by Jeffreys Copeland and Haemer

ERIC MUELLER

"It's not the voting that's democracy; it's the counting."
– Tom Stoppard

"He sighed wearily, 'I just accidentally wrote an SF novel, okay? I didn't mean to apply for citizenship in the Twilight Zone.'
"'I don't think you can apply, Jay. I think fandom takes hostages.'"
– Zombies of the Gene Pool,
by Sharyn McCrumb

Gernsback Meets Aristotle

For a long time, one of the ideals of western civilization has been the democratic election. And the ideal of the democratic election has been the majority winner. But what happens when you have more than two candidates? How can you ensure a majority winner? And how can you ensure that the winner really reflects the wishes of the voters?

We've talked before in these pages about different types of elections. Our final article for *RS/Magazine* and our first for this publication were a pair of columns about using a CGI script to tally U.S. Electoral College votes (see "Counting on the Net," *RS/Magazine*, February 1997, Page 29, and "Drawing on the Net," *SunExpert*, April 1997, Page 87, `http://sw.expert.com/C9/RS.C5.APR.97.pdf`.).

As you may recall from your high school civics class, each state in the United States has a number of electoral votes equal to the number of its federal legislators. Today, in most states, all those votes are cast for whoever wins the plurality of votes in the state. The new president is the candidate who wins a majority of the 538 electoral votes. Unfortunately, this means that to become president of the United States one merely has to win a plurality of the votes in the 11 most populous states, and not one additional vote.

Is there a better solution? The problem is old, well-studied and answered. That answer is "No."

In 1951, Nobel prizewinning economist Kenneth Arrow proved, in his Ph.D. dissertation, that no "best" voting scheme exists as soon as there is more than one voter. (You can always satisfy a dictator perfectly.) Arrow's Impossibility Theorem had roughly the same impact on political science that Gödel's Incompleteness Theorem had on mathematics.

Basically, the thing to do is pick a scheme that's mostly fair and get on with politics as usual.

That fact doesn't stop people from coming up with interesting and useful alternatives. For illustration, let's consider two examples from studies by Jean Borda and the Marquis de Condorcet for the French Academy in the 18th century.

Borda's is probably the more obvious method. The first-ranked candidate on an *n*-person ballot is given *n* points, the second-ranked candidate *n*-1 points and so on, with the last-ranked candidate being given one point. The points for each candidate are totaled and the winner is the one with a plurality. (This may be familiar to the sports-literate among our readers: Borda's is the method used to determine the annual ranking of college football teams by poll of sports writers and coaches. Why this method is used instead of simply seeing who wins the most games is a mystery to Copeland, who is a football anti-fan.)

Condorcet came up with a different technique. He suggested using the rankings on the ballot to figure out the results of elections between each pair of candidates. The winner of the election is the "most preferred" candidate in these pairwise tallies.

Some election activists suggest that

changing voting schemes to this sort of preferential balloting would break up our two-party system and decrease the polarization of our political process. Our counter-examples would be Israel and Italy. Both are parliamentary democracies with multiparty elections. However, in the former, the political process is polarized to the point of preventing a peace settlement; in the latter, the political process is so unstable that the average lifetime of a ruling coalition since the end of World War II has been less than 18 months.

Enough about geopolitics, let's change gears for a moment.

A Practical Application

At the 11th World Science Fiction Convention in 1953, the membership first voted on what has become a staple of the science fiction world: the Hugo Awards. The awards are named after the Luxembourg-born writer and editor Hugo Gernsback, who founded *Amazing Stories*, the first of the pulp SF magazines. That first year, the winners included Alfred Bester for his novel *The Demolished Man* and Phillip José Farmer as best new SF author. Nearly half-a-century later there are 13 categories of Hugos, and the "best new author" award has been institutionalized as The John W. Campbell Award, named for the editor of *Astounding*, and the man who shaped short SF for the middle part of the century.

