International Journal of Reliability, Quality and Safety Engineering
Vol. 1, No. 1 (1994) 95–102
© World Scientific Publishing Company

THRESHOLD VOTING IS FUNDAMENTALLY SIMPLER THAN PLURALITY VOTING

BEHROOZ PARHAMI
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA

Received 18 June 1993
Revised 21 December 1993

We show that n-way plurality voting on a large unordered object space has time and space complexities of $\theta(n^2)$ and $\theta(n)$, respectively. If the object space is ordered, then sorting can be used to reduce the time complexity to the optimal $\theta(n \log n)$. We then prove that weighted t-out-of-Σv_i threshold voting on such an object space has time complexity $O(np)$ and needs working storage space for only p objects, where $p = \lfloor (\Sigma v_i)/t \rfloor$. Thus, unless t is very small, threshold voting is considerably simpler than plurality voting. As a corollary, weighted majority voting can be performed in linear time with working storage for a single input object.

Keywords: Voting Scheme; Algorithm Complexity; N-Modular Redundancy; Threshold Voting.

1. Introduction

Voting is an important operation in the realization of ultrareliable systems that are based on the multi-channel computation paradigm. Voting is required whether the multiple computation channels consist of redundant hardware units, diverse program modules executed on the same basic hardware, identical hardware and software with diverse data, or any other combination of hardware/program/data redundancy and/or diversity.

With very few exceptions,[1–3] previous studies of voting have been restricted to simple input object spaces; primarily bits or numerical values. These are instances of small and totally ordered object spaces for which efficient linear-time (based on vote tallying) and $O(n \log n)$-time (based on sorting) algorithms can be devised. Widespread concern with computer system dependability gives rise to the need for sophisticated voting schemes on various other data types.

Before stating the objectives of this note, let us define the problem of weighted voting and introduce some needed notation.

1.1. *Definition*

Given n input data objects $x_1, x_2, \ldots, x_n$, with associated positive votes (weights) $v_1, v_2, \ldots, v_n$, compute the output y with the vote tally $w = \Sigma_{\{i|y=x_i\}} v_i$ such that:

WEIGHTED THRESHOLD VOTING SCHEMES:
Consensus voting: $w = \Sigma v_i$
Byzantine voting (see the notes below): $w > \frac{2}{3}\Sigma v_i$
Majority voting: $w > \frac{1}{2}\Sigma v_i$
m-out-of-n voting $(v_1 = v_2 = \cdots = v_n = 1) : w \geq m$ (if $m \leq \frac{n}{2}$, y may be non-unique)
t-out-of-Σv_i (generalized m-out-of-n) voting: $w \geq t$ (if $t \leq \frac{1}{2}\Sigma v_i$, y may be non-unique)
WEIGHTED PLURALITY VOTING SCHEME:
No other y' gets more votes (if $w \leq \frac{1}{2}\Sigma v_i$, y may be non-unique).

The above Byzantine voting scheme is a generalized form of the unweighted version used in distributed computing systems[4] where n independent n-way voting nodes/sites must arrive at consistent conclusions in the presence of fewer than $\frac{n}{3}$ faulty nodes (Byzantine faulty nodes may try to confuse other nodes by presenting to them inconsistent values). In this generalized version, less than $\frac{1}{3}$ of votes/weights can be associated with faulty nodes. □

1.2. *Notation*

Asymptotic behavior of functions (e.g., time and space bounds) are specified/compared as:

$f(n) = O(g(n))$ iff there exist constants c_1 and n_1 such that $f(n) \leq c_1 g(n)$ for all $n \geq n_1$.

$f(n) = \Omega(g(n))$ iff there exist constants c_2 and n_2 such that $f(n) \geq c_2 g(n)$ for all $n \geq n_2$.

$f(n) = \theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; i.e., $c_2 g(n) \leq f(n) \leq c_1 g(n)$ for large n.

Hence O, Ω and θ convey upper, lower, and exact bounds for growth rate, respectively. □

In this paper, we show that with a large unordered input object space (large, so that linear-time tallying of votes in an array of size equal to the object space size cannot be used; unordered, so that sorting is inapplicable), $\Omega(n^2)$ comparison operations and $\Omega(n)$ working storage are required in the worst case for any plurality voting algorithm. We then present a worst-case optimal weighted plurality voting algorithm with worst-case $O(n^2)$ and average-case linear running time in most application contexts. We also show that unless t is very small compared to Σv_i,

t-out-of-Σv_i threshold voting is intrinsically simpler in that it can be performed in time $O(np)$ and with working storage for p input objects, where $p = \lfloor (\Sigma v_i)/t \rfloor$.

