Fast Algorithms for Counting Ranked Ballots

Jeffrey C. O'Neill
jco8@cornell.edu

1 Introduction

This paper shows how some vote-counting methods can be implemented significantly faster by organizing ranked-ballot data into a tree rather than a list. I will begin by explaining how the tree data structure works and then apply it to Meek's method and Condorcet voting.

2 Tree-Packed Ballots

The most basic way of storing ballots is in a list. For example, suppose Alice, Bob, and Cindy are candidates and we have ten voters. The votes could be stored in a list, where each line corresponds to a ballot, and within each line, the candidates are listed in order of preference. I call this raw or unpacked ballot data, and an example is shown in Figure 1.1.

In this example, as is inevitable in any real election with ranked ballots, some voters will cast the exact same ballot. Instead, one could store only one copy of duplicate ballots along with the number of times the ballot occurred. I call this list-packed ballots. Figure 1.2 shows the same ballots from Figure 1.1 packed into a list.

Alice, Cindy
Cindy
Cindy, Alice
Bob
Bob
Alice
Cindy, Alice
Alice
Alice, Bob, Cindy
Bob

Figure 1.1: Raw ballots.

3	Bob
2	Cindy, Alice
2	Alice
1	Cindy
1	Alice, Cindy
1	Alice, Bob, Cindy

Figure 1.2: List-packed ballots.

Many vote-counting methods can use list-packed ballots instead of raw ballots and save computations. For example IRV, ERS97 STV, and Meek's method can all use list-packed ballots but Cambridge and Irish STV cannot. The reason Cambridge and Irish STV cannot is that the outcome is dependent on the order of the ballots, and order information is lost with list-packed ballots.

The ballots, however, can be packed even more densely into a tree, what I call tree-packed ballots. Figure 1.3 shows the same ballots packed into a tree. The root of the tree lists the total number of ballots, which is ten. From the root, branches go downward corresponding to the first-ranked candidates. The subsequent nodes list the number of times that candidate was ranked first on a ballot. Note that these three numbers add up to ten. The second level corresponds to the second-ranked candidates listed after the corresponding first-ranked candidates. Note that no candidate is ever ranked second after Bob. Further, note that four ballots have Alice first, but only two ballots list a candidate second after Alice. This is because two of the four voters who listed Alice first did not rank a candidate second.

For the three data structures, the size of the data struc-

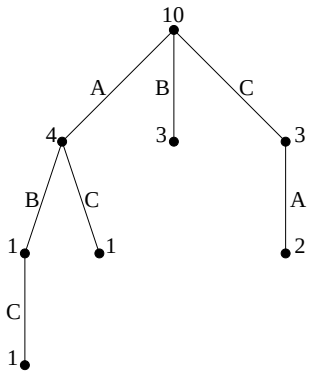

Figure 1.3: Tree-packed ballots.

Data Structure	Time
Load and No Packing	17.7
Load and List Pack	26.7
Load and Tree Pack	31.1

Table 1.1: Comparison of loading and packing times (in seconds).

Method	List	Tree
SNTV	0.6	
IRV	1.2	
ERS97 STV	5.5	
BC STV	4.7	
Meek STV	32.8	5.9 (18%)
Warren STV	30.8	3.0 (10%)
Condorcet	13.3	7.7 (59%)

Table 1.2: Timing of vote-counting methods with list-packed and tree-packed ballots (in seconds). The percentages in parenthesis indicate the computation time of the tree-packed implementation relative to the list-packed implementation.

ture corresponds to the number of entries, which is the number of times that candidate names are stored. For example, the size of the data structure in Figure 1.1 is 15, the size of the data structure in Figure 1.2 is 10, and the size of the data structure in Figure 1.3 is 7 (the root node isn't counted). Table 1.3 shows the sizes of the three data structures for the ballots from eight elections. B is the number of ballots, C is the number of candidates, and S is the number of seats to be filled.

List-packed ballots are 65% of the size of raw ballots. Tree-packed ballots are 45% of the size of list-packed ballots and 29% of the size of raw ballots. I expect the computation time of a particular implementation to be roughly proportional to the size of the data structure used. Thus, I expect the computation time with tree-packed ballots to be about 45% of the computation time with list-packed ballots. The more complicated data structures will also add some overhead that will increase the computation time to some extent.

Before presenting the details of implementing vote-counting methods with the different data structures, I will present the timing results with the different data structures. The timing results should only be considered in a rough sense since the efficiency of the particular implementations may vary. All timing results are cumulative for the above eight elections and are in seconds. First, the times in seconds for loading, loading and list packing, and loading and tree-packing are shown in Table 1.1.

Next I compare the computation times for a number of vote-counting methods using list-packed and tree-packed ballots. Because the relationship between raw and list-packed ballots is obvious, those times are not compared in this paper. [1] Further, only the slower methods are implemented with tree-packed ballots because these are the only ones that are in need of improvement. The methods are single non-transferable vote (SNTV), instant runoff voting (IRV), Electoral Reform Society STV (ERS97 STV), STV rules proposed for British Columbia in 2005 (BC STV), Meek STV, Warren STV, and Condorcet. [2] The computation times are shown in seconds in Table 1.2. The percentages in parentheses indicate the computation time of the tree-packed implementation relative to the list-packed implementation.

