Collection

George Havas and Tim Nicholson

Department of Mathematics,
Institute of Advanced Studies,
Australian National University,
Canberra.

Department of Pure Mathematics,
University of Sydney,
Sydney.

Abstract. Collection processes have been the basis of group investigations by many people, some using hand calculation, some machine calculation. We describe a collection process which is specially efficient in the context of nilpotent quotient algorithm programs. The principles underlying our collection process are applicable in general.

1. Introduction. A collection process is a method for converting a word (in a group) into a normal form. P. Hall [13] introduced the notion of "the collecting process" and a definition of a collection process in the context of nilpotent quotient algorithms is given by M.F. Newman [20]. Since the principles underlying our collection algorithm are generally applicable, especially for machine implementation, we mention all machine implementations of collection processes of which we are aware. It is likely that machine implementations which do not apply our principles could be significantly improved.

The first machine implementation was by H. Felsch [9] with a program for collecting words of the form $(ab)^n$. Then J.M. Campbell and W.J. Lamberth [2] wrote a general purpose program for collecting words in free nilpotent groups of finite class and rank.

More recently K. Dałek [6], [7], [8] and E. Czyzo [3] have determined some exponents in various Hall formulas by combinatorial means (ultimately based on collection) and by direct collection. As far as direct collection is concerned they use programs described by E. Czyzo [4], [5].

From another point of view, H. Jürgensen [15] described a method for calculation with elements of a finite group given by generators and defining relations of special forms (based on work of J. Neubüser [19] and W. Lindenberg [16], [17]). He gave methods for multiplying and inverting such elements using what in fact is a collection process. V. Felsch [10] describes a new implementation of these methods in machine independent FORTRAN.

T.W. Sag and J.W. Wamsley [21] have described a method for computing the Schur multiplier of a finite supersoluble group. Their method involves multiplication of group elements which they perform by a collection process.

I.D. Macdonald [18] and J.W. Wamsley [22] have developed algorithms which, given a presentation of a group G and a positive integer c, construct the largest nilpotent of class c quotient group of G. Computer programs implementing these algorithms for specific p-quotients have been developed by Macdonald and Wamsley, and later by Alford, Havas and Newman [1], W. Felsch [11] and Havas-Newman (see Newman [20]). Collection plays a fundamental role in each of these programs.

2. Two methods of collection. The collecting process of P. Hall is amplified in M. Hall [12, §11.1]. In this kind of collecting process the basic step is the selection of the leftmost occurrence of the least uncollected letter and the collection of this letter relative to the letter on its left. We call this collection to the left.

Neubüser was the first to describe the effective collection process now described by V. Felsch. In this kind of collecting process the basic step is the selection of the rightmost uncollected subword and the collection of this subword. We call this collection from the right.

Each collecting process comprises the repeated performance of its basic step. In general neither collecting process terminates. To ensure termination practical collections are done modulo some term in a central series.

H. Felsch, Campbell and Lamberth, Czyzo, Macdonald and W. Felsch have all implemented variations of collection to the left. Neubüser, Lindenberg, Jürgensen, Sag and Wamsley, Wamsley, V. Felsch, Havas and Nicholson have all implemented variations of collection from the right.

3. Comparison. Our implementation of collection is strongly influenced by the work of Wamsley. He asserts that in practice collection from the right is better than collection to the left in the sense that it generally involves fewer steps and shorter intermediate words. We support this claim.

In the context of collection relative to a prime-power-commutator presentation on n generators with prime p, Newman has shown that there is a useful bound on the length of any word occurring at any stage, using collection from the right. Under the above conditions, if the original word to be collected has length l, then in the process of collection from the right no word of length greater than $l - 2 + (p-1)(2n-1)$ ever occurs. In general this bound is dramatically exceeded in collections to the left.

