an automation. An automatic group is one that admits an automatic structure.

Example 1.3 Let us now consider a concrete example of an automatic structure for the group S_3 . Let S_3 be presented as follows (this is a ‘monoid’ presentation for S_3):

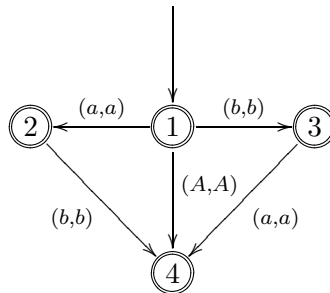
$$S_3 = \langle a, b, A, B \mid aA, Aa, bB, Bb, a^3, b^2, (ab)^2 \rangle .$$

An automatic structure for this presentation will consist of a word acceptor W , the equality recogniser M_ϵ , and a set of general multipliers M_x based on the set of semigroup generators $A = \{a, b, A, B\}$. Knowing that the six elements of S_3 are $\{e, A, a, b, ab, ba\}$, the word acceptor W is as follows:



Notice that the language accepted by the above (non-deterministic) finite state automata is identical to the set of six elements mentioned previously so that the map $\pi : L(W) \rightarrow G$ (in this case the identity map between the strings in $L(W)$ and the group elements in G) is indeed surjective.

Now for the multipliers let us first consider the equality multiplier M_ϵ :

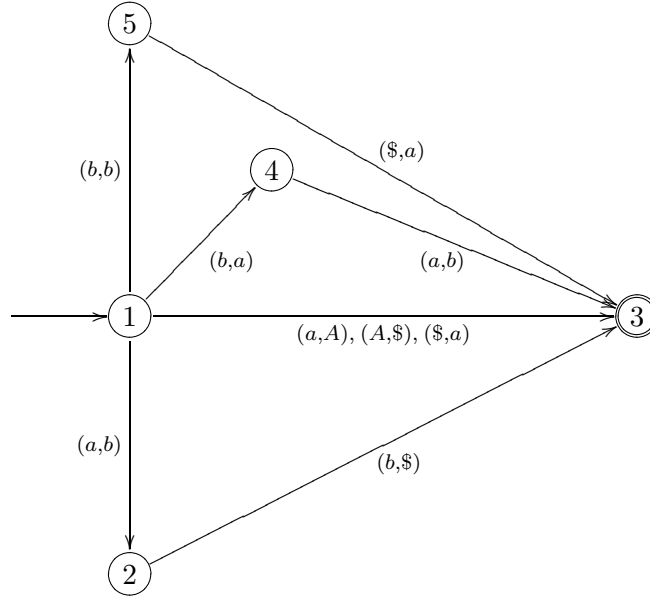


Notice that the language accepted by this multiplier is the set of strings

$$\{(\epsilon, \epsilon), (a, a), (b, b), (A, A), (ab, ab), (ba, ba)\},$$

i.e. the set of strings (u, v) such that $\bar{u} = \bar{v}$, as required. Notice the similarity between the word acceptor W and the equality multiplier.

For a more general multiplier, let us consider the multiplier M_a :



The language accepted by M_a is the set of strings

$$\{(a, A), (A, \$), (\$, a), (ba, ab), (b\$, ba), (ab, b\$)\}.$$

We can easily verify that these strings (u, v) satisfy $\bar{u}\bar{a} = \bar{v}$, e.g. it is obvious that for the first string (a, A) we have $\bar{a}\bar{a} = \bar{A}$ because ' $a^2 = a^{-1} = A$ '. In fact, all the above automaton does is to show what happens to the elements of S_3 when multiplied on the right by a , and a similar process occurs with each of the other multipliers M_A , M_b and M_B .

Remark 1.4 The multiplier automata are two variable finite state automata, which mean that they read two strings (u, v) at the same rate. In order to counter the case where the strings u and v are not of the same length, we introduce a padding symbol $\$$ and pad the shorter string with $\$$'s to make it equal in length to the larger string.

1.1.4 Solving the World Problem in Quadratic Time

Proposition 1.5 (*Bounded Length Difference*) *Let G be an automatic group and let (A, L) be an automatic structure for G (i.e. A is a set of semigroup generators for G and $L = L(W)$ is the language of the associated word acceptor W). There is a constant N such that, if $w \in L$ is an accepted*

word and $g \in G$ is a vertex of the Cayley graph at a distance of at most one from \bar{w} , we have the following situation:

- (a) g has some representative of length at most $|w| + N$ in L ; and
- (b) if some representative of g in L has length greater than $|w| + N$, there are infinitely many representatives of g in L .

