J. Symbolic Computation (1990) **9**, 725–733

Collection from the Left

M. R. VAUGHAN-LEE

Christ Church, University of Oxford, Oxford, UK

Dedicated to Tim Wall on the occasion of his 65th birthday

(*Received* 7 *April* 1988)

The heart of the nilpotent quotient algorithm for computing in finite p-groups is a collection algorithm for collecting semigroup words on the generators of the group into normal form. In applications of the nilpotent quotient algorithm almost all the computing time is spent doing collections, and so very sophisticated collection algorithms have been developed. A number of researchers recently have started investigating a variant of the algorithm known as collection from the left. A version of collection from the left is described in this article. It is designed as an alternative to the Havas–Nicholson algorithm for collection from the right which is incorporated in the Canberra version of the nilpotent quotient algorithm. Indications are that for many applications it runs faster than the Havas–Nicholson algorithm.

1. Introduction

The design of efficient multiplication algorithms is of central importance in computational group theory. Given two elements of a group G, how do we compute their product? A multiplication algorithm is defined for G if:

(a) a normal form is defined for the elements of G; and
(b) there is an algorithm which, given two elements in normal form, computes the normal form of their product.

If a group G is given as a group of permutations or as a group of matrices then a multiplication algorithm is clearly available. If G is a finitely presented group which is known to be finite then, provided its order is not too large, the Todd–Coxeter algorithm can be used to construct its coset table over the trivial subgroup (or over any core free subgroup). This coset table can be used as the basis of a multiplication algorithm. Finitely presented groups of much larger order can be handled with the nilpotent quotient algorithm, provided they are known to be finite p-groups. In this case the nilpotent quotient algorithm can be used to produce a power-commutator presentation (PCP) for G. A PCP consists of a set of generators $x_1, x_2, \ldots, x_n$ and a set of relations of the form

$$x_i^p = w_i \quad (1 \leqslant i \leqslant n),$$

$$[x_j, x_i] = w_{ij} \quad (1 \leqslant i < j \leqslant n),$$

where

$$w_i = x_{i+1}^{\alpha(i,i+1)} x_{i+2}^{\alpha(i,i+2)} \ldots x_n^{\alpha(i,n)}$$

for some $\alpha(i,j)$ with $0 \leqslant \alpha(i,j) < p$, and where

$$w_{ij} = x_{j+1}^{\alpha(i,j,j+1)} x_{j+2}^{\alpha(i,j,j+2)} \ldots x_n^{\alpha(i,j,n)}$$

0747–7171/90/050725+17 $03.00/0

for some $\alpha(i,j,k)$ with $0 \leqslant \alpha(i,j,k) < p$. If G has a presentation of this form then G has order dividing p^n and every element of G can be expressed in normal form

$$x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} \quad (0 \leqslant \alpha_i < p).$$

The presentation is consistent if the order of G is precisely p^n, and in this case every element of G has a unique expression in normal form. Every finite p-group G has a consistent PCP. If w is a semigroup word on the generators of G (such as the product of two normal words) then a collection process can be used to reduce w to normal form. If w is not already in normal form then it must contain a minimal non-normal subword x_i^p or $x_j x_i$ with $j > i$. In the first case we replace x_i^p in w by w_i, and in the second case we replace $x_j x_i$ in w by $x_i x_j w_{ij}$. We continue iterating this procedure until a normal word is obtained. (The process does terminate!) Usually w contains more than one minimal non-normal subword, and the efficiency of any particular collection algorithm depends vitally on which one is selected. In P. Hall's collection process described in M. Hall [1959, section 11], the leftmost minimal non-normal subword involving x_1 is selected, and, if there are none of these (so that *all* the x_1 have been collected), then the minimal non-normal subword involving x_2 which is closest to the left is selected, and so on. It is easy to see that this process terminates, but it leads to a hopelessly inefficient algorithm, both in terms of the number of iterations required, and in terms of the amount of storage required. A major breakthrough in collection algorithms was achieved when it was realised that selection of the rightmost minimal non-normal subword (collection from the right) leads to much faster collection times and much smaller storage requirements than the Hall process. A number of authors have implemented variations of collection from the right. The current version of the programming language Cayley embodies those of Felsch (1976) and Havas & Nicholson (1976). [See Cannon (1984) for a description of Cayley.] Recently, a number of researchers have started to investigate collection from the left (selection of the leftmost minimal non-normal subword). Collection from the left requires more storage than collection from the right, but indications are that far quicker collection times can be achieved. The nilpotent quotient algorithm used by Cayley is the Canberra version developed by Newman (1976), and incorporates the Havas–Nicholson algorithm for collection from the right. The algorithm outlined below for collection from the left was substituted for the Havas–Nicholson algorithm in Cayley, and a number of time tests were performed. The largest finite quotients of the six groups given in the table below were computed, both with collection from the left, and with collection from the right. Considerable time savings were achieved.