We remembered the Hugos a couple of weeks ago when the nominees for the 1999 awards were announced and we found our friend Guy Lillian on the ballot for his fanzine, *Challenger*. This reminded us that a column on tallying the Hugo ballots has been rattling around in our topic drawer for a while. (In yet another sign of how far in advance these columns are written, the nominees were announced in mid-April, but the final voting deadline is July 31. This leaves a month before the World Science Fiction Convention–in Chicago this year, see `http://www.chicon.org` for details–for Mike Nelson, the Hugo Awards administrator, to tally the ballots.)

In any case, this brings us to the Hare method, which dates to the middle of the 19th century. Because Hare's method is used in Australia, it's sometimes referred to as an Australian ballot, though it's more properly referred to as a preferential ballot. (The Australian ballot is in contrast to the Austrian ballot, in which you hold an election and the Holy Roman Emperor gets as many votes as he wants.)

In a Hare count, the first-ranked choices on each ballot are tallied. If no candidate has a majority, the candidate with the lowest vote count is eliminated and ballots for that candidate are redistributed among the remaining candidates based on their second-ranked choices. We continue until one candidate has a majority of the votes.

Let's begin by positing a file of ballots like this:

```
Moby Dick
Prisoner of Zenda
Pride and Prejudice
Tom Sawyer
War and Peace
No Award
/*
0 4 1 3 2 5
0 0 0 0 0 1
0 0 0 0 0 1
4 3 2 5 1 0
0 2 1 3 0 0
1 4 3 6 2 5
0 0 1 0 2 0
0 0 1 0 2 0
```

Here, we have a list of the candidates, a separator and a line containing each ballot as a list of ranks. The first voter ranked *Pride and Prejudice* first, followed by *War and Peace*, *Tom Sawyer*, *Prisoner of Zenda* and *No Award*. ("No Award" is a special case for Hugo voting: it's provided in case the voter feels nothing deserves the award that year.)

We can begin a Perl script to count the ballots:

```
#! /usr/local/bin/perl -w
# tally a Hugo-like preferential ballot
# $Id: tally,v 1.12 2000/05/10 17:09:58 jeff Exp $

use strict;
my %candidates;
  # hash of candidates names still alive
my %tally;
  # hash of arrayrefs of votes per
  # candidate for each runoff round
```

Because we've used `strict`, we must declare variables. We need a hash of the candidates who have not yet been eliminated, `candidates`. We'll need a similar hash, by candidate, of the votes at each elimination round, `tally`. There are two interesting features of note here. First, we haven't said what's stored against the candidate's name in the hash `candidates`. It doesn't matter: Once the candidate is eliminated, we delete the name from the hash, so the presence of a candidate means it's still active. Second, `tally` is actually a hash of arrays, or more accurately, a hash of pointers to arrays, or "arrayrefs" in Perl-speak. How we add and extract data from this structure will become apparent as we go along.

We're assuming the input file is on our standard input, even though we could have provided a named file on the command line instead. We need to grab the contents of the file nonetheless. We could build a very C-like loop to do this:

```
#  read candidate names
while( <> ) {
   chomp;
   last  if( /\/\*/ );
   push(@candidates, $_);
}

#  now read the ballots themselves
$count = 0;
while( <> ) {
   chomp;
   $i = 0;
```

```
   while( s/\s*(\d)(.*)/$2/ ) {
     $ballots{$candidates[$i++]}[$count] = $1;
   }
   $count++;
}
```

Instead, we'd rather do this in a Perl-like way. For example, to read the candidates' names, we use:

```
my @candidates;
  # ordered list of all candidates

#  grab the candidate list...
{
 local $/ = "/*\n";
 chomp ($_ = <>);
 @candidates = split "\n";
}
```

We redefine the record separator, `$/`, to be the marker between the candidates and the ballots. Once we've read the whole candidate list–`chomp`ing the record separator in the process–we `split` the list into our `candidates` array. We do this whole operation inside a block to insulate the `$/` redefinition. Notice that we've defined an array `@candidates`, which we shouldn't confuse with the hash `%candidates`. Perl keeps them distinct by the way they're referred to; so can we. We then invoke a similar loop to read the ballots themselves from the file:

```
my @ballots;
   # ordered list of all ballots

#  now grab the ballots
while(<>) {
   my %ballot;
   @ballot{@candidates} = split;
    # use the hash slice
   push @ballots, {%ballot};
}
```

We use a hash slice to split up each line of the ballot into its component rankings. What's a hash slice? In effect, the line `@ballot{@candidates} = split;` is a Perl multiple assignment that says

```
($ballot{$candidate[0]},
 $ballot{$candidate[1]}, ... ) =
split($_, / /);
```

This means we have values like `$ballot{'Moby Dick'} = 4`. We end the loop by pushing the hash `%ballot` onto the array `@ballots`.

