by Jeffreys Copeland and Haemer

PAUL STODDARD

Novus ordo seclorum.
– The Great Seal of the United States of America

Puzzle Posters, Part 2

This column is a continuation of last month's. Let's briefly review how we got here. First, perusing `comp.lang.perl.misc`, we found an interesting puzzle posted by Tim Bunce (shown below).

One response to Tim's puzzle came from Nat Torkington: "You *install* a *full set of tools*, like the *Lord God Almighty* intended. *Repent*, ye *prisoner* of *Bill*! The *Day of Judgement* is *at Perl*! Your *messengers* are obviously just *poor* substitutes for *reliable pipe communication*, which you'd have if you had a *real operating system* and not a *scurrilous piece of tool-challenged coprophilia*!"

Not only did we think the response was funny (it parodies postings by Nat's friend and coauthor Tom Christiansen), we also realized Nat was correct: preprocess the data, then pipe it to that venerable UNIX utility, `tsort`.

We sent our solution to Tim and he wrote back saying that Nat had been right for another reason: his operating system (from some company in Redmond, WA) doesn't come with a `tsort`. We volunteered to write Tim one in Perl.

Lazily, we simply looked up a `tsort` implementation from Jon Bentley's book, *More Programming Pearls*, published by Addison-Wesley Publishing Co., 1988, ISBN 0-201-1189-0. Bentley, in turn, reused the algorithm from Don Knuth's *The Art of Computer Programming*. (We might have done the same thing, except someone has walked off with our copy of Knuth.)

Finally, we wrote a column explaining the overall solution and promised we'd

```
A list of names in a specific order is given to a set of messengers in a remote land.
The messengers travel independently to a destination where they give the names
to you. The problem is that the messengers quite often, say 70%, miss out one or
more names and occasionally, say 10%, get the order wrong. Names are never added,
repeated or changed, only missed or reordered. The messengers always think
they've got it right. For example,

        Original list:      foo bar baz boo

        Messenger A says:   foo bar boo
        Messenger B says:   bar boo baz
        Messenger C says:   foo bar baz boo
        Messenger D says:   boo foo bar baz
        Messenger E says:   foo bar baz
        Messenger F says:   foo baz boo

The problem is to find the full list of names and the original order.
Tim
```

Listing 1. Our Perl Code

```
#!/usr/local/bin/perl -w
# $Id: tcsort,v 1.5 1998/08/04 20:29:07 jsh Exp jsh $

use strict;

use vars qw($opt_b $opt_d);
use Getopt::Std;
my $usage = "usage: $0 [-b|-d] [filename]\n";
getopts("bd") or die $usage;
die $usage if ($opt_b && $opt_d);

my %pairs;# all pairs ($l, $r)
my %npred;# number of predecessors
my %succ;# list of successors

while (<>) {
  my ($l, $r) = my @l = split;
  next unless @l == 2;
  next if defined $pairs{$l}{$r};
  $pairs{$l}{$r}++;
  $npred {$l} += 0;
  ++$npred{$r};
  push @{$succ{$l}}, $r;
}

# create a list of nodes without predecessors
my @list = grep {!$npred{$_}} keys %npred;

while (@list) {
  $_ = pop @list;
  print "$_\n";
  foreach my $child (@{$succ{$_}}) {
   if ($opt_b) {# breadth-first
    unshift @list, $child unless --$npred{$child};
   } else {# depth-first (default)
    push @list, $child unless --$npred{$child};
   }

  }
}
warn "cycle detected\n" if grep {$npred{$_}} keys %npred;

=head1 NAME

tcsort - topological sort

=head1 SYNOPSIS

  tcsort [filename]

=head1 DESCRIPTION

=over 2

Does a topological sort of input pairs.

For a more complete description, see the tsort(1) man page,
For a fine explanation of the algorithm, see the October 1998
Work column in SunExpert, or the references given below.

=back

=head1 OPTIONS AND ARGUMENTS

=over 8
```

write a second column about `tsort` implementation. This is it.

So what's a `tsort` and why does UNIX have one, anyway?