Proof: Let N be greater than the number of states in any of the automatic structure's finite state automata, let w and g be as in the statement of the proposition, and let w' be any representative of g in L . One of the pairs (w, w') or (w', w) is accepted by the appropriate automaton M_x , for some $x \in A \cup \{\epsilon\}$. If $|w'| > |w| + N$, the automaton M_x undergoes more than N transitions after reading all of w , and therefore visits the same state twice. One can then shorten w' by eliminating the loop between two visits to this repeated state, and still get an accepted word representing the same group element; this proves (a). Alternatively, one can lengthen w' arbitrarily by going over the loop repeatedly; this proves (b). \square

Theorem 1.6 (*Quadratic Algorithm*). *Let G be an automatic group and (A, L) an automatic structure for G . For any word w over A , we can find a string in L representing the same element of G as w , in time proportional to the square of the length of w .*

In particular, knowing some word e representing the identity element, we can solve the word problem in quadratic time, by finding a representative in L for the desired word, and feeding it and e to the equality recogniser.

Proof: We assume that the multiplier automata M_x , for $x \in A$, are normalised (a finite state automata operation). Suppose we are given an accepted string u and a generator x , and we want to find an accepted representative v for \overline{ux} . The idea is that M_x accepts some pair (u', v') , where u' equals u , possibly followed by some number of $\$$'s, and v' is an accepted word representing \overline{ux} , also possibly followed by $\$$'s. If we ignore the second element of its labels, M_x becomes a non-deterministic automaton in one variable accepting u' . We find a path of arrows corresponding to u' , and read off v' by looking at the second element of the labels in this path of arrows.

More precisely, let S_0 be the set whose only element is the initial state set s_0 of M_x . For $i > 0$, we inductively define T_i as the set of arrows with source in S_{i-1} and label (x_i, y_i) , where $y_i \in A \cup \{\$\}$ is arbitrary, and x_i equals either the i -th character in u (if $i \leq |u|$), or $\$$ (if $i > |u|$). We also define S_i as the set of targets of arrows in T_i . We let n be the smallest number such that $n \geq |u|$ and S_n includes an accept state of M_x . Working backwards, we read off the labels $y_n \dots y_1$ on a path of arrows joining the initial state to an accept state, and we form the string $v' = y_1 \dots y_n$, which represents \overline{ux} . We finally discard trailing $\$$'s to obtain v .

Since there are only finitely many arrows and states, the time taken by each step in this induction is bounded by a constant. Therefore the overall time is proportional to the number n defined above. By

Proposition 1.5, n can only exceed $|u|$ by a bounded amount N , otherwise it would not be minimal. This shows that, given an accepted string u , we can find a representative for \overline{ux} in time $O(|u|)$, and the representative's length is at most $|u| + N$. Replacing (x_i, y_i) by (y_i, x_i) in the previous discussion, we see that similar estimates apply when we multiply u by x^{-1} .

Now our goal in this proof is to show that we can find a string in L representing the same element of G as an arbitrary word w in the generators A in time proportional to the square of the length of w . Let us begin by considering the identity element 1_G of the group G . We can construct an accepted representative of this identity (i.e. find an $\ell \in L$ such that $\pi(\ell) = 1_G$) from the automatic structure: we start with any accepted string $x_1 \dots x_n$ and use the procedure described above to multiply successively by $x_n^{-1} \dots x_1^{-1}$. Let us assume that the length of a representative α of the identity is $|\alpha| = n_0$.

Given an arbitrary word w , assume that $w = x_1 \dots x_p$. The first step is to find an accepted representative of the word x_1 , and we do this by simply applying the above procedure with $u = \alpha$ and $x = x_1$ to give us an accepted representative β for $\alpha x_1 = x_1$. This can be done in time $O(n_0)$, and the representative's length is at most $n_0 + N$.

The next step is to find an accepted representative of the word $x_1 x_2$. To do this, we apply the procedure with $u = \beta$ and $x = x_2$ to give us an accepted representative γ for $\beta x_2 = x_1 x_2$. This can be done in time $O(n_0 + N)$, and the representative's length is at most $(n_0 + N) + N$. Continuing by induction, we see that it will be possible to find a representative for $w = x_1 \dots x_p$ in time $O(\sum_{i=1}^{|w|} (iN + n_0)) \leq O(\sum_{i=1}^{|w|} (|w|N + n_0)) = O(|w|(|w|N + n_0)) = O(|w|^2)$, and the representative's length will be at most $N|w| + n_0$. \square

We now know that given an arbitrary word w in the generators of an automatic group G , we can test to see whether w simplifies to give the identity word 1_G in quadratic time, and so the word problem for an automatic group is solvable in quadratic time.