The claim that collection from the right is

better than collection to the left is illustrated by two different nilpotent quotient algorithm programs. (In significant applications more than 90% of computer time is spent doing collections.) The Canberra version, incorporating the collection algorithm described in this paper, took 23.67 seconds on a Univac 1108 to calculate the complete nilpotent quotient of a group of order 2^{21} and class 9, given a defining presentation including long eighth power relations. Another nilpotent quotient algorithm program, using collection to the left, given 10 minutes on a comparable machine, ran out of time while collecting the eighth powers at class 3. It did not complete the calculation of class 3. (Note that combinatorial collection as described in §4 has no beneficial influence for 2-groups, so this is a valid comparison.)

Consider the following presentation for the class 3 quotient of the above group (for abbreviation we represent a_i by i and omit trivial squares and commutators)

$$G = \langle 1, 2, 3, 4, 5, 6 \mid 2^2 = 4,\ 3^2 = 5,\ 4^2 = 5;\ [2, 1] = 3,\ [3, 1] = 5,\ [3, 2] = 6,\ [4, 1] = 56 \rangle .$$

To give yourself a feeling for collection, collect $(12)^4$ to the left and from the right, with respect to G. Using the most basic forms of collection and counting each move or square elimination as a step, you end up with 182 steps using collection to the left and 16 steps using collection from the right. The longest word arising in the course of collection to the left is of length 27, while the longest word arising in collection from the right is of length 10. With increasing complexity of word to be collected and increasing complexity of the presentation the difference is magnified.

With collection to the left improved by eliminating squares when they arise, there are still 32 steps involved. Collecting $(12)^8$ to the left this improved way takes 251 steps, with longest word of length 35, as against 32 steps for collection from the right, with longest word of length 18.

From the above example it follows that a binary decomposition of the exponent would be worthwhile for collection of $(12)^8$. There is scope for investigation of how a word to be collected should be preprocessed.

4. Improvements. Lindenberg, Jürgensen and Wamsley observed that a major part of collection time was spent collecting expressions of the form yx^m and $y^n x^m$. Lindenberg and Jürgensen used presentations which include relations giving the values of $[y, x^m]$ for all necessary x, y and m, or alternatively computed these systematically prior to performing other collections. Wamsley went further in the computation of the 2 generator restricted Burnside group of exponent 5 [14] when he computed and stored the values of all commutators $[y^n, x^m]$.

The disadvantage of this kind of approach is that storage requirements are dramatically increased. For a p-quotient calculation storing additional tables multiplies the memory requirements by $(p-1)$ or $(p-1)^2$ depending on whether we store commutators of the form $[y, x^m]$ only or all commutators of the form $[y^n, x^m]$.

In our implementation we augment collection from the right by what we call combinatorial collection, to do collections of the above forms as quickly as possible in the absence of additional tables.

Consider yx^m where $y > x$, x has weight u, y has weight v. Then (by a combinatorial argument):

$$(1)\quad yx^m \equiv x^m y [y, x]^{\binom{m}{1}} [y, x, x]^{\binom{m}{2}} \cdots \cdots [y, \underbrace{x, \ldots, x}_{m}]^{\binom{m}{m}} \text{ modulo } \gamma_{2u+2v} .$$

Likewise:

$$(2)\quad y^n x^m \equiv x^m y^n [y, x]^{n\binom{m}{1}} [y, x, x]^{n\binom{m}{2}} \cdots \cdots [y, \underbrace{x, \ldots, x}_{m}]^{n\binom{m}{m}} \text{ modulo } \gamma_{u+2v} .$$

We use these formulas whenever applicable in our collection process. (It is possible to construct other formulas like (1) and (2) for calculating yx^m and $y^n x^m$ modulo different subgroups. We have not investigated these possibilities. Note that formula (1) is in fact correct modulo γ_{3u+2v}, but we only use it modulo γ_{2u+2v}. The reason for this is that if $[y, x] = \prod a_i^{\alpha_i}$ (in collected form) then we calculate $[y, x]^m$ by calculating $\prod a_i^{m\alpha_i}$ and $[y, x]^m \equiv \prod a_i^{m\alpha_i}$ modulo γ_{2u+2v}, but not necessarily modulo higher terms.)