Suppose you have a list of ordered pairs. Turning them into a single, ordered list, in which the second element of each pair is always after the first, is called *topological sorting*. The `tsort` utility performs topological sorts.

An example will help. If you know that A is before B, A is before C and B is before C, then the correct list order is A B C.

Things are not always this simple, however. Suppose, for example, we add more information: A is before Z and Z is before C. Now, the order could be A B Z C or A Z B C. Either of these two orders is a correct topological sort of the input data.

"But Z and B could be tied," you helpfully point out. Sure. This is even a problem with regular sorts. If we sort the following numbers, 2 1 3 2 2, the number twos can come in any order. `tsort` and `sort` try only to produce a single, ordered list consistent with the data. `tsort`'s only restriction is that there must be no cycles in the input data. If A is before B, B is before C and C is before A, `tsort` is stumped.

(The three number twos in our example remind us that there are sentences that you can speak but cannot write. For example, "There are three 'tuuz' in the English language: 't-o,' 't-o-o' and 't-w-o.'" Out loud, this sentence is factual and not at all artificial. Putting it on paper requires either rewording the sentence or using an artificial spelling, such as "tuuz," that you won't find in any dictionary. This lovely example is courtesy of the late Col. Alan G. Haemer, U.S.A.F.)

But why does UNIX come with a special utility to do topological sorting? Who uses it?

One way to find out is to look through all the executables in your path for any use of `tsort`:

```
for i in $(echo $PATH | sed 's/:/ /g')
do
 grep -l tsort $i/*
done
```

The only utility we could find when we did this was `x11perfcomp`. Note that if you `grep` for `sort`, instead of `tsort`, you'll see a big difference.

"But surely," you say, "this can't be all it's used for." Correct. And herein follows a history lesson.

Back in the old days, computers were much slower; you could often go out for coffee while your programs compiled and linked. Programmers did all sorts of things to minimize compilations that we no longer have to do, such as actually thinking about code before compiling it. One thing that helped

```
48 =item B<[-b|-d]>

49 breadth-first or depth-first (default) traversal

50 =item B<filename>

51 Optional input file.
52 Input format is pairs of white space-separated fields,
53 one pair per line.
54 Each field is the name of a node.

55 Output is the topologically sorted list of nodes.

56 Ignores lines without at least two fields.
57 Ignores all fields on the line except the first two.

58 =back

59 =head1 AUTHOR

60    Jeffrey S. Haemer, <jsh@boulder.qms.com>

61 =head1 SEE ALSO

62 tsort(1), tcsh(1), tchrist(1)

63 Algorithm stolen from Jon Bentley (I<More Programming Pearls>,
64 pp. 20-23), who, in turn, stole it from Don Knuth
65 (I<Art of Computer Programming,
66 Volume 1: Fundamental Algorithms>, Section 2.2.3)

67 =cut
```

was being able to link precompiled versions of utility routines into your executable. This removed the requirement to recompile, say, `printf()`, each time you compiled `hello, world`. An advance built on top of this was the ability to collect related, compiled object files into libraries, such as

```
/usr/lib/libc.a.
```

Linkers (`ld` on UNIX systems) allowed you to search through one or more libraries for routines that your program called but didn't define. Unfortunately, even searching libraries could be time-consuming. For example, if your code called `strdup()`, the linker would search `libc.a` to find `strdup.o`. But `strdup.o`, in turn, calls `malloc()`, so a second search of `libc.a` was needed to find and extract `malloc.o`. And because `malloc.o` calls `fprintf()`…and so it goes.

This could all be done with a single pass through the library by arranging the object files in an order that put each object file in the library before the external functions it called.

Starting to sound familiar?

To accomplish this, the `lorder` utility was written to find and list all the pairwise dependencies among object files and a second utility, `tsort`, took these dependencies and put them in the right order. This list, in turn, was given to `ar`, which created libraries in the order it was told.

On modern systems, a much-enhanced `ar` does all the work for you (and in a different way). Nevertheless, UNIX systems still come with `tsort`. After all, it works. And you can still occasionally use it to solve problems like the one posed by Tim.