In the next seminar, we shall look at how an automatic structure is constructed, both manually and by using the computer program *kbmag*.

2 Seminar 2: 5th February 2003

In the previous seminar, we discussed the definition of an automatic group and saw how the word problem can be solved in quadratic time given an automatic structure. In this seminar, we shall begin to see how we can construct an automatic structure and then test it to make sure that it really is a valid automatic structure.

2.1 Constructing Automatic Structures

Given a presentation for a group G , how do we attempt to construct an automatic structure for G ? The method we use is to interrupt the Knuth-Bendix procedure at a suitable stage, construct word-difference automata from the rewrite system obtained, apply finite state automata operations to obtain the word difference machine and the multiplier automata, and finally check to see whether the automata represent a valid automatic structure for G .

2.1.1 The Knuth-Bendix Procedure

The Knuth-Bendix procedure is an algorithm that takes as input a presentation for a group G and attempts to provide as output a complete rewrite system for G . This means that the procedure attempts to give back a set of rewrite rules allowing a word in the generators of G to be simplified or rewritten into ‘normal form’, i.e. an element of the group G . Specifically, the word problem can be solved given a complete rewrite system by testing to see whether the normal form of a test word w is the identity word or not.

Assume that the presentation for our group G has generators X and relators R . For each generator $a \in X$, (i) introduce generators $A = a^{-1}$ if they are not already there, and (ii) introduce relators $aA = 1$ and $Aa = 1$ if they are not already there, so that our group presentation is now a monoid or semigroup presentation $G = \langle X' \mid R' \rangle$. Fix an order on words in the generators X' , e.g. the shortlex order (which we shall use from now on unless otherwise stated) orders the words by length first and then alphabetically. We can now construct an initial rewrite system R_1 by writing each relator $l = r \in R'$ as a rewrite rule $l \rightarrow r$, making sure that $l > r$ in our chosen order.

The Knuth-Bendix procedure takes this initial rewrite system and constructs a complete rewrite system from it by constructing new rules from the analysis of the overlaps of left hand sides of rewrite rules. For example, if we had two rewrite rules $uv \rightarrow p$ and $vw \rightarrow q$, where u, v, w, p and q are words in the generators X' , then we see that the left hand sides of the two rules overlap to give an overlap word uvw . We can now reduce this word in two ways to give two words pw and uq , and after we simplify these two words using our current rewrite system (so that $pw \rightarrow \dots \rightarrow j$ and $uq \rightarrow \dots \rightarrow k$, say), we obtain a critical pair of words (j, k) . If $j = k$ then we do nothing but otherwise we add either the rule $j \rightarrow k$ or the rule $k \rightarrow j$ to our rewrite system depending on whether j or k is the biggest

word in our fixed word order. The Knuth-Bendix procedure stops when we have no non-trivial critical pairs arising from overlaps of left hand sides of rewrite rules in our current rewrite system — and the rewrite system is then termed complete.

The problem with the procedure is that it is highly repetitive and when implemented on a computer this repetitiveness means that even for simple examples the algorithm may not terminate quickly, even when tricks such as eliminating redundant rules and using shortcuts are used (e.g. replacing the two rules $u \rightarrow v$ and $v \rightarrow w$ by the rules $u \rightarrow w$ and $v \rightarrow w$). The beauty of constructing an automatic structure is that an automatic structure can be constructed even if the Knuth-Bendix procedure has not yet terminated, so we can solve the word problem without having to have the Knuth-Bendix procedure terminate. This means that for groups where the Knuth-Bendix procedure does not terminate, we may still be able to solve the word problem.

2.2 Constructing Word-Difference Automata

The process of finding an automatic structure starts when we interrupt the Knuth-Bendix procedure at a suitable stage, for example when no new rules have appeared for some time (of course, if the Knuth-Bendix procedure terminates, then we may use the complete rewrite system and not an intermediate rewrite system).

We must now form a candidate for a word-difference automaton D_1 to be used in constructing the word acceptor W . We do this as follows. $S(D_1)$, the state set of D_1 , is initially empty. Consider each rule (u, v) of our rewrite system in turn, and let $u = x_1x_2 \dots x_m$ and $v = y_1y_2 \dots y_n$, with $m \geq n$ (shortlex order). We pad v if necessary in the usual way, by putting $y_i = \$$ for $n < i \leq m$. Let $d_0 = e$ and for $1 \leq i \leq m$ let d_i be the reduction, using the current rules, of $x_i^{-1}d_{i-1}y_i$ (so d_i is a word-difference). Then we adjoin the states d_i to $S(D_1)$ and the transitions $d_{i-1} \xrightarrow{(x_i, y_i)} d_i$ to D_1 if any of them are not there already. The start and accept states are both e .