To illustrate the significance of combinatorial collection consider the following timing tests, run with an early version of the Canberra nilpotent quotient algorithm program with collection from the right without combinatorial collection, then with the full collection process including combinatorial collection. The only difference between the two programs was the collection routine.

Let

$$G = \langle a, b \mid a^5 = b^5 = (ab)^5 = (a^2 b)^5 = (ab^2)^5 = 1 \rangle .$$

The computation of the maximal 5-quotient of class 7 (order 5^{26}) took 17.14 seconds without combinatorial collection and took 2.57 seconds with combinatorial collection. For a maximal class 7-group the computation to the end of class 10 (order 7^{11}) took 12.53 seconds without as against 2.67 seconds with combinatorial collection.

5. Algorithm description.

5.1. Introduction. This algorithm description is made in one of the contexts in which the algorithm has been implemented, that is in the context of the Canberra nilpotent quotient algorithm program. (For other contexts some modifications are appropriate, but the same principles apply. Note, for example,

that this collection algorithm handles positive words only, because negative exponents are processed elsewhere in the nilpotent quotient algorithm program.) The collection described here is collection relative to a prime-power-commutator presentation:

$$\Big\langle a_1, \ldots, a_n \mid a_i^p = \prod_{k>i} a_k^{\alpha(i,k)} \ , \quad 1 \le i \le n \ ;$$
$$[a_j, a_i] = \prod_{k>j} a_k^{\alpha(i,j,k)} \ , \quad 1 \le i < j \le n \Big\rangle \ .$$

First it is necessary to describe the data structures. (This description is consistent with the data structures used in the Canberra implementation of the nilpotent quotient algorithm.)

Each word to be collected and all commutators and powers of the prime-power-commutator presentation are stored as strings of generator-exponent pairs. A string has a base address (*string*) and length (l) . Strings are processed from the right, and l is decremented as the string is processed, so that the l-th generator-exponent pair is the pair actually processed.

The collected result is computed as a vector of exponents (*exponent*), one exponent for each generator of the presentation. In the course of collection from the right this vector represents a number of normal word segments (not a collected part as is obtained at the left in collection to the left).

Thus all words are represented effectively with generator-exponent pairs. However the algorithmic treatment of the words is as free group words, that is $\prod_i a_i^{\alpha_i}$ is treated as $\prod_i \underbrace{a_i \ldots a_i}_{\alpha_i}$.

In collection from the right we find the rightmost uncollected subword (in this context either $a_j a_i$ with $j > i$ or a_i^p) and collect it. To collect $a_j a_i$ we move a_j past a_i to get $a_i a_j [a_j, a_i]$. This can lead to a new uncollected subword to the right of the letter we are moving (a_j) . The new uncollected subword needs to be processed before we finish moving a_j to its final position.

Thus this collection can be considered a recursive process. We achieve recursion by using a stack (*stack*) on which we save "the state of the collector" before starting to move a new letter to the right of one which is already in motion.

This algorithm is economical with storage because in addition to the presentation itself and the string to be collected all that is required is a small number of working variables and the stack. Copies are not made of any of the strings and the depth of the stack is limited (by the class of the presentation, in collection relative to a power-commutator presentation).

5.2. Variables used. In this collection algorithm there are eight main working variables, in four pairs: *string*, l ; *ug*, *ue* ; *cg*, *ce* ; m, r . In the course of collection *string*, l represent the string being processed. *ug*, *ue* represent the uncollected generator being moved and the number of occurrences (i.e. exponent) of *ug* which remain to be moved from the l-th generator-exponent pair of *string*. *cg*, *ce* represent the position, relative to the exponent vector, of those occurrences of *ug* actually being moved (in the sense that those occurrences of *ug* precede *ce* occurrences of generator *cg*). The position represented is the start of the trailing normal word segment. m is the actual number of occurrences (multiplicity) of *ug* being moved and r is a divisor used in computing combinatorial coefficients for evaluation of m from formulas.