Here B(r, n) is the r generator Burnside group of exponent n, and B(r, n; c) is the class c quotient of B(r, n). The group G is the largest finite two-generator group of exponent 8 where one generator has order 2 and the other generator has order 4. The group H is the

Group	Order	Class	Time in seconds to compute the group	
			from the left	from the right
B(3, 4)	2^{69}	7	26	41
B(2, 5)	5^{34}	12	99	567
B(2, 7; 12)	7^{408}	12	5040	73309
B(3, 5; 9)	5^{916}	9	13621	49644
G	2^{205}	26	28151	237887
H	2^{55}	50	347	1414

class 50 quotient of the space group described by Felsch & Neubüser (1976), which turned out to be a counter-example to the class breadth conjecture. Note that we are not claiming that B(2, 5) has order 5^{34}, only that its largest finite quotient has order 5^{34}.

The Algorithm

We consider a PCP on a set of generators $x_1, x_2, \ldots, x_n$ with relations $x_i^p = w_i$ $(1 \leqslant i \leqslant n)$ and $[x_j, x_i] = w_{ij}$ $(1 \leqslant i < j \leqslant n)$, as described above. In the Canberra version of the nilpotent quotient algorithm a normal word on the generators can be stored in one of two ways: either as an exponent vector $(a_1, a_2, \ldots, a_n)$ with $0 \leqslant a_i < p$, or as a string of generator-exponent pairs $(i, a), (j, b), \ldots, (k, c)$ with $1 \leqslant i < j < \ldots < k \leqslant n$ and $0 < a, b, \ldots, c < p$. The exponent vector $(a_1, a_2, \ldots, a_n)$ represents the normal word $x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n}$, and the entries $a_1, a_2, \ldots, a_n$ are stored in n successive locations of computer memory. The string of generator-exponent pairs $(i, a), (j, b), \ldots, (k, c)$ represents the normal word $x_i^a x_j^b \ldots x_k^c$, and if there are s pairs in this string then these pairs are stored in s successive locations of computer memory as the integers $a \,.\, 2^{16} + i, b \,.\, 2^{16} + j, \ldots, c \,.\, 2^{16} + k$. The normal words w_i, w_{ij} are stored as strings of generator-exponent pairs (when they are non-trivial).

The input to the algorithm is an exponent vector representing a normal word u, and a string of generator-exponent pairs representing a normal word v. The algorithm returns an exponent vector representing the product uv. During the collection process the original exponent vector representing u is modified, and pointers to a number of strings of generator-exponent pairs are stored on a stack. The stack is "three wide". Each point of the stack stores a triple of integers (str, len, exp). If str < 0 then −str is the base address of a string of generator-exponent pairs of length len. If this string represents the normal word v then (str, len, exp) represents v^{exp}. If str > 0 then str must lie in the range $1 \leqslant \text{str} \leqslant n$, and (str, len, exp) represents $x_{\text{str}}^{\text{exp}}$. (In this case len is irrelevant.) The depth of the stack is represented by a stack pointer (sp). Thus, at any given point during the collection process the exponent vector represents a normal word w, and the stack represents sp powers of normal words $v_1^a, v_2^b, \ldots, v_{\text{sp}}^c$. Together, they represent the product $w v_{\text{sp}}^c \ldots v_2^b v_1^a$. Initially, the single input string is stored on the stack (and sp = 1). The collection process is complete when the stack is empty (sp = 0). We denote the gth entry of the exponent vector by expvec(g).

First we describe the most basic form of collection from the left, without any of the frills which are incorporated in the algorithm. The step numbers given in this description correspond to step numbers in the full algorithm. Before we define the algorithm itself we define a procedure for placing pointers to two types of words on the stack.

Procedure push(w^a)
If w is a generator x_i then
  Let str = i, len = 0, exp = a.
  Let sp = sp + 1.
  Let stack(sp) = (str, len, exp).
Endif
If w is represented by a non-empty string of generator-exponent pairs then
  Let k be the base address of the string of generator-exponent pairs representing w and let s be its length.
  Let str = $-k$, len = s, exp = a.