We talked about the hash `%candidates` earlier. We now need to populate it for the remaining active candidates–at this point, all of them. Again, we use a hash slice:

```
@candidates{@candidates} = @candidates;
```

Again, this is a multiple assignment that says

```
($candidates{$candidates[0]},
 $candidates{$candidates[1]}, ...) =
($candidates[0], $candidates[1], ...);
```

(Notice how we can tell when we're referring to the hash `%candidates` and when we really mean the array by checking what brackets we use?)

Counting the votes is a simple matter as we outlined above. We count the first-place votes, and if there is no candidate with a majority, we eliminate the lowest-ranked candidate, redistribute its second-place votes to the remaining candidates and repeat. Reduced to code, we say:

```
while (1) {
   my %results = one_round @ballots;
   if ($results{Winner}) {
    print "We have a winner!  ",
      "$results{Winner}\n";
    last;
   }
   warn "We have a loser! $results{Loser}\n";
   foreach (@ballots) {
    $_->{$results{Loser}} = 0;
    # throw away votes for eliminated
    #  candidate: make them "don't rank"
   }
   delete $candidates{$results{Loser}};
    # keeps @candidates the list
    #  of ACTIVE candidates
   die "No more candidates!"
        unless %candidates;
}
```

This infinite loop tallies each round of ballots. We'll look at the subroutine to do that below. For the moment, though, you need to know only that `one_round()` returns a hash containing two possible key values, `Winner` or `Loser`. If there was a majority winner, it is contained in `$results{Winner}`; if there was not, the candidate with the least votes is in `$results{Loser}`. As a side-effect, `one_round()` also populates the `%tally` array, as we'll see anon.

If we actually have a winner, we print a message and break out of the infinite loop; if not, we print the name of the candidate to be eliminated and set that candidate's rank to zero on each ballot. In other words, we assert that no one voted for the candidate. Then we eliminate the candidate in `%candidates` using `delete`. If there are no more candidates for the next round, we have a real problem and we take a fatal exit.

The loop is fairly simple in concept and made simple in expression by our notation, but a lot of the action is hidden. For example, what magic happens in `one_round`? Quite a bit, it turns out.

Magic Routines and Results

The `one_round` routine is deceptively simple:

```
#  return a winner or loser
sub one_round {
   my @pref_list = pref_list @_;
   my $min = @pref_list;
   my $winner;
   my $loser = "No losers!";

   foreach my $candidate (keys %candidates) {
    my @votes =
     grep {$_ eq "$candidate"} @pref_list;
      # pref_list is a list of
      # highest-ranked active
      # candidates from each ballot;
      # we "grep" to get just
      # the current candidate.
    $winner = $candidate
     if(@votes > @pref_list/2);
    ($loser, $min) =
      ($candidate, scalar @votes)
       if @votes < $min;
      # "scalar" ensures that count
      # is used
    push(@{$tally{$candidate}}, scalar @votes);
   }
   return $winner ?
    (Winner=>$winner) : (Loser=>$loser);
}
```

The largest magic, which we'll defer examining, happens in the first line, where we use the routine `pref_list` to create a similarly named array containing an array of the highest-ranked candidate on each ballot. That is, `@pref_list` contains something like "Moby Dick, Tom Sawyer, Moby Dick, No Award, War and Peace, War and Peace…" We set the minimum number of votes–that is, the lowest tally we've seen so far–to the length of that array. We declare local variables for `Winner` and `Loser`.