The stack is "eight wide", in the sense that each of these eight variables is stored to represent the state of the collector. The depth of the stack is represented by a stack pointer (*sp*) . The variables stored on the stack have the same significance as described above, except that *cg*, *ce*, m , and r refer to the unprocessed part of *string*, that is everything up to and including the l-th generator-exponent pair, bar those occurrences of *ug* already being moved.

The significance of the other variables is as follows: p is the prime of the prime-power-commutator presentation; *class* is the class of the presentation; n is the number of generators in the presentation; *splim* is the stack level at which the change over from ordinary to combinatorial collection is made; *noofugmoved* indicates how many occurrences of *ug* (out of the *ue* available) we are actually moving; *noofugmoved* = 0 indicates that we have not yet determined how many we are moving, *noofugmoved* = 1 indicates that we are moving one occurrence of *ug* , while *noofugmoved* = -1 indicates that we are moving all *ue* occurrences of *ug* . (It is at the changeover level, *sp* = *splim* , that *noofugmoved* matters. Below that level we are doing ordinary collection so that we are moving only one *ug* , above that level we are doing full combinatorial collection so that we are moving all *ue* occurrences of *ug* .)

5.3. Break up of the algorithm. The algorithm can be broken up into three parts: ordinary collection; evaluation of p-th powers; and combinatorial collection. To follow the algorithm it is probably easiest to work level by level.

The lowest level, ordinary collection (from the right) without evaluation of p-th powers, involves steps 1, 2, 3, 5, 11, 12, 13, 14, 15 and 16 only. The variables m, r, *splim* and *noofugmoved* can basically be ignored. For this case $m = r = 1$ always, let $p = \infty$, let *sp* = *splim* in steps 13 and 14 , and let *noofugmoved* = 1 .

For the next level, ordinary collection with evaluation of p-th powers, reset p and allow steps 17 and 18 . (Note that in practice evaluation of some p-th powers is delayed to the end, so that this is not literally collection from the right.)

For combinatorial collection first, for simplicity, ensure that *noofugmoved* is not -1 (by setting it to 1 in step (6)) and allow all steps of the algorithm subject to this condition. Finally remove this restriction and you have the full algorithm. Observe that once we commence combinatorial collection we continue combinatorially till we return to the changeover level (*sp* = *splim*) of the stack.

5.4. The algorithm.

(1) (Initiallize collector) *exponent* (*) ← already collected part; *string*, l ← word to premultiply the already collected part; $sp \leftarrow 0$; $cg \leftarrow 0$; $ce \leftarrow 1$ (the whole of *string* is in front of the exponent vector which corresponds to one normal word segment); $m \leftarrow 1$; $r \leftarrow 1$.

(1a) ug, $ue \leftarrow l$-th generator-exponent pair of *string*.

(2) (Save state of collector) $sp \leftarrow sp + 1$; *stack* $(sp) \leftarrow$ *string*, l, ug, ue, cg, ce, m, r ; *noofugmoved* $\leftarrow 0$ (note that we have not yet determined how many of the ue occurrences of ug we will move).

(3) (Process next letter in the trailing normal word segment) $ce \leftarrow ce - 1$;

(3a) if $ce > 0$ (at least one occurrence of cg is to the right of ug) go to (4); (3b) (no occurrence of cg is to the right of ug so proceed to the next generator-exponent pair of the trailing normal word segment) $cg \leftarrow cg + 1$; $ce \leftarrow$ *exponent* (cg) ; if $cg < ug$ (ug and cg are out of order provided cg actually occurs, i.e. provided $ce > 0$) go to (3a); else ($ug = cg$, i.e. ug has reached its correct position relative to the exponent vector) $splim \leftarrow sp$ (note that this stack level is the change over level from ordinary collection to combinatorial collection); (since ug has reached its correct position, ug is to be inserted into the exponent vector) go to (11).

(4) (We are going to move ug past cg . Check if we can switch to combinatorial collection.) If $2 \times (\text{weight}(ug)+\text{weight}(cg)) > class$ (combinatorial collection is applicable) go to (6).