Let sp = sp + 1.
Let stack(sp) = (str, len, exp).
Endif
End procedure push.

Now we define the collection algorithm.

Step (0). (Initialise collector.)
Let k be the base address and let s be the length of the initial input string of generator-exponent pairs.
Let str = $-k$, len = s, exp = 1.
Let sp = 1, stack(sp) = (str, len, exp).

Step (1). (Process next triple on the stack.)
If sp = 0 return.
Let (str, len, exp) = stack(sp), sp = sp − 1.
If str > 0 then
  (The triple (str, len, exp) represents $x_{\text{str}}^{\text{exp}}$.)
  If exp > 1 then push($x_{\text{str}}^{\text{exp}-1}$).
  Let i = str.
Else
  (The triple (str, len, exp) represents a string of generator-exponent pairs of length len. In this basic form of the algorithm exp must equal 1.)
  Let (i, a) be the first generator-exponent pair of the string.
  (Place remainder of the string on the stack.)
  If len > 1 let sp = sp + 1, stack(sp) = (str − 1, len − 1, exp).
  If $a > 1$ push(x_i^{a-1}).
Endif

Step (3). (Collect generator x_i, scanning exponent vector from the right-hand end towards the left.)
For $g = n$ down to $i+1$ do
  Let b = expvec(g).
  If $b \neq 0$ then
    Let expvec(g) = 0.
    If w_{ig} is trivial then
      push(x_g^b).
    Else
      For j from 1 to b do
        push(x_g).
        push(w_{ig}).
      End
    Endif
  Endif
End

Step (4). (Add 1 to ith entry of exponent vector, reduce it mod p, and add x_i^p to the exponent vector if necessary.)

Let expvec(i) = expvec(i) + 1.
If expvec(i) = p then
  Let expvec(i) = 0.
  If w_i is non-trivial then
    For each (j, b) in the string of generator-exponent pairs representing w_i let expvec(j) = b.
  Endif
Endif
Go to Step (1).

The maximum stack depth required for this algorithm is $(p-1)n(c+1)$, where c is the class of the group. This compares with a maximum stack depth of c for the Havas–Nicholson algorithm for collection from the right. It would be possible to reduce the maximum stack depth from $(p-1)n(c+1)$ to nc by modifying Step (3) above. At Step (3), $(x_i w_{ig})^b$ is stored on $2b$ levels of the stack (with b at most $p-1$), even though it would be perfectly possible to store it on a single level of the stack either by widening the stack or by encoding more information in (str, len, exp). However, the gain in doing this does not appear to be very significant for two reasons. Firstly, no matter how $(x_i w_{ig})^b$ is stored on the stack, it will still need to be processed in $2b$ separate steps corresponding to the $2b$ levels of the stack used in the algorithm described above. Secondly, even though $(p-1)n(c+1)$ grows quadratically with n (taking n as an upper bound for c), the amount of space required to store the PCP grows with n^3, so that as n increases the amount of space required to store the stack becomes small relative to the amount of space required to store the PCP. For example, the Canberra nilpotent quotient algorithm uses 1280 words to store the PCP of the largest finite quotient of B(2, 5) and $(p-1)n(c+1) = 1768$ for this group. But the Canberra nilpotent quotient algorithm uses 132 663 words to store the PCP of B(3, 5; 9), and for this group $(p-1)n(c+1) = 36\,640$. It is known that the largest finite quotient of B(3, 5) has order at most 5^{2282} and class at most 17, so that $(p-1)n(c+1)$ is at most 164 304 in this case. But detailed estimates indicate that 5 600 000 words would be needed to store the PCP of this group.

Weighted Presentations

The Canberra nilpotent quotient algorithm produces weighted power-commutator presentations. That is, each generator x_i is assigned a weight wt(i), with the following properties.

(a) $1 = \text{wt}(1) \leqslant \text{wt}(2) \leqslant \ldots \leqslant \text{wt}(n) = c$.
(b) The word w_i representing x_i^p only involves generators x_k such that $\text{wt}(k) > \text{wt}(i)$. (That is $\alpha(i, k) \neq 0$ implies $\text{wt}(k) > \text{wt}(i)$.)
(c) The word w_{ij} representing $[x_j, x_i]$ only involves generators x_k such that $\text{wt}(k) \geqslant \text{wt}(i) + \text{wt}(j)$.

We exploit these weights in a similar way to the way in which the Havas–Nicholson algorithm exploits them.