We need to determine the tally for each remaining candidate, that is, any candidate with a key in `%candidates`. If we had `@pref_list` in a file, we would say:

```
grep "Moby Dick" pref_list | wc -l
```

In Perl, we do the same thing with the `grep` line in the code fragment. The count of entries in the resulting `@votes` array gives us the number of votes received. If that number is greater than half the length of `@pref_list`–that is, the number of ballots still active–we have a winner. Why not just return here? Because we really need to know the vote tallies for the remaining candidates in this runoff round. If the vote count for this candidate is less than the previous lowest talley, we save that data with a multiple assignment. Notice that we're explicitly saying `scalar @votes`, which gives us the number of entries in the array here; if we didn't, we'd just get the first entry in that array assigned to `$min`.

Next, we add the vote count to the `tally` hash entry for each candidate. The odd bit of syntax `@{$tally{$candidate}}` is the array referred to by the hash entry. Remember that `tally` is actually a hash containing arrayrefs, which we need to dereference before adding the new votes. Finally, falling out of that loop, we return one of `Winner` or `Loser`.

We finish up our subroutines by looking at what happens under the covers of `pref_list`:

```
# make a preference list
sub pref_list {
   my @ballots = @_;
   my @pref_list;
    # an array of ballots
    # each, an array of active candidate
    # names, in preference order

   foreach (@ballots) {
    my %rank = reverse(%$_);
    delete $rank{0};
       # delete any "don't rank"s
    push @pref_list, $rank{(sort keys %rank)[0]}
         if(%rank);
   }
   @pref_list;
}
```

We begin with the array of ballots passed as an argument. Remember that this is actually an array of hashes indexed by candidate. For each ballot in the list we are presented, we invert the sense of the hash using `reverse(%$_)`, which gives us a list of candidates keyed by rank. In other words, we started with entries in `ballots` like `$ballot{'Moby Dick'} = 4`, which we've turned into entries like `$rank{4} = Moby Dick`. In the inversion, any candidate with a rank of zero–that is, one who received no vote or was eliminated–is

overlaid into `$rank{0}`, which we delete. Then if there are still entries in the hash `%rank`, we add the entry with the lowest sorting key (the highest-ranked candidate) to the end of `pref_list`. It's `pref_list` we return once we've examined all the ballots.

This highest-ranked business bears a moment's examination. In the normal course of events, when one candidate is eliminated, we want to promote the ranks of all the remaining candidates. If *Moby Dick* were eliminated, for example, we'd promote everything ranked lower by one position, even if *Moby Dick* weren't in first place. But that doesn't really matter. All we really need to do is change *Moby Dick* to "unvoted," since we're using the highest *remaining* rank in the `pref_list` subroutine. Put another way, the absolute rank is unimportant, it's the relative rank remaining that matters.

But what about the results? Even though we've printed out the name of the winner in our main `while` loop, it would be nice to see the totals at each runoff. We could use a simple loop like the following:

```
foreach my $name (@candidates) {
   print $name;
   while( my $n = pop(@${tally{$name}}) ) {
     print $n;
   }
   print "\n";
}
```

There are two problems with this scheme. First, it's in candidate order rather than in some order related to the vote totals. Second, it doesn't format the tally grid in a very appealing way.

We can solve the first problem by sorting the `tally` hash. But we want to sort the hash by the last entry in the array, that is, the last vote total each candidate received before being eliminated. We can use a convenient bit of Perl syntax, and say `@{$tally{$x}}[-1]` to get the last vote count for candidate `$x`. Alternately, we can use a different bit of syntactic sugar and say `$tally{$x}->[-1]`, as we've done below. We want to sort the `tally` list in reverse order of that entry:

```
sort {
   $tally{$b}->[-1] <=> $tally{$a}->[-1];
    } keys %tally
```

This uses a slightly more complicated comparison than the normal Perl `sort` example:

```
sort { $a <=> $b } @array
```

To print out the full results, we say:

```
my $sort =
   sub { $tally{$b}->[-1] <=> $tally{$a}->[-1] };

foreach (sort $sort keys %tally) {
   my @a = @{$tally{$_}};
   printf "%-20s" . "%5d"x@a . "\n", $_, @a;
}
```

We could have printed the vote counts by `shift`ing them out of the array, but this shows them printed in a single statement. Note how we've constructed the `printf` format specifier out of pieces, the central one of which (which prints the numbers) depends on the size of the array `@a`. We also extract the `sort` into an anonymous subroutine just to keep our `foreach` line uncluttered. This gives us the following output:

```
Pride and Prejudice   355  359  408  455  573
Prisoner of Zenda     235  236  264  321  414
War and Peace         178  179  215  272
Tom Sawyer            197  202  211
Moby Dick             147  152
No Award               56
```

Notice that we're counting on the array being in chronological order, which we got by using the earlier `push()`.