(5) (Use ordinary collection) If $[ug, cg]$ is trivial go to (3b) (continue moving ug); (at this stage we have a new string $[ug, cg]$ to be inserted after $cg.ug$. The new string is not in its correct position. By simple recursion we process this new string and return to the current string when we have finished with the new string) *string*, $l \leftarrow [ug, cg]$; to to (1a).

(6) (Commence combinatorial collection) $splim \leftarrow sp$ (note that the current level is the changeover level).

(7) (Check if ug can be put into its correct position without creating any more commutators) If weight(ug) + weight(cg) > $class$ (ug commutes with everything following it, so we can put the appropriate number of occurrences of ug in their correct position) go to (11).

(8) (Use combinatorial collection) If $[ug, cg]$ is trivial go to (10b); (as in step (5) we have a new string $[ug, cg]$ to insert not in its correct position); if *noofugmoved* = 0 (we have not yet determined how many occurrences of ug we can move together, so we do so now) then {if $2 \times$ weight(ug) + weight(cg) > $class$ (full combinatorial collection is applicable) *noofugmoved* $\leftarrow -1$ (note that we will move all ue occurrences of ug together), $m \leftarrow ue$ (set multiplicity for ug); else *noofugmoved* $\leftarrow 1$ (we must move ug's past cg one at a time and m remains set at 1)} ; {collect $ug^m\, cg^{ce}$ by term-wise formula evaluation. During the evaluation of terms in the formula m occurrences of ug are moved past all ce occurrences of cg in one step, but other terms are left behind} *string*, $l \leftarrow [ug, cg]$; ug, $ue \leftarrow l$-th generator-exponent pair of *string* ; $m \leftarrow (m \times ce)/r$ (compute multiplicity of *string* in the evaluation of the formula); $r \leftarrow r + 1$ (compute divisor for multiplicity of the next term, if any, in the formula evaluation).

(9) (Save state of collector) $sp \leftarrow sp + 1$; *stack*$(sp) \leftarrow$ *string*, l, ug, ue, cg, ce, m, r ; $m \leftarrow m \times ue$ (compute multiplicity for ug).

(10) (Find the next letter to be moved past This step is analogous with step (3), q.v. for comments.) $ce \leftarrow ce - 1$; (10a) if $ce > 0$ go to (7); (10b) $cg \leftarrow cg + 1$; $ce \leftarrow$ *exponent*(cg) ; $r \leftarrow 1$ (we are on to a new generator-exponent pair in the trailing normal word segment, so we reset the divisor for a new formula evaluation); if $cg < ug$ go to (10a) else go to (11).

(11) If *noofugmoved* = 0 (we have not yet determined how many occurrences of ug to move, so ug has commuted with everything between its initial position and its correct position) *noofugmoved* $\leftarrow -1$, $m \leftarrow ue$ (we can move all ue occurrences of ug); (m occurrences of ug have reached their correct position, so we can insert them in the exponent vector) *exponent*$(ug) \leftarrow$ *exponent*$(ug) + m$; if *exponent*$(ug) \geq p$ and $2 \times$ weight$(ug) <$ *class* go to (17) [ug^p occurs and must be evaluated now because it could cause more collections to the right of the current position. Observe that any exponent that is allowed to build up is at worst involved in combinatorial collection, never normal collection, so the exponent build up does not slow the process.]

(12) (We have finished with m occurrences of ug so we are ready to consider the previous letter, if any, in the current string) cg, $ce \leftarrow$ *stack*(sp) (recall the position relative to the trailing normal word segment of any unprocessed part of *string*).

(13) (Find the previous letter on *string* , if any. This could be another occurrence of ug from the current generator-exponent pair.) $ue \leftarrow ue - 1$; if $ue = 0$ (there are no more occurrences of ug from this generator-exponent pair) go to (14) (proceed to previous generator-exponent pair of *string* , if any); if $sp > splim$ (we were doing combinatorial collection, but not at the changeover level, so we moved all occurrences of ug from this generator-exponent pair together) go to (14); if $sp = splim$ and *noofugmoved* = -1 (we were at the changeover level and we did move all occurrences of ug from this generator-exponent pair together) go to (14); (we have moved only one occurrence of ug and at least one remains) if $sp < splim$ (we have just finished collecting a p-th power) *noofugmoved* $\leftarrow 1$ (note that we are moving one occurrence of ug); go to (15).

