Centre for Discrete Mathematics and Computing

School of Computer Science & Electrical Engineering
and Department of Mathematics,
The University of Queensland, Qld 4072

Technical Report #25

Title:	ACME 1.000: an Andrews-Curtis move enumerator
Author:	Colin Ramsay
Date:	June 27, 2001
Version:	the rough guide

1 Introduction / Background

TBA ...

2 General commands

```
  bye / exit / q[uit] ;
```

Exit ACME.

```
  h[elp] ;
```

Print summary of all commands.

```
  opt[ions] ;
```

Dump date/time of compilation, and machine name.

```
  sys[tem] : <string> ;
```

Pass the string along to a shell.

```
  text : <string> ;
```

Dump the string – use for pretty-printing the output.

```
  # ... <newline> - a comment (ignored)
```

As it says.

3 Parameter settings

Apart from the `rel` command, if a command which normally takes an argument is entered without any argument, then its current value is printed.

```
  as[is] : [0/1] ;
```

Before any of the commands is run, the presentation is passed through a massaging routine. If `asis` is false (ie, `0`), then this will freely and cyclically reduce the presentation, and sort the relators into length plus lexicographic order. The default is for `asis` to be true; ie, use the presentation as given.

Note that the routines which implement AC-moves will usually (but not in all cases) perform free reductions as the relators are built up; so relators are usually freely reduced. Potential cyclic reductions, introduced by conjugation, are not normally done. However, cyclic conjugates of relators which have potential cyclic reductions have potential free reductions, which will usually be performed. This can sometimes cause confusion, so take care. (For example, if running `plan9` with `equiv:1`, a proof chain may appear to start with a shorter presentation than the one indicated at the start of the current iteration.)

```
cull : [0 / 1,int / 2,int / 3,int] ;
```

Culling controls whether or not newly generated presentations are actually looked up in the tree (or whatever) and then added if they're really new; culled presentations are rejected prior to the lookup stage. The first argument sets the `cull` parameter and the second (if present) sets `clen`. If `cull` is `0`, there is no culling. If `cull` is `1`, then presentations are culled if any of the relators have length more than `clen`, with `clen` ≥ 0. If `cull` is `2`, then presentations are culled if any of the relators are more than `clen` longer than their initial lengths, with `clen` ≥ 0. If `cull` is `3`, then presentations are culled if the total presentation length is more than clen, with `clen` ≥ 0.

Note that the default is `cull:2` & `clen:0`, and that not all culling modes are implemented in all of the commands. Selecting an unimplemented mode selects `cull:0`; ie no culling.

```
def[aults] ;
```

Restores all the parameters to their defaults; this is, everything in this section apart from `gr` & `rel` (which are not altered).

```
dump : [-1 / 0 / 1 / 2,int] ;
```

Dumping controls which presentations, with their proof chains, are printed; every time a new presentation is seen and added to the tree (or whatever), it may be dumped out. The first argument sets the `dump` parameter and the second (if present) sets `dlen`. If `dump` is `-1`, there is no dumping. If `dump` is `0`, then every new presentation is dumped. If `dump` is `1`, then solutions (ie, presentations of total length n) are dumped. If `dump` is `2`, then solutions which reduce the presentation length by at least `dlen` are dumped, with `dlen` ≥ 0.

Note that the default is `dump:1`, and that not all dumping modes are implemented in all of the commands. Selecting an unimplemented mode selects `dump:-1`; ie, no dumping.

The dump is done by tracing backwards from the newly generated solution, so the root node will appear at the bottom. Each presentation is prefaced by its total length, and lengths shorter than the starting length are flagged by a `*`. Although not visible, the printout of the relators includes a trailing space. Thus, each relator has both a leading & a trailing space, so the output can be conveniently searched by `grep` or somesuch utility without ambiguity.

```
equiv : [0/1] ;
```

Given a presentation with n generators & relators, and with the relators having lengths l_i, then the presentation is one member of a class of equivalent presentations of size $n!2^n \prod l_i$, all of which are AC-equivalent. It is often helpful, when extracting shortest proofs, to ignore initial or terminal AC-moves which simply move between such equivalent presentations; we are interested in the essential length of a proof.