The generator x_i commutes with all generators with weight greater than $c - \text{wt}(i)$. Furthermore, all powers and commutators arising in collecting x_i also commute with these generators. So when collecting x_i we can by-pass all entries in the exponent vector corresponding to generators with weight greater than $c - \text{wt}(i)$, and there is no need to stack generator-exponent pairs corresponding to these entries [see Step (2) and Step (6) below].

If $\mathrm{wt}(i) > c/2$ then collecting x_i cannot generate any commutators and so x_i^a can be collected by adding a to the ith entry of the exponent vector [see Step (4) below].

If $v = x_i^a x_j^b \ldots x_k^d$ is a word in normal form, and if $\mathrm{wt}(i) > c/2$, then for any integer e, $v^e = x_i^{ae} x_j^{be} \ldots x_k^{de}$ and v^e can be collected by adding $ae, be, \ldots, de$ to the ith, jth, …, kth entries of the exponent vector [see Step (5) and Step (6) below].

If $\mathrm{wt}(i) > c/3$ then any commutator arising in collecting x_i will commute with all generators x_j such that $j > i$. Furthermore, if $j > i$ then $[x_j^b, x_i^a] = [x_j, x_i]^{ab}$ and $[x_j, x_i]^{ab}$ can be collected as in the paragraph above. So x_i^a can be collected without stacking entries in the exponent vector, and without stacking commutators. Care is needed, however, if adding a to the ith entry of the exponent vector increases this entry to a value greater than or equal to p [see Step (6) below].

If $2\mathrm{wt}(j) + \mathrm{wt}(i) > c$ then

$$[x_j^b, x_i^a] = [x_j, x_i]^{ba} . [x_j, x_i, x_i]^{b\binom{a}{2}} . \ldots . [x_j, x_i, \ldots, x_i]^{b\binom{a}{a}}.$$

[See Step (2) below.]

We now define the full algorithm. The procedure push(w^a) is as defined above.

Step (0). (Initialise collector.)
Let k be the base address and let s be the length of the initial input string of generator-exponent pairs.
Let str $= -k$, len $= s$, exp $= 1$.
Let sp $= 1$, stack(sp) $=$ (str, len, exp).

Step (1). (Process next triple on the stack.)
If sp $= 0$ return.
Let (str, len, exp) $=$ stack(sp), sp $=$ sp $- 1$.
If str > 0 then
  (The triple (str, len, exp) represents $x_{\mathrm{str}}^{\mathrm{exp}}$.)
  Let $i =$ str, $a =$ exp.
  If $2\mathrm{wt}(i) > c$ then go to Step (4).

Else
  (The triple (str, len, exp) represents a string of generator-exponent pairs of length len raised to the power exp.)
  Let (i, a) be the first generator-exponent pair of the string.
  If $2\mathrm{wt}(i) > c$ then go to Step (5).
  (Place remainder of the string on the stack. In this case exp $= 1$.)
  If len > 1 let sp $=$ sp $+ 1$, stack(sp) $=$ (str $- 1$, len $- 1$, exp).
Endif
If $3\mathrm{wt}(i) > c$ go to Step (6).

Step (2). (Combinatorial collection of x_i^a.)
Let k be minimal subject to $2\mathrm{wt}(k) + \mathrm{wt}(i) > c$, and let j be maximal subject to $\mathrm{wt}(j) + \mathrm{wt}(i) \leqslant c$. (Note that $k \leqslant j$.)

```
For r from 1 to a do
  For g from j down to k do
    Let b = expvec(g).
    If b ≠ 0 then
      Let expvec(g) = 0.
      push(x_g^b).
      If w_ig is non-trivial then
        For each (t, d) in the string of generator-exponent pairs representing w_ig let
        expvec(t) = expvec(t) + bd(a−r+1)/r.
      Endif
    Endif
  End
End
For g from j down to k do
  (Stack up relevant stretch of exponent vector.)
  Let b = expvec(g).
  If b ≠ 0 then
    Let expvec(g) = 0.
    push(x_g^b).
  Endif
End
For g from j+1 to n do
  (Tidy up powers in unstacked stretch of exponent vector.)
  Let b = expvec(g).
  If b ≥ p then
    Let u, v be integers with 0 ≤ v < p and b = up+v.
    Let expvec(g) = v.
    If w_g is non-trivial then
      For each (t, d) in the string of generator-exponent pairs representing w_g let
      expvec(t) = expvec(t) + du.
    Endif
  Endif
End
```

