Voting Algorithms

Behrooz Parhami, Senior Member IEEE
University of California, Santa Barbara

Key Words — **Approval voting, Computational complexity, Design diversity, Inexact voting, Majority voting, Multi-channel computation, Multi-version software, *N*-modular redundancy, Plurality voting, Threshold voting.**

Reader Aids —
General purpose: Compare voting algorithms and present new ones
Special math needed for derivations: Simple combinatorics
Special math needed to use results: None
Results useful to: Designers of fault-tolerant and distributed computing systems

Summary & Conclusions — **Voting is important in the realization of ultrareliable systems based on the multi-channel computation paradigm. In an earlier paper (1991 Aug) I dealt with voting networks, *viz*, hardware implementation of certain voting schemes. A voting algorithm specifies how the voting result is obtained from the input data and can be the basis for implementing a hardware voting network or a software voting routine. This paper presents efficient *n*-way plurality and threshold voting algorithms based on the type of voting (exact, inexact, or approval), rule for output selection (plurality or threshold) and properties of the input object space (size & structure). Exact voting is the most common voting method and is the easiest to implement. Inexact voting algorithms are more complicated due to intransitivity of approximate equality. As an example, when approximate equality $a \cong b$ for numerical inputs a & b is defined as $|a - b| \leq \epsilon$, then $a \cong b$ and $b \cong c$ do not imply $a \cong c$. In approval voting, each input to the voting process consists of a finite or infinite set of values that have been "approved" by the corresponding computation channel and the value, or set of values, with the highest approval voting must emerge as output. Multiple approved values can result from non-unique answer to a given problem or from uncertainties in the solution process. For exact voting, the complexity of an *n*-way voting algorithm depends on the structure of the input object space. Threshold voting often requires less time & space, except when the threshold is very small. I extend the techniques for designing efficient exact voting algorithms to inexact and approval voting schemes. My results show that optimal linear-time (in *n*) voting algorithms are available when the input object space is small. Next in the time-complexity hierarchy is the case of a totally-ordered object space that supports worst-case order-$[n \cdot \log(n)]$-time algorithms for both exact and inexact voting as well as for certain approval-voting schemes. These algorithms are intimately related to sorting and have the same time complexity. An unordered input object space leads to worst-case quadratic-time exact and inexact voting algorithms, even when a distance metric can be defined on pairs of input objects.**

1. INTRODUCTION

Voting is important for 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. Depending on the data volume and the frequency of voting, hardware or software voting schemes can be appropriate. Low-level voting with high frequency requires hardware voters whereas high-level voting on the results of fairly complex computations can be performed in software without serious performance degradation or overhead.

The use of voting for obtaining highly reliable data from multiple unreliable versions was first suggested in the mid 1950s [53]. Since then, the concept has been practically used in fault-tolerant computer systems and has been extended & refined in many ways, *eg*,

- reliability modeling of voting schemes by considering compensating errors [46],
- handling of imprecise or approximate data [13],
- combination with standby or active redundancy [30],
- reconfigurable voting with declining replication factor upon detected failures [31],
- voting on digital "signatures" obtained from computation states to reduce the amount of information to be voted on [49],
- dynamic modification of vote weights based on a priori reliability data [42].

More recently, generalized voting with unequal vote weights has been suggested for maintaining the reliability & consistency of data stored with replication in distributed computer systems [19]. This has become a very active research area.

Hardware voters described in the literature are essentially "bit-voters" that compute a majority function on n input bits [23, 48]. Combined bit voting and disagreement detection has been discussed [14]. Hardware voting on words and higher-level data objects has traditionally been handled by using s-independent parallel bit-voters or feeding the data sequentially through a single unit. Such s-independent bit-voting yields results that are optimistic, particularly when s-correlated errors are likely. Several algorithms and design techniques for hardware voters with adjustable or variable vote weights have been published [35, 37, 38], although none has been implemented. These hardware voting schemes all deal with voting on exact values (typically bit strings representing logical decisions or integer numerical values). Approximate or other context-dependent voting algorithms have not been implemented in hardware.

Hardware voters used in actual systems include 3-way voters in the:

- MIT FTMP design [21],
- Carnegie-Mellon University C.vmp system [47],
- August Systems industrial control computers [56].