2. Plurality Voting

Intuitively, in an unordered space, the equality of two objects can only be established by direct comparison (the transitivity feature of equality is of no help in the worst case where all objects happen to belong to distinct equivalence classes). Thus, $n(n-1)/2$ comparisons are needed in the worst case. Also, in the worst case, the last comparison may provide the only non-singleton equivalence class of size 2. Thus, working storage for $n-1$ objects and their associated vote tallies is needed. The following theorem formalizes these intuitive notions.

2.1. *Theorem*

With a large unordered input object space, n-way (weighted or unweighted) plurality voting has worst-case time and space complexities $\Omega(n^2)$ and $\Omega(n)$, respectively.

Proof. Consider instances of the weighted plurality voting problem for which input weights satisfy $v_1 < v_2 < \cdots < v_n$, the n input objects belong to $n-1$ or n equivalence classes (i.e., all distinct except possibly for x_q and x_r that are equal), and $v_i + v_j > v_k$ for all i, j, and k. Clearly, for these instances, the algorithm cannot produce a guaranteed correct output if it has not established whether the number of input equivalence classes is $n-1$ or n (the correct output is x_q if $x_q = x_r$ and is x_n otherwise). To determine whether $x_q = x_r$, the inputs x_q and x_r must be compared directly; all other comparisons only reveal inequality of values and provide no basis for judging whether $x_q = x_r$. Since the algorithm has no way of knowing in advance which two inputs will turn out to be equal, it may in fact sequence its operations in a way that the comparison of x_q and x_r is done after all other comparisons. Hence the need for $\Omega(n^2)$ worst-case comparisons. To complete the proof for the weighted case, we note that to compare all possible pairs of objects, $\Omega(n)$ working storage will be required. The proof for unweighted plurality voting is similar but slightly more involved. $\square$

Having established the above lower bounds, we now show that an $O(n^2)$-time, $O(n)$-space weighted plurality voting algorithm can be devised for a large unordered input space. Following is a high-level description of the algorithm. The qualifier "exact" in the title of the algorithm is to distinguish it from more complicated algorithms such as those needed for inexact and approval voting,[2,3] neither of which will be discussed in this paper.

2.2. *Algorithm*

Exact Weighted Plurality Voting — Large Unordered Object Space

Let z and u be the distinct-input and vote-tally vectors (z_j is the jth distinct input encountered, v_j is the vote tally for z_j)

```
$k := 1$; $y := z_1 := x_1$; $w := u_1 := v_1$
for $i = 2$ to $n$ do
    if $\exists j \leq k$ such that $x_i = z_j$
    then $u_j := u_j + v_i$
    else $k := k + 1$; $z_k := x_i$; $u_k := v_i$
    endif
endfor
for $i = 1$ to $k$ do
    if $u_i > w$ then $y := z_i$; $w := u_i$ endif
endfor □
```

The $O(n^2)$ worst-case time complexity of Algorithm 2.2 results from the $(n-1)$-iteration loop and the $O(n)$ linear search required in each iteration. The $O(n)$ worst-case space requirement is due to the fact that k can grow as large as $n - 1$. For an ordered object space, a sorting algorithm can be applied to inputs and the voting result obtained by a sequential scan of the sorted list. Thus, the time and space requirements become $O(n \log n)$ and $O(n)$, respectively.

The average-case performance of Algorithm 2.2 depends on the application context. In the context of dependable computing, each computation channel is highly reliable and thus the n channels produce matching results with very high probability (0.995 with 5-way voting and 0.999 reliability for each channel). Then the execution time is almost linear due to the fact that k remains 1 with very high probability. In other contexts, there may be a need to resort to more efficient search schemes (such as those based on hashing) for finding j inside the "for" loop in order to achieve subquadratic average-case running time.

With respect to average-case performance, it may be advantageous (depending on the details of algorithm implementation and the host machine) to precede the body of Algorithm 2.2 (or any other voting algorithm for that matter) with the following statement:

$$\text{if } x_1 = x_2 = \cdots = x_n \text{ then } (y,\, w) = (x_1,\, \Sigma v_i) \text{ stop endif}.$$

It may be even be worthwhile to handle the case of a single disagreeing input (the next most likely case) separately also before resorting to the general algorithm. These special tests increase the code length but improve the performance significantly.