Example 2.1 Let us consider the construction of the word difference machine D_1 for the following monoid presentation of the group S_3 :

$$S_3 = \langle a, b, A, B \mid aA, Aa, bB, Bb, a^3, b^2, (ab)^2 \rangle,$$

where $a^{-1} = A$, $b^{-1} = B$ and the shortlex order chosen is $a < b < A < B$. For this presentation, the Knuth-Bendix procedure terminates quickly with the following complete set of rewrite rules:

$$\begin{array}{ll} (1) & aA \rightarrow e \\ (2) & Aa \rightarrow e \\ (3) & b^2 \rightarrow e \\ (4) & a^2 \rightarrow A \\ (5) & bab \rightarrow A \end{array} \quad \begin{array}{ll} (6) & B \rightarrow b \\ (7) & aba \rightarrow b \\ (8) & A^2 \rightarrow a \\ (9) & Ab \rightarrow ba \\ (10) & bA \rightarrow ab \end{array}$$

Using the method outlined above, let us now construct the word difference set $WD = S(D_1)$ for this example (where $\sigma(r)$ denotes the reduction of the word r using the ten rules above):

Rule 1: $d_0 = e, d_1 = \sigma(A) = A, d_2 = \sigma(aA) = e.$

Rule 2: $d_0 = e, d_1 = \sigma(a) = a, d_2 = \sigma(Aa) = e.$

Rule 3: $d_0 = e, d_1 = \sigma(B) = b, d_2 = \sigma(Bb) = e.$

Rule 4: $d_0 = e, d_1 = \sigma(A^2) = a, d_2 = \sigma(Aa) = e.$

Rule 5: $d_0 = e, d_1 = \sigma(BA) = ab, d_2 = \sigma(Aab) = b, d_3 = \sigma(Bb) = e.$

Rule 6: $d_0 = e, d_1 = \sigma(bb) = e.$

Rule 7: $d_0 = e, d_1 = \sigma(Ab) = ba, d_2 = \sigma(Bba) = a, d_3 = \sigma(Aa) = e.$

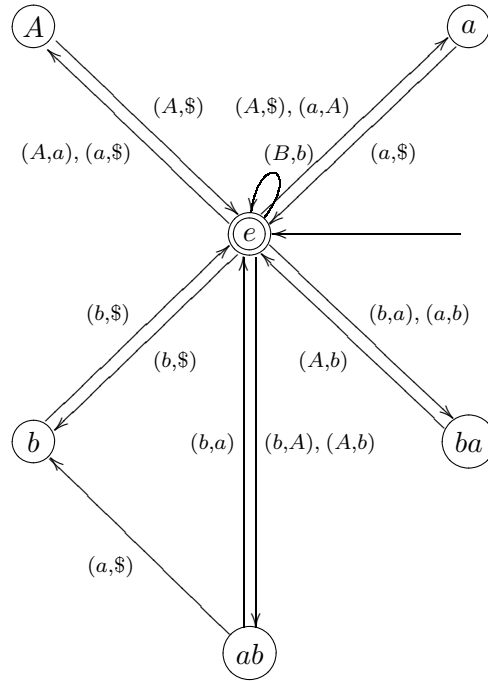
Rule 8: $d_0 = e, d_1 = \sigma(aa) = A, d_2 = \sigma(aA) = e.$

Rule 9: $d_0 = e, d_1 = \sigma(ab) = ab, d_2 = \sigma(Baba) = e.$

Rule 10: $d_0 = e, d_1 = \sigma(Ba) = ba, d_2 = \sigma(abab) = e.$

Conclusion: $WD = \{e, A, a, b, ab, ba\}.$

We can now construct the word difference machine D_1 according to the recipe provided earlier to give the following diagram:



2.3 Calculating the Word Acceptor W

The next step is to construct the word acceptor W as follows:

$$W = (A^*.E(D_1 \wedge GT).A^*)',$$

where GT is a two variable automaton accepting $(u, v)^+$ iff $u > v$ ($^+$ = padded; $u, v \in A^*$), and $E(X)$ is a one variable automaton accepting $\alpha \in A^*$ iff $\exists (\alpha, \beta) \in A^* \times A^*$ such that $(\alpha, \beta)^+$ is accepted by

the two-variable automaton X . Expressed in words, W accepts $w \in A^*$ if and only if it does not have a substring u such that there exists a $v \in A^*$ with $u > v$ and $(u, v)^+$ accepted by D_1 .