The Jet Propulsion Laboratory's STAR computer [3] used a special 2-out-of-5 voter that was symmetrically connected to

0018-9529/94/$4.00

3 active modules and 2 standby spares for its critical Test & Repair Processor (aerospace computers used hardware voters even before STAR). The effect of switch complexity on reliability and the design of low-complexity, and thus highly reliable, switch-voters for hybrid redundancy were considered by Siewiorek & McCluskey [44, 45]. This work was instrumental in attracting attention to the importance of simple switch-voters and led to several other designs including the self-purging [29] and sift-out [16] variations of adaptive voting with vote weights in $\{0,1\}$.

Proposed software voters are quite varied and possess a wide range of features. The earliest software voters were used in the design of modular multiprocessors with replicated software. For example, the voter routine in the SRI International SIFT design [55] is invoked by any task which requires inputs for a new iteration. It uses tables provided by the local reconfiguration task to determine which processors contain copies of the required output (and in which of their buffers), reads the data from the appropriate buffers and uses a majority rule to obtain a single value. In the Space Shuttle 4-way software voting scheme [49], selected data items are computationally combined to form "compare words" that are periodically exchanged & compared in 4 out of the 5 on-board computers. Another example is the "Stepwise Negotiating Voting" scheme of [24] which amounts to a 2-out-of-n threshold voting strategy. The advantages of such "relaxed" (non-majority) voting schemes have been discussed by others as well [1, 41].

Researchers in software diversity have designed voters suited for processing the results obtained by multiple versions of a program and have contributed techniques for handling approximate results [4, 10, 25, 52]. Similar considerations apply to voting schemes with data diversity [2]. Software voters have also been designed in connection with the management of replicated data in distributed systems [5, 6, 18, 19, 22, 51] to assure database reliability and/or consistency. A common way to achieve this goal is to assign votes to participating nodes in the distributed system and to implement mutual exclusion by requiring each operation agent to "collect" a certain number of consenting votes. Typical issues in this area are vote assignments to maximize reliability or to minimize average transaction response time (*eg*, by adjusting "read" and "write" quorums). The delays resulting from synchronized voting in real-time systems and the expected time to collect a prescribed number of votes from distributed sites with different response characteristics have been studied in [43, 34], respectively.

Research on voting algorithms has dealt with both implementation & effectiveness of voting schemes. In algorithm implementation, the main focus has been simple majority voting, a special case of finding repeated elements in a set [9, 12, 20, 33]. In another line of attack [28], the notion of "approximate voting", henceforth applied only to real-valued numerical results, was generalized by demonstrating that forms of inexact voting can be easily applied to any metric space or, for weighted averaging, to any real vector space. As for effectiveness, [28] qualitatively compares several voting schemes under various error conditions. The problem of selecting the best voting scheme to maximize the probability of obtaining a correct result has been investigated in [7, 32], with some refinements in [8]. Voting algorithms have also been studied from the point of view of complexity [36].

This paper deals with voting algorithms. In a companion paper [35], I dealt with voting networks, *viz*, hardware implementation of certain voting schemes. A voting algorithm specifies how the voting result is obtained from the input data and can be the basis for a voting network (hardware) or a voting routine (software). However, since voting networks are based on very simple algorithms and deal with equally simple input data objects (usually logical bit strings or numeric words) in the interest of practical realizability, the bulk of this paper relates only to software voting.

Section 2 introduces some concepts & definitions. Section 3 deals with exact voting, covering theoretical results and actual algorithms for various types of input object spaces. Section 4 includes similar results for inexact voting; *ie*, when "equality" of input objects is approximate and, hence, intransitive. Section 5, a) introduces the notion of approval voting, b) provides examples of where it might be useful, and c) presents voting algorithms for several special cases, including those for ordered input object spaces where approved sets of input values are represented by lists (multisets) or intervals. Section 6 discusses results and directions for further research. All proofs are in the appendix.

Notation

δ	size of the input object space (number of different values x_i can take)
$d(x_i, x_j)$	distance between x_i & x_j in inexact voting
i	identification for an input data object, $i = 1,2,\ldots,n$
n	number of inputs to the voting algorithm
t	minimum total votes needed in threshold voting
u_j	vote tally associated with input class j
v_i	input vote associated with x_i
V	$\sum_{i=1}^{n} v_i$
w	output vote associated with y
x_i	input data object i
y	output data object produced by the voting algorithm.

Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

2. PRELIMINARIES

Regardless of the type of input objects and the type of implementation (hardware or software) that is most appropriate, I view a voting algorithm as dealing with n input data objects x_i having associated votes/weights v_i (n input data-vote pairs $\langle x_i, v_i \rangle$) and producing the output data-vote pair $\langle y, w \rangle$. The voting algorithm can also produce a set of n "support bits" s_i, one for each input, that indicate whether a given input "supports" or "agrees with" y (the notion of "support" is defined later). The input votes and output vote need not be explicitly represented or even present at all, as shown shortly.

