A Fast Majority Vote Algorithm

J Strother Moore

Technical Report #32

Institute for Computing Science

2100 Main Building

The University of Texas at Austin

Austin, Texas 78712

(512) 471-1901

This work was done while the author was in the Computer Science Laboratory of SRI International, Menlo Park, California, 94025. The research reported here was supported in part by NASA Contract NAS1-15528, NSF Grant MCS-7904081, and ONR Contract N00014-75-C-0816.

Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712

Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape.

Key Phrases: fault-tolerance, redundancy, searching

1. Introduction

Reliability may be obtained by redundant computation and voting in critical hardware systems. What is the best way to determine the majority, if any, of a multiset of n votes? An obvious algorithm scans the votes in one pass, keeping a running tally of the votes for each candidate encountered. If the number of candidates is fixed, then this obvious algorithm can execute in order n. However, if the number of candidates is not fixed, then the storage and retrieval of the running tallies may lead to execution time that is worse than linear in the number of votes -- such an algorithm could run in order n^2.

If the votes can be simply ordered, an algorithm with order n execution time can be coded first to find the median using the Rivest-Tarjan algorithm [2] and then to check whether the median received more than half the votes. The Rivest-Tarjan algorithm is bounded above by 5.43 n - 163 comparisons, when n>32.

In this paper we describe an algorithm that requires at most 2n comparisons. The algorithm does not require that the votes can be ordered; only comparisons of equality are performed.

2. The Algorithm

Imagine a convention center filled with delegates (i.e., voters) each carrying a placard proclaiming the name of his candidate. Suppose a floor fight ensues and delegates of different persuasions begin to knock one another down with their placards. Suppose that each delegate who knocks down a member of the opposition is simultaneously knocked down by his opponent. Clearly, should any candidate field more delegates than all the others combined, that candidate would win the floor fight and, when the chaos subsided, the only delegates left standing would be from the majority block. Should no candidate field a clear majority, the outcome is less clear; at the conclusion of the fight, delegates in favor of at most one candidate, say, the nominee, would remain standing--but the nominee might not represent a majority of all the delegates. Thus, in general, if someone remains standing at the end of such a fight, the convention chairman is obliged to count the nominee’s placards (including those held by downed delegates) to determine whether a majority exists.

Thus our algorithm has two parts. The first part pairs off disagreeing delegates until all remaining delegates agree. We call this the "pairing" phase. Perhaps nonobviously, pairing can be done with n comparisons. If pairing leaves any delegates standing then those delegates unanimously favor a single candidate--the nominee--who must be in the majority if a majority exists. The second part of the algorithm, called the "counting" phase, determines whether the nominee received more than half the votes. The counting phase obviously requires at most n comparisons. The focus of this paper is on the pairing phase.

Here is a bloodless way the chairman can simulate the pairing phase. He visits each delegate in turn, keeping in mind a current candidate CAND and a count K, which is initialized to 0. Upon visiting each delegate, the chairman first determines whether K is 0; if it is, the chairman selects the delegate’s candidate as the new value of CAND and sets K to 1. Otherwise, the chairman asks the delegate whether his candidate is CAND. If so, then K is incremented by 1. If not, then K is decremented by 1. The chairman then proceeds to the next delegate. When all the delegates have been processed, CAND is in the majority if a majority exists.

Proof: Suppose there are N delegates. After the chairman visits the Ith delegate, $1 \leq I \leq N$, the delegates he has processed can be divided into two groups: a group of K delegates in favor of CAND, and a group of delegates that can be paired in such a way that paired delegates disagree. From this invariant we may conclude, after processing all of the delegates, that CAND has a majority, if there is a majority. For suppose there exists an X different from CAND with more than N/2 votes. Since the second group can be paired, X receives at most (N-K)/2 votes from that group. Thus, X must have received a vote from the first group, contradicting the fact that all votes in the first group are for CAND.

3. Discussion

Note that the algorithm fetches the elements of A in linear order. Thus, the algorithm can be used efficiently when the number of votes is so large that they must be read from magnetic tape. One tape rewind may be necessary after the first phase.

If it can be assumed that a majority exists, the counting phase may be eliminated. The algorithm can then be implemented to poll the delegates in real time (rather than store the votes for batch processing).

A FORTRAN version of this algorithm has been mechanically proved correct by the FORTRAN verification system described in [1].

In [3], Misra and Gries describe a generalized version of this algorithm that finds the candidates that receive more than n/k votes. They also discuss the complexity of finding repeated elements and show that this algorithm is optimal among algorithms based on comparing candidates.

References

1. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

2. D. E. Knuth. *The Art of Computer Programming. Volume 3/ Searching and Sorting.* Addison-Wesley Publishing Co., Reading, MA, 1973.

3. J. Misra and D. Gries. Finding Repeated Elements. In *Science of Computer Programming 2*, North Holland, 1982, pp. 143-152.

Table of Contents