Our hope is that the W we construct by the above formula has a language $L(W)$ so that the map $\pi : L(W) \rightarrow G$ is surjective as required by the definition of an automatic structure for an automatic group. Of course this depends on the machine D_1 we construct and therefore on the moment we interrupt the Knuth-Bendix procedure, and so there are checks (made later on) to make sure that the language $L(W)$ has the required property. If a check is failed, e.g. it is found that the map π is not surjective, we then have to restart the Knuth-Bendix procedure and try again with a different word difference machine.

Now how do we calculate W using the formula

$$W = (A^*.E(D_1 \wedge GT).A^*)' ?$$

Basically, all the operators in the formula correspond to finite state automata operations which we could calculate by hand. However, it is much more sensible (and a lot easier!) if we use a computer program to do our work, and it so happens that the computer program *kbmag* contains all the finite state operations we require to calculate W correctly. *kbmag* stands for "Knuth-Bendix on Monoids and Automatic Groups" and is freely available from the Warwick ftp site. We shall see this program in action in future seminars when we shall use it to construct automatic structures, but for now we shall only concentrate on its finite state automata capabilities in order to calculate W .

To start with, our machine D_1 is represented as follows by *kbmag*:

```
_RWS.diff1 := rec(
  isFSA := true,
  alphabet := rec(
    type := "product",
    size := 24,
    arity := 2,
    padding := _,
    base := rec(
      type := "identifiers",
      size := 4,
      format := "dense",
      names := [a,b,A,B]
    )
  ),
  states := rec(
    type := "words",
    size := 6,
```

```

alphabet := [a,b,A,B],
format := "sparse",
names := [
  [1,IdWord],
  [2,A],
  [3,a],
  [4,b],
  [5,b*a],
  [6,a*b]
]
),
flags := ["DFA","trim"],
initial := [1],
accepting := [1],
table := rec(
  format := "sparse",
numTransitions := 17,
  transitions := [[2,5],[3,3],[5,2],[6,5],[8,6],[10,4],[11,2],
    [12,6],[15,3],[17,1]],
    [[15,1]],
    [[5,1]],
    [[10,1]],
    [[10,3],[12,1]],
    [[5,4],[6,1]]
  ]
)
);

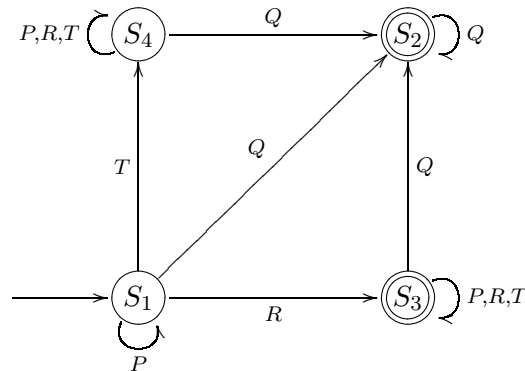
```

The above output follows the GAP record structure (GAP is a computer algebra package, where GAP = Groups, Algorithms and Programming). If we study the output a little, we can see that it represents a finite state automata with six states and seventeen transitions, where the states are labelled with elements from an alphabet and the transitions are labelled with numbers which correspond (in this case) to the recipe shown below:

| | | | | |
|-------------|--------------|--------------|--------------|--------------|
| 1 = a, a | 6 = b, a | 11 = A, a | 16 = B, a | 21 = $\$, a$ |
| 2 = a, b | 7 = b, b | 12 = A, b | 17 = B, b | 22 = $\$, b$ |
| 3 = a, A | 8 = b, A | 13 = A, A | 18 = B, A | 23 = $\$, A$ |
| 4 = a, B | 9 = b, B | 14 = A, B | 19 = B, B | 24 = $\$, B$ |
| 5 = $a, \$$ | 10 = $b, \$$ | 15 = $A, \$$ | 20 = $B, \$$ | |

The transitions are listed by source, so that (for example) from the first state there are 10 transitions — a transition labelled with 2 (i.e. (a, b)) going to state 5, a transition labelled with 3 going to state 3, etc. I hope it is now possible to see how the above output allows us to reconstruct the diagram of D_1 we saw earlier.