3. Threshold Voting

Threshold t-out-of-Σv_i voting can be shown to be simpler than plurality voting, unless the threshold value t is very small. Consensus voting clearly has linear complexity. Less obvious are the following threshold voting algorithm and corresponding complexity theorem.

3.1. *Algorithm*

Exact Threshold Voting — Large Unordered Object Space

We need working storage space or "slots" for $p = \lfloor (\Sigma v_i)/t \rfloor$ different objects $z_1, z_2, \ldots, z_p$, each with an associated vote tally u_j.

```
$z_1 := x_1;\ u_1 := v_1;\ u_j := 0\ (2 \le j \le p)$
for $i = 2$ to $n$ do
    if $\exists\ u_j \ne 0$ such that $x_i = z_j$
    then $u_j := u_j + v_i$
    else if $\exists\ u_j = 0$
        then $z_j = x_i;\ u_j := v_i$
        else let $m = u_k$ be a minimum of all $u_j s$ $(1 \le j \le p)$
            if $v_i \le m$
            then $u_j := u_j - v_i\ (1 \le j \le p)$
            else $z_k := x_i;\ u_k := v_i;\ u_j := u_j - m\ (1 \le j \le p)$
            endif
        endif
    endif
endfor
$u_j := 0\ (1 \le j \le p)$
for $i = 1$ to $n$ do
    if $\exists\ j$ such that $x_i = z_j$ then $u_j := u_j + v_i$ endif
endfor
Output $z_j$ with $u_j \ge t$ □
```

Following is a textual description of Algorithm 3.1. The first input object and its associated vote are stored in the first slot and all other $u_j s$ are initialized to 0, thus designating the remaining $p - 1$ slots as empty. The next input object x_i to be considered is compared to the stored objects. If $x_i = z_j$ for some j, then u_j is incremented by v_i. If x_i is not equal to any z_j and fewer than p objects have been stored in the p slots, then (x_i, v_i) is stored in an available slot. If all of the p slots are occupied, then the minimum vote tally $m = u_k$ for the stored objects is found. If $v_i \le m$, then x_i is discarded and all stored vote tallies are reduced by v_i. If $v_i > u_k$, then all vote tallies are reduced by m and $(x_i, v_i - m)$ replaces one of the objects which is left with 0 vote tally. A second pass through the input, comparing each x_i to all remaining $z_j s$, and tallying the actual vote for each z_j, completes the algorithm.

3.2. *Theorem*

With an unordered input space, t-out-of-Σv_i voting can be performed with time complexity $O(np)$ and space complexity $O(p)$, where $p = \lfloor (\Sigma v_i)/t \rfloor$.

Proof. We assert that once all n input objects have been examined in the first "for" loop of Algorithm 3.1, any object with total vote tally of t or more will be among the final p objects stored in the p available slots. Since only objects whose vote tallies become zero through the vote reduction process are discarded, this assertion can be proven by showing that the total of the votes lost by any object is less than t. The proof is by contradiction. Suppose an object loses t or more votes in this process. Each time an object loses votes (due to the vote reduction process described above), p other distinct objects also lose the same vote. Thus if an object loses t votes in this process, a total of $(p+1)t$ votes must have been lost. But this is impossible since $(p+1)t$ is greater than Σv_i, the sum of all input votes. Each of the two "for" loops requires $O(np)$ time, thus proving the desired result. □

According to Theorem 3.2, when the threshold t is comparable in value to Σv_i (i.e., when p is a small constant), a small amount of working storage is required and the running time is linear in n, the number of inputs. Clearly, constant storage requirement and linear running time are the best that can be expected, since any algorithm must examine all n inputs.

3.3. *Corollary*

Weighted majority voting (t-out-of-Σv_i voting with $t > \frac{1}{2}\Sigma v_i$) can be performed in $O(n)$ time using working storage for a single object. □

3.4. *Corollary*

Unweighted m-out-of-n voting can be performed in $O(n^2/m)$ time. Unweighted majority, or $(\lfloor n/2 \rfloor + 1)$-out-of-$n$, voting can be performed in $O(n)$ time using working storage for a single object. □