If `equiv` is `1`, then all $2^n \prod l_i$ equivalent presentations due to relator cyclings & inversions are put onto the tree as root nodes before any AC-moves are made. (Permuting the order of the relators cannot effect the length of a shortest proof, so we don't bother.) So any chains of AC-moves printed have had any inessential initial moves stripped off. The default for `equiv` is false (ie, `0`), where only the presentation as given is a root node.

```
  gr[oup generators] : [<letter list>] ;
```

A list of lower-case letters, with no repeats, perhaps separated by commas. These are the group's generators; numeric generators are not allowed.

```
  len[lim] : [0 / +int] ;
```

In some commands, a length limit parameter might be needed. If so, this is it. The default of `0` means that it's inactive, and a positive value is active.

```
  lev[lim] : [0 / +int] ;
```

In some commands, a level limit parameter might be needed for the tree. If so, this is it. The default of `0` means that it's inactive, and a positive value is active.

```
  mess[ages] / mon[itor] : [int] ;
```

Sets the interval between progress messages. These messages are in terms of the number of AC-moves made; ie, the number of (potentially) new presentations generated. A value of `0` (the default) turns this feature off. Any positive value turns it on. The messages are flagged with `AP` and give counts of `cap` (calls to the "add presentation" routine, which counts the number of newly generated presentations), `nap` (the number of new presentations actually added) & `nar` (the number of new relators added).

Note that the CPU time is accumulated during a run, and that the counter for this often does strange things after 2^{31} or 2^{32} ticks (typically 35 min or 71 min). To get accurate timings for long runs, make sure messaging is on, since the CPU time increment is calculated at every progress printout.

```
  nap[lim] : [0 / +int] ;
```

In some commands, a "number of added presentations" limit parameter might be needed for the tree (or whatever). If so, this is it. The default of `0` means that it's inactive, and a positive value is active.

```
  param[eters] / sr ;
```

Dumps the value of all parameters, including `gr` & `rel`.

```
  rel[ators] : <relation list> ;
```

The group's relators. A variety of formats are accepted, as in ACE. Note that the various commands will refuse to run if the presentation is not balanced.

```
  stat[s] : [0/1] ;
```

If active (an argument of `1`), this includes in the printout details of the levels of the tree as they're processed. After all the presentations at one level have been processed, details of the sizes of the parent and child levels are dumped; the figures are the number of new presentations and new relators at that level. The root is level zero, so the number of moves in a chain of printed moves is the last level plus one. The feature is on by default, and can be turned off by a `0` argument.

```
  term : [0 / 1 / 2,int] ;
```

Termination controls under what circumstances the search will be halted, over and above any termination due to the fact that the search tree has been exhausted. The first argument sets the `term` parameter and the second (if present) sets `tlen`. If `term` is `0`, then there is no termination condition other than the exhaustion of the BF-tree (if this occurs). If `term` is `1`, then stop as soon as a (new) presentation of length n (ie, a solution) is added to the tree. If `term` is `2`, then stop if a presentation length reduction of at least `tlen` is seen, with `tlen` ≥ 0.

Note that the default is `term:1`, and that not all termination modes are implemented in all of the commands. Selecting an unimplemented mode selects `term:0`; ie, terminate on exhaustion.

4 `prog8` – breadth-first search (tree)

The `prog8` command does a breadth-first search through the tree of AC-moves. The root is the presentation as given (by default), or the $2^n \prod l_i$ members of its equivalence class (if `equiv:1`). The unique relators and presentations seen are stored in trees, and the presentation tree is threaded with linked lists for the BF-search and for tracing a node's ancestors. These trees are still in existence when a command returns, and are only destroyed at entry to the next command (there may be a delay while the memory used is freed).

The printout starts with a dump of the active parameters and the presentation, and then contains any `stat`, `mess` & `dump` stuff as requested. Any dumped AC-move chains are preceded by a count, so you can work out whether or not you've seen all members of some class of presentations. The printout ends with a few statistics regarding the total number of moves made (calls to "addpres"), the number surviving culling (calls to "fndrel"/"fndpres"), and the number of new presentations and relators seen. Note that these counts do not include the initial presentation(s). Finally, the accumulated CPU time for the run is dumped.

CULLING: `cull` modes `0`, `1`, `2` & `3` are implemented.

DUMPING: `dump` modes `-1`, `0`, `1` & `2` are implemented.

TERMINATING: `term` modes `0`, `1` & `2` are implemented. `levlim` & `naplim` are implemented. For `levlim` to work, `stat` must be active. There may be a bit of slop in `naplim` as it is only tested each time all $3n^2$ moves from the current parent have been made.

5 `plan9` – random(ish) walk (tree)

The `plan9` command implements a (random) walk in the search space, attempting to find an AC-move chain which reduces the presentation length as requested. It's only actually random in the sense that varying the `naplim` parameter varies which presentation is used to start the next iteration; there is not (as yet) any indeterminism. None of the `dump` or `term` modes should be used; instead, set `lenlim` to the total presentation length at which (or below) you wish to stop. Although the normal cull modes are present, they are w.r.t. the presentation at the start of the current iteration; so you should use the absolute length limit given by `cull:3` (or, perhaps, `cull:1`).