While we expected the computation times with tree-packed ballots to be 45% of the times for list-packed ballots, they are much faster for Meek and Warren STV. Why this is so will be explained below.

[1] Implementing a particular method with raw or list-packed ballots uses nearly the same code. The code iterates over the raw ballots or iterates over the list-packed ballots. The computation time is simply proportional to the number of loop iterations. In contrast, with tree-packed ballots, the code needs to be rewritten from scratch as is discussed below.

[2] The timing for Condorcet is only for computing the pairwise comparison matrix. Computing the Condorcet winner from the pairwise comparison matrix is generally much faster than computing the pairwise computation matrix.

3 Meek STV with Tree-Packed Ballots

I will now give the details of how to implement Meek STV using tree-packed ballots. The process is very similar for Warren STV. A full description of Meek STV is beyond the scope of this paper [1, 2, 3]. Instead, I will present the details most relevant to the fast implementation.

In each stage of counting votes with Meek STV, all the votes must be counted from scratch. This is distinct from other STV methods where some votes are simply transferred from one candidate to another and a full recount is not necessary at each round. With Meek STV, each candidate is assigned a fraction, f[c], where c denotes the candidate. At the beginning of the count, all the fractions are 1.0, and the fractions remain 1.0 as long as a candidate is under the quota. When a candidate has more than a quota, the fraction essentially discounts the value of that candidate's votes to bring the candidate back down to a quota. With a discount less than 1.0, the subsequently ranked candidates on a ballot will receive a portion of the vote.

In each round of a Meek STV count, the fractions f[c] will be updated and the ballots recounted. The following is a segment of Python pseudo-code for counting ballots for one round of a Meek count. Note that it uses list-packed ballots. The ith packed ballot is b.packed[i] and the corresponding weight of that packed ballot is b.weight[i].

```
# Iterate over all of the ballots.
for i in range(nBallots):
  # Each ballot is worth one vote.
  remainder = 1.0
  # Iterate over the candidates on this ballot.
  for c in b.packed[i]:
    # If the candidate is already eliminated
    # then skip to the next candidate on the
    # ballot.
    if c in losers:
      continue
    # This candidate gets a portion
    # of this ballot.  For the first non-losing
    # candidate on the ballot, the remainder will
    # be 1.0. If the candidate is under quota,
    # then f[c] is also 1.0 and this candidate
    # gets all of the ballot.  Otherwise the
    # candidate gets less than the full value,
    # and will share the ballot with
    # subsequently ranked candidates.
    count[c] += remainder * f[c] * b.weight[i]
    # Calculate how much of this ballot remains,
    # if any, to be counted for subsequently
    # ranked candidates.
    remainder *= 1 - f[c]
    # Stop if this ballot is used up.
    if remainder == 0:
      break
```

This code can be rewritten to use tree-packed ballots. The computations are exactly the same as before, they are just done in a different order so that similar computations can be done together. Consider the ten ballots presented above. Alice is ranked first on four ballots. With list-packed ballots, it would take three loop iterations to count these three ballots, but with tree-packed ballots all the first-place votes for Alice are counted at the same time, thus saving computations.

The code is more complicated, because it involves a depth-first traversal of the tree. The following shows how the nodes of the tree are accessed and also the order of a depth-first traversal.

```
tree[n] = 10
tree[Alice][n] = 4
tree[Alice][Bob][n] = 1
tree[Alice][Bob][Cindy][n] = 1
tree[Alice][Cindy][n] = 1
tree[Bob][n] = 3
tree[Cindy][n] = 3
tree[Cindy][Alice][n] = 2
```

A convenient way to implement the depth-first traversal is to use a recursive subroutine. Note that the subroutine calls itself by passing one branch of the tree, which is just a smaller tree, and possibly a diminished value for the remainder.

```
def updateCountMeek(tree, remainder):
  # Iterate over the next possible candidates.
  for c in tree.nextCands():
    # Copy the remainder for each iteration.
    rrr = remainder
    # Skip over losing candidates.
    if c not in losers:
      # Count the votes as before but weight with
      # the tree-packed data instead of the
      # list-packed data.
      count[c] += rrr * f[c] * tree[c][n]
      # Calculate how much of this ballot remains,
      # if any, to be counted for subsequently
      # ranked candidates.
      rrr *= 1 - f[c]
    # If there are any candidates ranked after
    # the current one and this ballot is not used
    # up, then recursively repeat this procedure.
    if tree[c].nextCands() != [] and rrr > 0:
      updateCountMeek(tree[c], rrr)
```

The initial call to the subroutine uses the base of the tree, and as before, the initial value of the remainder is 1.0

```
updateCountMeek(self.b.tree, 1.0)
```

Now that I have explained the fast algorithm, I can explain why it works much faster than expected. The unexpected speed increase arises from the fact that in any STV election, it is overwhelmingly the top choices on the ballots that are counted. In the first round of a Meek election, only the first-ranked candidates are

counted. Consider the ballots for the Dublin North 2002 election. With list-packed ballots, one needs to count the 138,647 weighted ballots, but with tree-packed ballots, one needs to count only the twelve nodes of the tree corresponding to the first rankings of the twelve candidates. As the rounds progress, more and more nodes in the tree will be needed for the count, but generally this will be far less than the total number of nodes in the tree and even further less than the number of list-packed ballots.