3.5. *Example*

Consider 6-way, 8-out-of-15 voting with the vote weights 4, 3, 3, 2, 2, 1. Take an instance of the voting problem with input $(A, 3)$, $(B, 2)$, $(B, 2)$ $(A, 1)$, $(C, 3)$, $(A, 4)$ in presentation (input) order. A single working storage slot (z_1, u_1) is required that will successively hold the values $(A, 3)$, $(A, 1)$, $(B, 1)$, $(-, -)$, $(C, 3)$, and $(A, 1)$ as we proceed through the steps of Algorithm 3.1. Therefore, A is a candidate value for the voting result and a second pass through the input will yield its actual vote tally for comparison with 8. □

In practice, exact threshold voting has been done by simply using variants of Algorithm 2.2, followed by a comparison of the tallied votes with the threshold t. Since the average-case time complexity of such algorithms is linear and the number of input objects is usually small (< 10), researchers were perhaps not motivated to seek more efficient algorithms. With the use of higher degrees of replication along with more complex input objects, requiring both more storage space and more time to determine object equality, algorithm efficiency becomes important. Algorithm 3.1 is a generalized version of efficient non-weighted majority voting (a special case of the problem of finding repeated elements in a set) and m-out-of-n voting algorithms.[5–8]

4. Comparisons and Conclusions

The time complexity of n-way plurality and majority voting for a large unordered object space were shown to be $\theta(n^2)$ and $\theta(n)$, respectively. These are sequential complexities (i.e., for a single thread of computation). If parallel operations are allowed as in hardware-based voting networks,[9] then plurality voting can be performed in $O(n)$ steps using $\theta(n^2)$ network elements,[10] whereas majority voting requires $\theta(\log n)$ time and has $\theta(n^2)$ complexity since the results of n^2 comparisons can be combined using $(\log n)$-depth adder trees.

Our results do not suggest that threshold voting is "better" than plurality voting; only that it is "simpler". The choice of a voting scheme is primarily dependent on application context and the types of data elements or structures being voted on, with algorithm complexity being at best a secondary factor. However, once threshold voting has been selected as the voting scheme of interest, there is no excuse for tallying the votes as in Algorithm 2.2 in order to determine the voting result. The simpler Algorithm 3.1 should be used.

Analyses offered in this paper for the complexity of the various voting algorithms have been asymptotic and worst-case. Since in most practical cases the number of input data objects that participate in voting is small, more detailed analyses are needed for comparing and selecting algorithms. There are situations however when voting with a fairly large number of inputs is needed. One example is in image processing filters where each point may be replaced by voting on a collection of neighboring points.[11] Another example is in distributed fault diagnosis where voting might be used to determine the signature of a fault-free processor from the self-diagnosis signatures of participating processors.[12]

References

1. P. R. Lorczak, A. K. Caglayan and D. E. Eckhardt, "A theoretical investigation of generalized voters for redundant systems", *Proc. Int. Symp. Fault-Tolerant Computing* (Chicago, 1989), pp. 444–451.
2. B. Parhami, "Optimal algorithms for exact inexact, and approval voting", *Proc. Int. Symp. Fault-Tolerant Computing* (Boston, 1992), pp. 444–451.
3. B. Parhami, *IEEE Trans. Reliability*, to appear in 1994.
4. D. Dolev, L. Lamport, M. Pease, and R. Shostak, "The Byzantine generals", in *Concurrency Control and Reliability in Distributed Systems*, ed. B. K. Bhargava, Van Nostrand Reinhold (New York, 1987), pp. 348–369.
5. J. Misra and D. Gries, *Science of Computer Programming* **2**, 143 (1982). See also correspondence in *The Computer Journal* **35(3)**, 298 (1992).
6. D. Gries, "A hands-in-the-pocket presentation of a k-majority vote algorithm", in *Formal Development of Programs and Proofs*, ed. E. W. Dijkstra (Addison-Wesley, 1990), pp. 43–45.
7. R. S. Boyer and J. S. Moore, "MJRTY — A fast majority vote algorithm", in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, ed. R. S. Boyer (Kluwer, 1991).
8. D. Campbell and T. McNeill, *The Computer Journal* **34(2)**, 186 (1991).
9. B. Parhami, *IEEE Trans. Reliability* **40**, 380 (1991).

10. B. Parhami, "The parallel complexity and weighted voting", *Proc. of the Int. Symp. on Parallel and Distributed Computing and Systems* (Washington, DC, 1991), pp. 382–385.
11. D. R. K. Brownrigg, *Commun. ACM* **27(8)**, 807 (1984).
12. S. Y. H. Su, M. Cutler and M. Wang, *IEEE Trans. Computers* **40(11)**, 1252 (1991).