(14) (Make the preceding generator-exponent pair of *string* , if any, current) $l \leftarrow l - 1$; if $l = 0$ (we have reached the beginning of *string*) go to (16) (return to the previous collection); ug, $ue \leftarrow l$-th generator-exponent pair of *string* ; *stack*$(sp) \leftarrow l$, ug (update the state of the collector, all other stack entries, except ue , are correct, and ue is updated in step (15); if $sp \leq splim$ (we are in ordinary collection) *noofugmoved* $\leftarrow 0$ (note that we have not determined how many occurrences of ug we can move together).

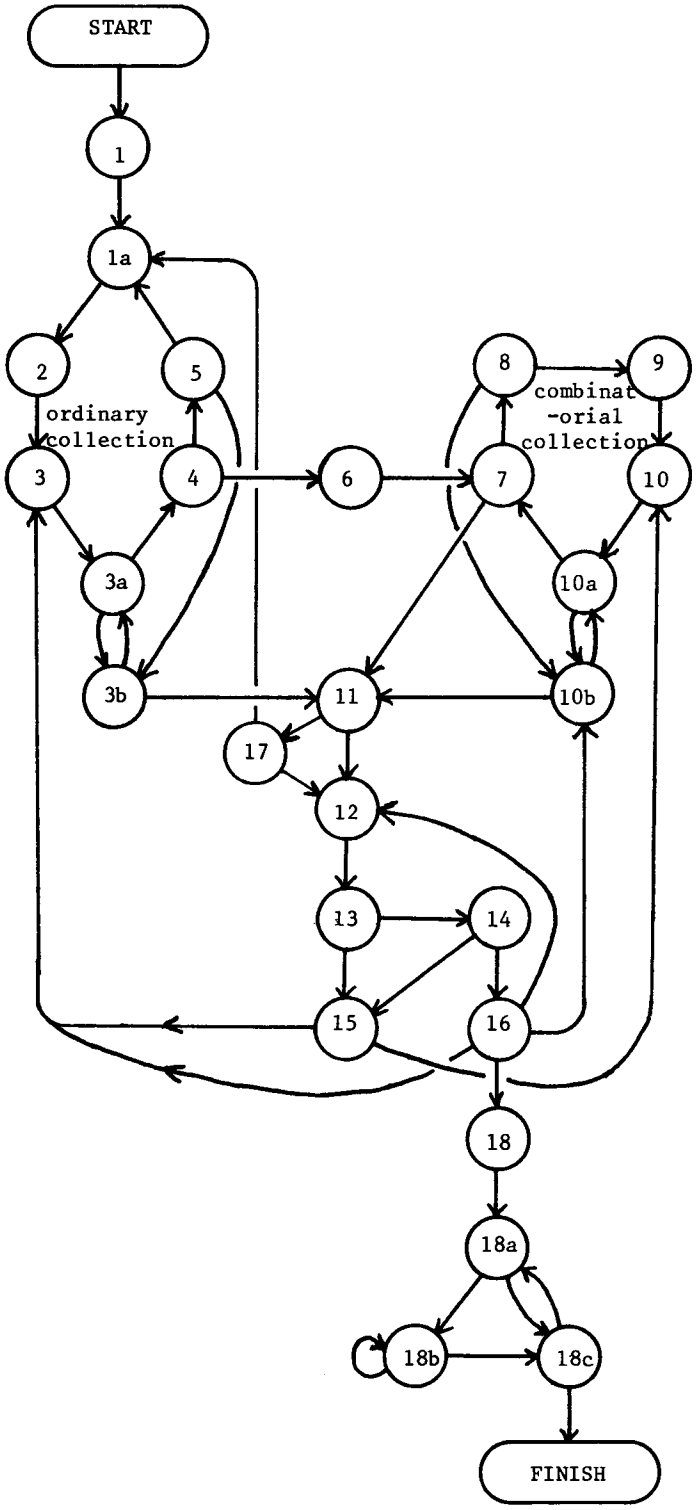
(15) (Process the next occurrence of ug from the current generator-exponent pair) $stack(sp) \leftarrow ue$ (update the state of the collector, all other stack entries are correct); $m, r \leftarrow stack(sp)$ (recover the multiplicity of the current string); if $sp \leq splim$ (we are using ordinary collection or at the changeover level) go to (3) (return to ordinary collection); else (we are using combinatorial collection) $m \leftarrow m \times ue$ (calculate the full multiplicity of ug); go to (10) (return to combinatorial collection).