Readers who understand the differences between Meek STV and Warren STV will immediately realize why Warren STV is much faster than Meek STV with the tree-packed ballots: Warren STV is less likely than Meek STV to use lower-ranked choices on a ballot.

4 Condorcet with Tree-Packed Ballots

Tree-packed ballots can also be used to compute the pairwise comparison matrix in a Condorcet election. The pairwise comparison matrix, pMat[c][d], counts the number of times that candidate c is ranked higher than candidate d on the ballots. Computing the pairwise comparison matrix is straightforward with list-packed ballots:

```
# Iterate over all the ballots.
for i in range(nBallots):
  # Copy the list of candidates.
  remainingC = candidates[:]
  # Iterate over the candidates the ballot.
  for c in b.packed[i]:
    # Get list of lower-ranked candidates.
    remainingC.remove(c)
    # Iterate over all lower-ranked candidates.
    for d in remainingC:
      # c is ranked higher than d.
      pMat[c][d] += b.weight[i]
```

This code can also be rewritten to use tree-packed ballots. As before it involves the depth-first traversal of the tree.

```
def ComputePMat(tree, remainingC):
  # remainingC is a list of candidates not higher in
  # the ballot than the current candidate.  Initially
  # it is a list of all the candidates.
  # Iterate over the next possible candidates.
  for c in tree.nextCands():
    # Copy the list of remaining candidates.
    rc = remainingC
    # Remove candidate from remaining list.
    rc.remove(c)
    for d in rc:
      # Current candidate is ranked higher than
      # candidates in remaining list.
      pMat[c][d] += tree[c][n]
    # Continue if more candidates.
    if tree[c].nextCands() != []:
      ComputePMat(tree[c], rc)

# First call is with entire tree and list of all
# candidates.
ComputePMat(tree, allCands)
```

Computing the pairwise comparison matrix is faster with tree-packed ballots, but the improvement is not nearly as great as for Meek STV. The reason for this is that computing the pairwise comparison matrix requires traversing the entire tree, thus the computation times are roughly proportional to the relative sizes of the data structures. The overhead involved with using tree-packed ballots makes the implementation with tree-packed ballots a little slower than expected.

5 Conclusions

Using tree-packed ballots instead of other data structures can greatly increase the speed of some vote-counting methods. Such speed improvements need to be weighed against the time needed to create the tree-packed ballots and the cost of maintaining more complex code. Meek and Warren STV are approximately five and ten times faster, respectively, with tree-packed ballots than with list-packed ballots. Such enormous speed improvements clearly outweigh the costs. In contrast, with Condorcet voting, the time saved is about equal to the time required for tree-packing the ballots so any benefits are minimal. Other methods, such as ERS97 STV and BC STV, are so fast with list-packed ballots that tree-packed ballots are clearly not beneficial.

My implementation of all of the vote counting methods mentioned in this paper (and others) is available for download at `http://stv.sourceforge.net`.

6 References

[1] Nicolaus Tideman, *The Single Transferable Vote*, Journal of Economic Perspectives, Vol. 9, No. 1, pp. 27-38 (1995).

[2] B. L. Meek, *A New Approach to the Single Transferable Vote*, Voting matters, Issue 1, p. 1 (Mar. 1994).

[3] I. D. Hill, B. A. Wichmann, and D. R. Woodall, *Algorithm 123 – Single Transferable Vote by Meek's Method*, Computer Journal, Vol. 30, No. 3, pp. 277-281 (1987).

Election	B/C/S	Raw	List	Tree
Dublin North 2002	43,942/12/4	218,933	138,647	57,568
Dublin West 2002	29,988/9/3	132,726	69,860	23,730
Meath 2002	64,081/14/5	298,106	174,737	74,105
Cambridge 1999 City Council	18,777/29/9	106,585	90,816	47,813
Cambridge 2001 City Council	17,126/28/9	95,440	79,385	40,566
Cambridge 2001 School Committee	16,489/16/6	66,254	33,860	12,907
Cambridge 2003 City Council	20,080/29/9	115,232	98,055	54,182
Cambridge 2003 School Committee	18,698/14/6	66,389	29,637	9,764
Total		1,099,665	714,997	320,635

B/C/S = Ballots/Candidates/Seats

Table 1.3: Sizes of the three data structures for the eight elections. The size of a data structure is the number of entries. See the text for more details.