The first stage in calculating W is to calculate the automaton corresponding to $D_1 \wedge GT$ (a string is accepted by this machine if it is accepted by both D_1 and GT). For this we need the following ‘Greater-Than’ finite state automaton, using the length-lex ordering $a < b < A < B$:



Key: $P = \{(a, a), (b, b), (A, A), (B, B)\}$
 $Q = \{(a, \$), (b, \$), (A, \$), (B, \$)\}$
 $R = \{(b, a), (A, a), (B, a), (A, b), (B, b), (B, A)\}$
 $T = \{(a, b), (a, A), (a, B), (b, A), (b, B), (A, B)\}$

The finite state automaton corresponding to $D_1 \wedge GT$ is obtained by using the *kbmag* program *fsaand*. This gives back a fsa with 11 states and 44 transitions. We then calculate $E(D_1 \wedge GT)$ with program *fsaexists*, use *fsaconcat* to calculate $A^*.E(D_1 \wedge GT).A^*$, and finally use *fsanot* to obtain $(A^*.E(D_1 \wedge GT).A^*)'$:

```
[root@linux test]# fsaand D1.in GT.in D1GT.in
#"And" fsa with 11 states computed.
```

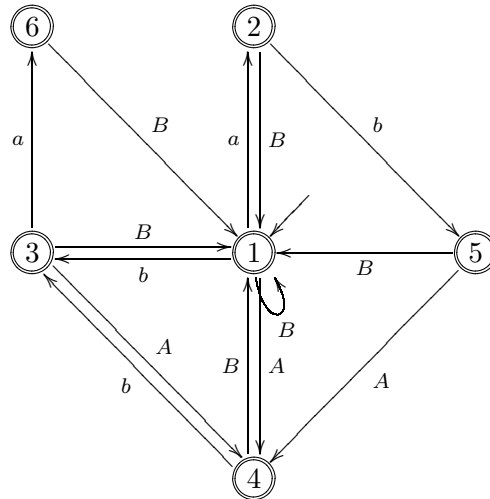
```
[root@linux test]# fsaexists D1GT.in
#"Exists" fsa with 23 states computed.
```

```
[root@linux test]# fsaconcat AStar.in D1GT.in.exists ASD1GT.in
#"Concatenated" fsa with 20 states computed.
```

```
[root@linux test]# fsaconcat ASD1GT.in AStar.in ASD1GTAS.in
#"Concatenated" fsa with 7 states computed.
```

```
[root@linux test]# fsanot ASD1GTAS.in
#"Not" fsa with 6 states computed.
```

All that the above programs do is perform finite state automata operations (which I shall not go into here), and the W we obtain at the end of this process is as follows:



Notice that the above W does not correspond to the W we saw in the last seminar. This is because the W we saw in the last seminar was calculated using a word difference machine D_2 to obtain a simpler W , and we shall discuss the construction of this machine D_2 and the construction of the multipliers in the next seminar.

3 Seminar 3: 12th February 2003

At the end of the previous seminar we had arrived at the stage where we had calculated the word acceptor W for the example that we were considering. In this seminar we will first calculate the multipliers M_x and then go on to check to see if the finite state automata that we have calculated form part of an automatic structure for our presentation of S_3 .

3.1 Calculating the Multipliers M_x

In order to calculate the general multipliers M_x , we first need to calculate another word difference machine D_2 which is obtained from our earlier machine D_1 as follows (the following is a simplified description of the construction of D_2):

Let I be the set of all state labels $\tau \in S(D_1)$, and let $S(D_2) = I \cup I^{-1} \cup A \cup \{\epsilon\}$ (remembering to reduce elements in this set w.r.t. our current rewrite system and then eliminate duplicates). For each $\tau \in S(D_2)$, we test to see if the expression $x^{-1}\tau y$ simplifies to give a state label in $S(D_2)$ for each possible pair (x, y) (in our example the 24 pairs listed in the previous seminar). If this is the case then this defines a transition. For example, if $\tau = e$, we have $\tau^1 = \tau^{(a,a)} = a^{-1}ea \rightarrow e$, $\tau^2 = a^{-1}b \rightarrow Ab \rightarrow ba$, etc.

Once we have the word difference machine D_2 , the method used to construct the general multiplier automata M_x is as follows: we have state set $S(W) \times S(W) \times S(D_2)$, initial state $(\sigma_0(W), \sigma_0(W), \sigma_0(D_2))$ ($\sigma_0(X)$ = initial state of fsa X), and transition $(\sigma_1, \sigma_2, \tau)^{(x,y)} = (\sigma_1^x, \sigma_2^y, \tau^{(x,y)})$, provided all three components are defined.

The M_x for different x differ from each other only in their accept states: the accept states of M_x are all $(\sigma_1, \sigma_2, \tau)$ such that $\sigma_1, \sigma_2 \in \mathcal{A}(W)$ and $\tau \equiv x$. However, once the M_x have been constructed they are immediately minimised, after which they no longer have the same sets of states, and are no longer necessarily word-difference automata.