If, just for fun, you want to see the pair at work, try this:

```
$ mkdir /tmp/tsort_demo
$ cd /tmp/tsort_demo
$ ar x /usr/lib/libc.a   # extract copies
                         # of all .o files
$ lorder *.o | tsort
```

The tree-linearization news article code we wrote in our July and August columns ("Cathedrals, Bazaars, and News Readers," Page 57 and "Virtual Threaded News Reader," Page 54, respectively) was related to this problem: The news problem can be partially solved with `.CR tsort`.

Question for our readers: Does anyone know why UNIX always comes with Bessel functions of the second kind, `y0`, `y1` and `yn`?

An Implementation

Enough already. Listing 1 contains our code. Herein follows a dramatic reading.

Lines 1 through 3 are our usual, cowardly boilerplate. We want Perl to tell us about our stupid mistakes, and we keep the code under Revision Control System (RCS) so we can retrieve older versions with fewer stupid mistakes. Lines 4 through 8 do argument parsing and handle the usage message. Lines 9 through 11 declare some hashes, and it's worth pausing here for a minute to talk about the data structures.

We're going to keep all the elements to be sorted as arbitrary strings. Each input line (for example, "age beauty," to mean "age comes before beauty") has two elements. Unless this pair has been seen before, which we'll track by defining a hash element, `$pair{"age"}{"beauty"}`, an input line will have at least two effects:

1. The hash entry `$npred{"beauty"}`, which counts the number of predecessors of `beauty`, will be incremented.

2. `beauty` will be added to the list of successors of `age`, `$succ{"age"}`. Note that `$succ{"age"}` will be a reference to an array containing all the successors of `age`.

Lines 12 through 20 populate these structures. The remaining lines traverse these structures, printing them out as the sorted list.

So how do they work? The margins of this column aren't large enough to illustrate that lines 21 through 31 provide a topological sort, so we'll leave this as an exercise for our readers. We will, however, give you more of a hint than you'll find in either Knuth or Bentley, and show you how to get two different `tsort`s from one piece of code.

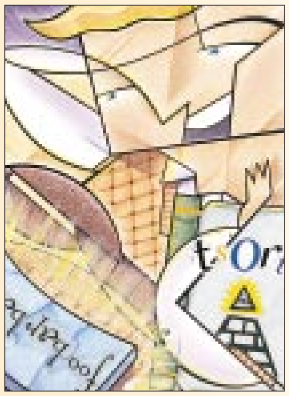
Traversing Trees

Let's start by talking about tree traversal. The most familiar way to traverse a tree is with a depth-first search. Here's our favorite algorithm for a depth-first search:

- Start with an empty stack.
- Push the root of the tree onto the stack to initialize.
- Pop the stack, print what you find and push its children back on the stack in its place.
- Continue until the stack is empty.

Consider a trimmed-down UNIX directory tree as an example: Push the root onto the stack (`/`); pop off the root and print it, then push its children on (`/etc /lib /usr`); next, pop off the first child, `/etc`, print it and push on its children (`/etc/rc /etc/passwd /lib /usr`); pop again, print `/etc/rc`, push on any children of `/etc/rc`; and continue like this until the stack is empty.

In a tree, a parent can only be linked to its immediate children. In an arbitrary, cycleless graph, parent nodes can also have links to offspring nodes farther down the tree.



Not only is this easy, but replacing the stack by a queue gives a breadth-first traversal instead.

This is essentially what lines 21 through 31 are doing, with a twist. In a tree, a parent can only be linked to its immediate children. In an arbitrary, cycleless graph, parent nodes can also have links to offspring nodes farther down the tree. To handle this complication, we keep track of how many untraversed predecessors each node has and only push it on the queue when none remain.

To keep track of this, we only consider a node of the graph a "child" eligible to be pushed on to `@list`, when we've just printed its immediate parent. How do we know when we're at the immediate parent? We use the hash `%npred` to keep track of how many predecessors are left. When the node is out of predecessors, it's really a child. In essence, we're turning a graph into a tree as we traverse it.

We'd like to thank Tim Bunce again for his entertaining and educational puzzle and Nat Torkington (and, indirectly, Tom Christiansen) for just the right clue.

Until next time, happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats and being a thorn in the side of his local school board.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work.html`.