I start with weighted voting for 3 reasons:

1. It is more general than simple voting and thus useful in a wider context. Setting all the weights equal to 1 yields simple non-weighted voting as a special case.

2. In most cases, weighted voting is not harder to implement than simple voting, especially in software-based implementations (adding an arbitrary weight to a vote tally isn't any harder than adding 1). When non-weighted voting is considerably simpler than weighted voting, this is pointed out.

3. Some fixed-weight schemes are essentially adaptive over the long run. For example, disabling faulty units or reintroducing repaired units in some n-way voting schemes can be equivalent to changing their votes from 1 to 0 or 0 to 1, respectively.

The four main components of a voting algorithm (input data, output data, input votes, output vote) can be used to impose a binary 4-cube classification scheme (see figure 1), leading to 16 classes.

Nomenclature (Classification Coordinates)

- Exact/Inexact. The exact/inexact dichotomy concerns whether input objects are viewed as having inflexible values or as representing flexible "neighborhoods".
- Consensus/Compromise. This paper deals primarily with consensus voting. Several important "compromise" voting schemes, such as median and mean voting, are not emphasized because they are specific to certain input types and do not apply to a general object space. I deal with such voting schemes only where applicable.
- Preset/Adaptive. The preset/adaptive dichotomy corresponds to v_i being set at design time or allowed to change. Adaptive voting schemes can have adjustable votes (stored in writable memory) or variable votes (presented as inputs).
- Threshold/Plurality. Threshold voting requires that w exceed a certain preset threshold whereas plurality voting identifies an output y with maximum support from the inputs. Both threshold & plurality voting are covered. ◄

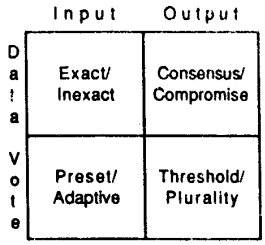

Figure 1. Binary 4-Cube Classification for Voting Algorithms [based on variations in input/output data and input/output votes]

Definition 2.1 encompasses virtually all consensus voting schemes of practical interest (exact/inexact, preset/adaptive, threshold/plurality).

Definition 2.1 — Weighted Consensus Voting

Given n input data objects x_i, with n associated non-negative real votes (weights) v_i, compute the output y and its vote w such that y is "supported by" several input data objects with votes totalling w, where w satisfies a condition associated with the desired threshold or plurality voting subscheme.

A. Threshold voting subschemes

Unanimity voting: $w = V$
Byzantine voting: $w > \frac{2}{3}V$
Majority voting: $w > \frac{1}{2}V$
m-out-of-n voting ($v_i \equiv 1$): $w \geq m$; if $m \leq \frac{1}{2}n$, then y can be non-unique
t-out-of-V (generalized m-out-of-n) voting: $w \geq t$; if $t \leq \frac{1}{2}V$, then y can be non-unique

B. Plurality voting subscheme

No other y' is supported by inputs having more votes; if $w \leq \frac{1}{2}V$, then y can be non-unique.

Nomenclature

- *Supported by*. This term can be defined in several ways, leading to different voting schemes, each of which has the subschemes A & B.

  i. Exact voting: an input object x_i supports y iff $x_i = y$.
  ii. Inexact voting: approximate inequality ($\cong$) is defined in some suitable way (*eg*, by providing a comparison threshold ϵ for numerical values or, more generally, a distance measure d in a metric space) and x_i supports y iff $x_i \cong y$.
  iii. Approval voting: y must be a member of the approved set of values that x_i defines. (More on this in section 5). ◄

The Byzantine voting scheme in definition 2.1 is a generalized form of the unweighted version used in distributed computing [17] where n autonomous n-way voting nodes/sites must arrive at consistent conclusions in the presence of fewer than $\frac{1}{3}n$ faulty nodes; Byzantine faulty nodes might try to confuse other nodes by presenting inconsistent values to them. In this generalized form, less than $\frac{1}{3}$ of votes/weights can be associated with faulty nodes.

Figure 2 shows a classification of input object spaces associated with voting algorithms. The input objects to be voted upon can be atomic or composite. Composite objects, consisting of structured collections of atomic objects, have not received due attention in previous works on voting. The only examples of composite inputs dealt with in this paper are sets that are represented as lists or intervals; other structures can be envisaged as well. Assuming atomic objects, the input object space can be small or large.