The printouts for iterations are separated by `#--` lines, and each iteration starts with the AC-move chain from the previous starting presentation to the current starting one. A new iteration is started when `naplim` new presentations have been generated; the next starting presentation is the last new presentation generated. The CPU time for each iteration is the total CPU time so far. Note that, when the command finishes, either because the requested length reduction has been found or because the BF-tree is exhausted, then the current presentation is the presentation at the start of the last iteration.

CULLING: `cull` modes `0`, `1`, `2` & `3` are implemented.

DUMPING: Only `dump:-1` is implemented; use `lenlim`.

TERMINATING: Only `term:0` is implemented; use `lenlim`. `levlim` is not implemented. `naplim` controls iteration size, not termination.

OTHER: The counting of dumped AC-move chains (ie, the number of iterations) is not implemented. If `equiv:1`, it is active at the start of every iteration.

6 `rev10` – breadth-first search (reverse)

The `rev10` command is intended to run a BF-search in reverse, from the standard presentation (although you can start from whatever you want). Any AC-move chains should be read backwards (ie, from top to bottom), since the starting presentation is now the end of the chain, not the beginning. Conjugation & inversion are undone by conjugation & inversion, so these moves are unchanged. Appending one relator to another is undone by an inversion, an append, and an inversion; so the append move has been replaced by an "append the inverse" move.

CULLING: `cull` modes `0`, `1`, `2` & `3` are implemented.

DUMPING: `dump` modes `-1` & `0` are implemented.

TERMINATING: `term` mode `0` is implemented. `levlim` & `naplim` are implemented (see `prog8` notes).

OTHER: The counting of dumped AC-move chains is not implemented.

7 `temp11` – template (proof) search

The `temp11` command is intended to investigate the possible proof word templates which can be generated by AC-moves. The template of a proof word is simply the proof word stripped of all its conjugation, and with each (initial) relator represented by a letter. We are interested is using a proof word generated by some means to generate a proof of AC-equivalence; a necessary condition for this is that we can generate the template by AC-moves. The possible templates are easily generated by modifying `prog8` to disallow the conjugation move, to disallow free cancellations, and by starting with the standard presentation. This is what `temp11` does, although you are free to start with whatever presentation you want.

CULLING: `cull` modes `0`, `1`, `2` & `3` are implemented.

DUMPING: `dump` modes `-1` & `0` are implemented.

TERMINATING: `term` mode `0` is implemented. `levlim` & `naplim` are implemented (see `prog8` notes).

OTHER: The `equiv` option is not implemented. The counting of dumped AC-move chains is not implemented.

8 `duodec` – bidirectional search

If you want to prove that a proof chain is shortest, and are unable to do so using `prog8` due to memory blow-out, then try `duodec`. This does a `rev10` type search for `levlim` levels, keeping the presentations produced, and then does a `prog8` type search while checking each new presentation against the reverse search. So you can do exhaustive searches to a greater depth. It can also, with suitable parameters, be much faster than a unidirectional search for general (constrained) searches.

The reverse search starts by priming the root node with all eight presentations equivalent to (a, b), and then building a tree of presentations out to the `levlim` level. This tree is preserved, and then the forward search starts building its tree; both presentation trees use the same relator tree. The forward search primes the root node with the $4l_1l_2$ equivalent presentations (ie, `equiv:1` is assumed). If a newly added presentation is in the reverse tree, then we've found a (shortest) proof, which we dump and then exit.

All other things being equal, you should run this command with equal size forward & reverse searches, to minimise time & space usage. The reverse search is faster than the forward one (since we don't need to keep checking for a match), and its space usage is known in advance (since it always starts with the same presentation). (An unconstrained reverse tree to eight levels needs circa 2Gb, on a 32-bit machine.) So, erring on the side of a larger (but not by too much) reverse search is usually good.

Note that the reverse tree is checked to see if it contains the initial presentation (or an equivalent thereof) before the forward search starts. If so, we print a proof and stop; this proof may not be a shortest (essential) proof, since it is simply the first match found between the two trees. To support a claim of shortest, both searches must go at least one level.

CULLING: `cull` modes `0`, `1`, `2` & `3` are implemented. However, the relative mode doesn't make too much sense, given that there are two different starting presentations. You should use `cull:0` for an exhaustive search, or `cull:1` or `cull:3` for constrained searches.

TERMINATING: The only termination for the reverse search is `levlim` or BF-tree exhaustion. Tree exhaustion terminates the command, while `levlim` automatically goes on to the forward search. The only termination for the forward search is success, `naplim` or BF-tree exhaustion.

DUMPING: There is no dumping for the reverse search. Successful termination will dump out the proof chains for the reverse and forward parts of the search.

OTHER: This command is only available for the $n = 2$ case.

9 Examples / Future Work / Conclusions

TBA ...