Now the construction of the automaton D_2 and the automata M_x is something best left to a computer, even in a simple example — in our S_3 example, the machine D_2 has $6 \times 24 = 144$ transitions alone (with $S(D_2) = S(D_1)$). We can use the computer program *kbmag* to do our calculations for us, and the automata it provides as output are written to disc using the format discussed in Seminar 2. Incidentally, we have already seen some of the automata the *kbmag* program provides in the S_3 example in Seminar 1: the machines W and M_a seen in Seminar 1 were the ones provided by the *kbmag* program.

3.2 Testing the Automata

Up until now, we have discussed the construction of the automata W and M_x required to form part of an automatic structure for a group without worrying if the techniques we are using are correct or indeed provide correct output. We shall now introduce a set of axioms that a set of automata must satisfy in order to form part of an automatic structure for a group. We shall then go on to describe how *kbmag* verifies these axioms for its output and briefly describe what happens when one or more of the axioms are failed.

Note that the following axioms are taken directly from Section 6.1 of the book “Word Processing in Groups” by Epstein et. al., 1992.

3.2.1 Axiom Checking

Let G be a finitely presented group with presentation $\langle A/R \rangle$, and let $\langle A'/R' \rangle$ be the associated semigroup presentation for G . Suppose we are given a finite state automaton W over A' and, for each $x \in A'$, a finite state automaton M_x over (A', A') . These data form an automatic structure for G , with $L(W)$ prefix closed and consisting of unique representatives for the elements of G , if and only if the following conditions hold:

1. If $(v, w) \in L(M_x)$ for some $x \in A'$, then $v, w \in L(W)$.
2. If $(v, w) \in L(M_x)$, then $\overline{vx} = \overline{w} \in G$.
3. $L(W)$ is non-empty.
4. Let $v \in A^*$ and $x \in A$. If $vx \in L(W)$, then $(v, vx) \in L(M_x)$.
5. Let $(\epsilon, u) \in R'$ be a relation, where $u = x_1 \dots x_n$. Then two accepted strings $w_0, w_n \in L(W)$ are equal if and only if there exist accepted strings $w_1, \dots, w_{n-1} \in L(W)$ with $(w_{i-1}, w_i) \in L(M_{x_i})$ for $1 \leq i \leq n$.

The proof of the above appears in the book, but notice that the axioms essentially follow directly from the definition of an automatic group, especially the first two axioms.

A note should be made here that the above axioms only apply if the length-lex (or *ShortLex*) ordering is used. This allows us to use the above axioms instead of a set of more general axioms given in section 5.1 of the book (the simplification comes from the fact that using the length-lex ordering provides a prefix-closed language with the uniqueness property). What this means is that the computer program *kbmag* can only deal with groups using the length-lex ordering, and so some of the generality of the theory is lost — in particular, using the length-lex ordering ensures that the empty string maps on to the identity element of the group.

3.2.2 Axiom Checking in *kbmag*

So how do we verify that the automata that we have constructed using the techniques discussed in these seminars satisfy the above axioms, i.e. how does *kbmag* check the axioms? According to the book, the algorithms used in constructing W and the M_x given a starting group presentation ensure that the first four axioms are automatically satisfied, i.e. we carefully construct the automata so that these axioms are satisfied. For example, the third axiom is always satisfied because we specify that D_1 always has an accept state (the initial state), and so W will always accept the empty word and so $L(W)$ will never be empty.

Therefore, all that is left to do is to check the fifth axiom, and according to the paper “The Warwick Automatic Groups Software” this is checked in three stages:

- During the construction of the M_x , check that W is not accepting two words for the same group element;
- Check that the automata $W \wedge E(M_x)'$ all have empty accepted language;
- For each relator r from the set of defining relators \mathcal{R} of the group G , check that each composite multiplier M_r has language equal to the language of the equality recogniser M_ϵ .

Stage 1. It sometimes happens that during the construction of the automata M_x , we find distinct words $u, v \in A$, both accepted by W , for which $u = v$ in G . This means that W is accepting two words for the same group element, and so W cannot be correct. We must therefore restart the Knuth-Bendix procedure as we must have terminated it too early.

Stage 2. Suppose that W is correct but M_x is not correct, for some $x \in A$. From the construction of M_x , it is clear that, if M_x accepts $(u, v)^+$, then W accepts u and v , and $ux = v$ in G . Thus, if M_x is not correct for some $x \in A$, there exist words u and v in A^* such that W accepts u and v and $ux = v$ in G , but M_x does not accept $(u, v)^+$. But then, since v is the unique word accepted by W that is equal to ux in G , M_x cannot accept $(u, w)^+$ for any $w \in A^*$. We test for this by constructing the automata $W \wedge E(M_x)'$ for each $x \in A$. If all of these automata have empty accepted language and W is correct, then the M_x must also be correct, and we proceed to Stage 3 for the final verification. Otherwise, we find some explicit words u which are accepted by some $W \wedge E(M_x)'$, and we let v be the reduction of ux (using D_1); then W accepts v and $ux = v$ in G . Since M_x does not accept (u, v) , there must be prefixes $p(u)$ and $p(v)$ of u and v of the same length, such that $p(u)^{-1}p(v) \notin \mathcal{S}(D_2)$. We therefore adjoin all such elements and their inverses to $\mathcal{S}(D_2)$, and then recalculate D_2 itself.