(16) (We have finished processing the current string remembered in $stack(sp)$, so go back to the previous string, if any) $sp \leftarrow sp - 1$; if $sp = 0$ (we have virtually finished the entire collection, it remains to tidy up p-th powers in the exponent vector) go to (18); $ug, ue, string, l, m, r \leftarrow stack(sp)$ (restore previous collection in progress); if $ce = 0$ (last string was generated by a p-th power) go to (12) (return to where the p-th power came from); if $sp < splim$ (we are back to ordinary collection moving a letter that has already been moved, causing the introduction of an additional commutator) $noofugmoved \leftarrow 1$ (note that we have moved one occurrence of ug), go to (3) (return to normal collection); (we are back to combinatorial collection) if $sp > splim$ or $noofugmoved = -1$ (we are back to full combinatorial collection) $m \leftarrow m \times ue$ (calculate total multiplicity of ug); go to (10b) (return to combinatorial collection).

(17) [Evaluate ug^p] $exponent(ug) \leftarrow exponent(ug) - p$ (remove the p occurrences of ug that we are about to collect from the exponent vector); if ug^p is trivial go to (12) (return to where the p-th power came from). If $noofugmoved = -1$ (we are moving all occurrences of ug from this generator-exponent pair) $stack(sp) \leftarrow ue \leftarrow 1$ (alter the stored ue so that when we return we know that all occurrences of ug have been moved); $m \leftarrow 1$; $r \leftarrow 1$ [reset parameters for collection of ug^p]; $cg \leftarrow ug$; $ce \leftarrow 0$ [note position of ug^p relative to the exponents, $ce = 0$ indicates that this is a p-th power collection, and is used in step (16)]; $string, l \leftarrow ug^p$; go to (1a) (return to collection).

(18) (Tidy up the exponent vector by removing any exponents greater than p) $cg \leftarrow$ first generator with $2 \times$ weight $\geq class$ (for each generator with $2 \times$ weight $\geq class$ we allow the exponents to exceed p because the p-th power commutes with all subsequent generators so can be directly incorporated into the exponent array); (18a) $ce \leftarrow exponent(cg)$; if $ce < p$ go to (18c); $exponent(cg) \leftarrow exponent(cg) \bmod p$; if cg^p is trivial go to (18c); $m \leftarrow$ integer part of (ce/p) (calculate number of p-th powers to insert into the exponent array); $string, l \leftarrow cg^p$; (18b) $ug, ue \leftarrow l$-th generator-exponent pair of $string$; $exponent(ug) \leftarrow exponent(ug) + ue \times m$ (incorporate appropriate number of occurrences of ug into exponent vector); $l \leftarrow l - 1$; if $l > 0$ go to (18b) (process previous generator-exponent pair of $string$); (18c) $cg \leftarrow cg + 1$; if $cg \leq n$ go to (18a) (process next generator-exponent pair of the exponent vector); (we now have a vector of exponents corresponding to the collected result).

5.5. Flow diagram.



REFERENCES

[1] William A. Alford, George Havas and M.F. Newman, "Groups of exponent four", *Notices Amer. Math. Soc.* 22 (1975), A-301.