The Australian ballot is in contrast to the Austrian ballot, in which you hold an election and the Holy Roman Emperor gets as many votes as he wants.

Notice also that *Tom Sawyer*, which finished well-ahead of *War and Peace* in the first two rounds of voting, was eliminated in the very next round. Despite our good intentions and careful programming, it's possible to concoct input data that produce even more egregious outcomes: Arrow's Impossibility Theorem in action. And–now that you have the code–we leave it to you to convince yourself of this.

Finishing Up

The program we've presented is fully functional as far as it goes. Using idiomatic Perl tightens up the notation over the more complicated code we would have used in a C or Pascal version. In fact, there are roughly 75 lines of Perl code here, not counting didactic comments, in contrast with about 900 in the corresponding C program. However, we've deliberately left out some things which take up space in the C version. We present them to you now as exercises:

• If there is a two-or-more-way tie for first place, our program will declare the first of the tied candidates a loser, and go on to the next round. Fixing the code to recognize this situation is an easy exercise. How would you resolve ties like this? Remember that there is no "right" way: "fairness" is in the eye of the beholder.

• A related situation occurs if there is a tie for last in earlier rounds, but you can add the option of discarding all the last-placers at once. How would you implement this?

• How would you wrap this code into a program to tally all 13 Hugo categories?

• In a Hare-tallied election, the second-place winner is not necessarily the candidate who comes in second in the first tally. It's actually the candidate who would have won if the first-place winner were eliminated. Fix the code to determine second place, third place and so on. Show the intermediate vote tallies.

• When we started writing this column we were intending to talk about the input problem, too. How would you write a program to capture the data into the input file we've used for this program?

• As important as input is validation of the data. We've just blithely assumed that the data we're seeing has no invalid ballots. Two kinds of invalid ballots are ballots on which two candidates are given the same (nonzero) rank, and ballots that have a gap in their rankings. How would you detect such errors? Other errors? Can some of these botched ballots be fixed? Should they?

We've deliberately simplified both our discussion of electoral theory and the specific history of the Hugo Awards, just touching on enough of the high points to set up the problem. (This is a problem we have a bit of experience with. Copeland and his wife, Liz, have actually administered the Hugo Awards, and wrote the code that does the real tallying.) On the issue of the history of voting, there are some good references out there if you want to do further study. We can suggest the following:

• Rob Lanphier's article on "Perl, Politics, and Pairwise Voting" in the Autumn 1996 issue of *The Perl Journal* (see `http://itknowledge.com/tpj/contents.html#issue3`).

• Richard Niemi and William Riker's "The Choice of Voting Systems" in the June 1976 issue of *Scientific American*.

• Donald G. Saari's book, *Basic Geometry of Voting*, published by Springer-Verlag New York Inc., 1995, ISBN 3-540-60064-7.

In our coding efforts, we owe quite a bit to Tom Christiansen and Nathan Torkington's *Perl Cookbook* (published by O'Reilly & Associates Inc., 1998, ISBN 1-56592-243-3). Tom and Nat's book provides more than 700 pages of interesting problems with copious discussion of the hows and whys of the Perl tricks used to solve them.

On the other hand, if you're interested in the past history of the Hugos and other literary awards, check out the AwardWeb page at `http://dpsinfo.com/awardweb`. If you're a rules lawyer, you can find the codified process for Hugo nomination and voting in the constitution of the World Science Fiction Society at `http://www.worldcon.org/rules.html`. Or for a more up-to-date version, see `http://www.chicon.org/wsfs/constit.htm`.

Next month, we'll explore a new and different problem that momentarily distracted us. Until then, happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at Minolta-QMS 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` *or alternately at* `ftp://ftp.expert.com/pub/Work`.