Remark 3.1 Stage 2 above is an example of where the construction of an automatic structure can help in completing the Knuth-Bendix procedure. If we need to recalculate D_2 in the above, this means that there were some crucial missing word differences which should have been found using the Knuth-Bendix procedure but were not, word differences which could take many iterations of the

Knuth-Bendix procedure to find. However, word differences which did not arise because of the absence of some rewrite rules have now been found using our checks, and so we can use this information to our advantage.

Stage 3. In this final stage, the composite multipliers required are constructed using the following logical automaton operation: If Z_1 and Z_2 are any two two-variable automata with the same input alphabet $A^+ \times A^+$, then we define their composite to be the two-variable automaton with language

$$\{(u, v)^+ \mid \exists w \in A^+ : (u, w)^+ \in L(Z_1) \text{ and } (w, v)^+ \in L(Z_2)\}.$$

The composite operation is easily seen to be associative, and so we can define M_u for any $u \in A^*$ by repeated application. Therefore, we can calculate all the M_r for all relators r as required, and test the equality $L(M_r) = L(M_\epsilon)$ using the finite state automata operations of *kbmag*. If the equality is failed for one or more relators, this means that the collection of automata we have calculated do not form part of an automatic structure for the presentation of the group G we were given, and so we must go back to the Knuth-Bendix procedure and start again, letting the Knuth-Bendix procedure run for a longer time during this next iteration.

Remark 3.2 Since the operation of building a composite multiplier involves making a non-deterministic automata deterministic, it potentially involves an exponential increase in the number of states at each step. Therefore, this final stage is often the most space and time demanding part of the entire algorithm. To try to make the algorithm more efficient, it is usual for a long relator r to be split into two or more pieces and the composite multiplier is then built up bit by bit.

3.3 Conclusions

We have now gone through the entire process of constructing an automatic structure for an automatic group: interrupt the Knuth-Bendix procedure at a suitable stage, construct word-difference machines from the rewrite rules obtained from which we can construct test automata W and M_x , and finally test the automata to make sure that they satisfy a list of axioms which govern when and when not a set of automata form an automatic structure for a given presentation of a group G . It is clear that automatic structures are important because we can solve the word problem quickly in an automatic group, and the process of constructing an automatic structure also helps us in completing the Knuth-Bendix procedure — witness the quick performance of *kbmag* when it calculates complete rewrite systems using the Knuth-Bendix procedure and some automata.

3.4 Automatic Gröbner Bases?

Since the construction of complete rewrite systems using the Knuth-Bendix procedure and the construction of Gröbner Bases using Buchberger's algorithm are closely related, the question arises as to

whether there exists such things as Automatic Gröbner Bases to match with the existence of Automatic Groups. In the Knuth-Bendix procedure, the goal is to provide a complete rewrite system so that the word problem can be solved, i.e. given an arbitrary word w in the generators of some group G , is w equal to the identity word or not? When we interrupt the Knuth-Bendix procedure and attempt to construct an automatic structure, we are attempting to provide tools so that the word problem can be solved in quadratic time using the automatic structure, which is a collection of finite state automata. In Buchberger's algorithm, the goal is to provide a set of polynomials forming a Gröbner basis for an ideal I so that the *ideal membership problem* can be solved, i.e. given an arbitrary polynomial in the variables used in I , is the polynomial a member of the ideal I or not? The natural question is to ascertain whether there is an algorithm that interrupts Buchberger's algorithm at a suitable time and constructs an 'automatic structure' for the ideal I (finite state automata? petri nets?) enabling the ideal membership problem to be solved quickly (in polynomial time?) If there is, a new notion of the class of automatic Gröbner bases would be introduced, i.e. those Gröbner bases whose ideal membership problem can be solved quickly. Perhaps this is an area that should be investigated in more detail.

References

1. D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M. S. Peterson, and W. P. Thurston, *Word Processing in Groups*, Jones and Bartlett, 1992.
2. D. F. Holt, *The Warwick Automatic Groups Software*, in *Geometrical and Computational Perspectives on Infinite Groups*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science **25** (1996), 69-82.
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*, J. Symbolic Computation **12** (1991), 397-414.