[2] J.M. Campbell and W.J. Lamberth, "Symbolic and numeric computation in group theory", *Proc. Third Austral. Computer Conf.* (Canberra, 1966), 293-296 (Australian Trade Publications, Chippendale, Australia, 1967). CR8#12,454.

[3] E. Czyżo, "On the determination of Hall polynomials", *Bull. Acad. Polon. Sci. Sêr. Sci. Math. Astronom. Phys.* 23 (1975), 739-745.

[4] Emanuel Czyżo, "An attempt of mechanization of Hall collecting process", *Algorytmy* 9 (1972), 5-17. MR46#5419.

[5] E. Czyżo, "An automatization of the commutator calculus", *Algorytmy* 10 (1973), 23-34. MR48#1509.

[6] K. Dałek, "Computation of exponents in Hall formula", *Bull. Acad. Polon. Sci. Sêr. Sci. Math. Astronom. Phys.* 19 (1971), 711-718. MR48#11327.

[7] K. Dałek, "On certain general forms of the exponents of simple commutators in Hall formula", *Bull. Acad. Polon. Sci. Sêr. Sci. Math. Astronom. Phys.* 20 (1972), 973-980. MR48#11338.

[8] K. Dałek, "On the Hall formula in nilpotent and solvable groups", *Bull. Acad. Polon. Sci. Sêr. Sci. Math. Astronom. Phys.* 23 (1975), 829-837.

[9] H. Felsch, "Die Behandlung zweier gruppentheoretischer Verfahren auf elektronischen Rechenmaschinen", Diplomarbeit, Kiel, 1960.

[10] Volkmar Felsch, "A machine independent implementation of a collection algorithm for the multiplication of group elements", *These Proceedings*.

[11] Waltraud Felsch, "Ein commutator collecting Algorithmus zur Bestimmung einer Zentralreihe einer endlichen p-Gruppe", Staatsexamensarbeit, Aachen, 1974.

[12] Marshall Hall, Jr., *The Theory of Groups* (The Macmillan Co., New York, 1959). MR21#1996.

[13] P. Hall, "A contribution to the theory of groups of prime-power order", *Proc. London Math. Soc.* (2) 36 (1934), 29-95.

[14] George Havas, G.E. Wall, and J.W. Wamsley, "The two generator restricted Burnside group of exponent 5 ", *Bull. Austral. Math. Soc.* 10 (1974), 459-470.

[15] H. Jürgensen, "Calculation with the elements of a finite group given by generators and defining relations", *Computational Problems in Abstract Algebra* (Proc. Conf. Oxford, 1967), 47-57 (Pergamon, Oxford, 1970). MR41#3600.

[16] Wolfgang Lindenberg, "Über eine Darstellung von Gruppenelementen in digitalen Rechenautomaten", *Numer. Math.* 4 (1962), 151-153. MR26#7174, CR4#4007.

[17] W. Lindenberg, "Die Struktur eines Übersetzungsprogramms zur Multiplikation von Gruppenelementen in digitalen Rechenautomaten", *Mitt. Rh.-W. Inst. Instr. Math. Bonn* 2 (1963), 1-38.

[18] I.D. Macdonald, "A computer application to finite p-groups", *J. Austral. Math. Soc.* 17 (1974), 102-112.

[19] J. Neubüser, "Bestimmung der Untergruppenverbände endlicher p-Gruppen auf einer programmgesteuerten elektronischen Dualmaschine", *Numer. Math.* 3 (1961), 271-278. MR24#B489.

[20] M.F. Newman, "Calculating presentations for certain kinds of quotient groups", *These Proceedings*.

[21] T.W. Sag and J.W. Wamsley, "On computing the minimal number of defining relations for finite groups", *Math. Comp.* 27 (1973), 361-368.

[22] J.W. Wamsley, "Computation in nilpotent groups (theory)", *Proc. Second.Internat. Conf. Theory of Groups* (Canberra, 1973), 691-700 (Lecture Notes in Mathematics, 372. Springer-Verlag, Berlin, Heidelberg, New York, 1974). MR50#7350.