Step (*3*). (Ordinary collection of x_i^a. Scan exponent vector from the $k-1$ position towards the left.)

```
For g = k−1 down to i+1 do
  Let b = expvec(g).
  If b ≠ 0 then
    Let expvec(g) = 0.
    If w_ig is trivial then
      push(x_g^b).
    Else
      If a > 1 then
        push(x_i^(a−1)).
        Let a = 1.
      Endif
      For j from 1 to b do
```

```
        push(x_g).
        push(w_ig)
      End
    Endif
  Endif
End
```

Step (4). (Add a to ith entry of exponent vector, reduce it mod p, and stack a power of x_i^p if necessary.)

```
Let b = expvec(i).
Let u, v be integers with 0 ≤ v < p and a + b = up + v.
Let expvec(i) = v.
If u > 0 and w_i is non-trivial push(w_i^u).
Go to Step (1).
```

Step (5). (Add the word represented by (str, len, exp) to the exponent vector.)

```
For each (i, a) in the string of generator-exponent pairs with base address —str and length
len do
  Let b = expvec(i).
  Let u, v be integers with 0 ≤ v < p and up + v = b + a.exp.
  Let expvec(i) = v.
  If u > 0 and w_i is non-trivial push(w_i^u).
End
Go to Step (1).
```

Step (6). (Collect x_i^a without stacking entries in exponent vector.)

```
Let k be maximal subject to wt(i) + wt(k) ≤ c.
For g from i+1 to k do
  Let b = expvec(g).
  If b ≠ 0 and w_ig is non-trivial then
    For each (t, d) in the string of generator-exponent pairs representing w_ig do
      Let e = expvec(t).
      Let u, v be integers with 0 ≤ v < p and pu + v = e + abd.
      Let expvec(t) = v.
      If u > 0 and w_t is non-trivial push(w_t^u).
    End
  Endif
End
Let b = expvec(i).
Let u, v be integers with 0 ≤ v < p and pu + v = a + b.
Let expvec(i) = v.
If u > 0 and w_i is non-trivial then
  (Stack up entries in the exponent vector.)
  For g from k down to i+1 do
    Let b = expvec(g).
    If b ≠ 0 then
      Let expvec(g) = 0.
      push(x_g^b).
```

Endif
End
push(w_i^u).
Endif
Go to Step (1).

There are a number of variations on this algorithm which present themselves. For example, rather than applying combinatorial collection to x_i^a at Step (2) we could place x_i^{a-1} on the stack at Step (1) and then only apply combinatorial collection to x_i. This would simplify the programming and could be done with only two passes: one to add in commutators (without stacking any entries), and one to stack up entries in the exponent vector. The advantage of applying combinatorial collection to x_i^a rather than to x_i only accrues if no commutators are generated in Step (3). For if commutators are generated in Step (3) then we have to place x_i^{a-1} on the stack in Step (3) and then later on we have to apply combinatorial collection to x_i^{a-1} all over again.

Steps (5) and (6) could also be altered. Instead of reducing the entries in the exponent vector modulo p and placing pth powers on the stack as we go along, we could tidy up the powers at the end of the step, in the same way as the powers are tidied up at the end of Step (2).

My thanks to Leonard Soicher at Queen Mary College, London, whose investigations into collection from the left prompted me to write this algorithm, to Tim Nicholson at the University of Sydney who made a number of helpful suggestions, and especially to John Cannon who offered advice and vast amounts of CPU-time. My thanks also to the Mathematics Department at the University of Sydney for hospitality and financial support while I was researching this article.

References

Cannon, J. J. (1984). An Introduction to the Group Theory Language, Cayley. *Computational Group Theory* (Atkinson, M. D., ed.) London: Academic Press, pp. 145–183.

Felsch, V. (1976). A machine independent implementation of a collection algorithm for the multiplication of group elements. In: *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*. New York: Assoc. Comput. Mach., pp. 159–166.

Felsch, V., Neubüser, J. (1976). An algorithm for the computation of conjugacy classes and centralizers in p-groups. *Lecture Notes in Computer Science* **72**. Berlin: Springer, pp. 452–465.

Hall, M. (1959). *The Theory of Groups*. New York: Macmillan.

Havas, G., Nicholson, T. (1976). Collection. In: *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*. New York: Assoc. Comput. Mach., pp. 9–14.

Newman, M. F. (1976). Calculating presentations for certain kinds of quotient groups. In: *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*. New York: Assoc. Comput. Mach., pp. 2–8.