- Small object spaces: Further classification is unimportant, as they support very efficient voting algorithms.

- Large metric object spaces: The ability to define a distance metric, and as an important special case, if the objects can be ordered, improves the voting efficiency (definition 4.1.1 of *distance* is slightly more general than that of metric spaces).
- Large unordered object spaces: Voting algorithms tend to be less complex if the notion of *support* (see definition 2.1) is transitive[1].

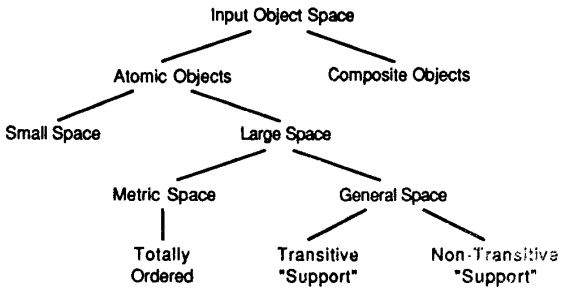

Figure 2.Classification of Voting Schemes
[based on the size & structure of the input object space]

Sections 3 - 5 discuss voting algorithms (based on definition 2.1) in relation to the classification of input object spaces shown in figure 2. We need the following notation in discussing asymptotic-time and space-complexities of voting algorithms.

Definition 2.2 — Complexity bounds

Asymptotic behavior of functions (*eg*, time & space bounds) are specified/compared as follows:

$f(n) = O(g(n))$ iff there exist constants c_1 & n_1 such that

$f(n) \leq c_1 \cdot g(n)$ for all $n \geq n_1$.

$f(n) = \Omega(g(n))$ iff there exist constants c_2 & n_2 such that

$f(n) \geq c_2 \cdot g(n)$ for all $n \geq n_2$.

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$;

ie, $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ for large n.

In other words, $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$ convey upper, lower, exact bounds, respectively, for the growth rate of $f(n)$. ◄

3. ALGORITHMS FOR EXACT VOTING

Although exact voting has been studied & used extensively, no general discussion of exact-voting algorithms appears in the literature. The reason is that exact non-weighted n-way voting for bits or numerical data with $n=3$ or $n=5$ is quite simple. But for larger n or for composite data types, the situation changes. The discussion of algorithms is divided according to the size & structure of the input object space.

Assumption (section 3 only)

1. *Equality* is transitive since we are dealing with *exact* comparisons.

3.1 Small Object Space

For a small object space, an efficient linear-time voting algorithm can be devised.

Let the set of possible values or classes of objects be encoded by integers $\{1, 2, 3, \ldots, \delta\}$. The algorithm consists of tallying the votes for each of the δ possible values/classes and then selecting the appropriate output.

Algorithm 3.1.1: Exact Voting — Small Object Space

Notation

u_i vote-holder for object class i, $i=1,2,\ldots,\delta$.

for $i = 1$ to δ do $u_i := 0$ endfor
for $i = 1$ to n do $j := \text{class}(x_i)$;
  $u_j := u_j + v_i$ endfor
$(y, w) := \text{select}(u)$ ◄

For all varieties of voting covered by definition 2.1, the selection function for output (last step) can be computed in time $O(\delta)$. Therefore, the execution time of algorithm 3.1.1 is $O(n+\delta)$.

Algorithm 3.1.1 can be easily implemented in hardware. Direct implementation for bit-voting ($\delta=2$) is depicted in figure 3. With bit voting, both hardware & software implementations can be simplified by tallying the vote for only 1 of the 2 values. With this simplification, the design of figure 3 becomes the *arithmetic-based* design technique for bit-voting networks [35]. Despite its seemingly low $O(n \cdot \delta)$ complexity and $O(\log(n))$ delay, hardware implementation of this algorithm is practical only for very small δ (perhaps only for bit-voting). The software version, however, remains attractive for larger δ, as long as the $O(\delta)$ working space is acceptable. The obvious $\Omega(n)$ time lower bound for n-way voting leads to theorem 3.1.2.

Theorem 3.1.2. Algorithm 3.1.1 is an optimal n-way voting scheme for $\delta = O(n)$. ◄

3.2 Totally Ordered Object Space

With a large object space (*eg*, 32-bit integers), algorithm 3.1.1 is inapplicable. This section shows that if the input object space is totally ordered, then algorithm 3.2.1 (based on sorting) is optimal.

[1](X supports Y) AND (Y supports Z) implies (X supports Z).

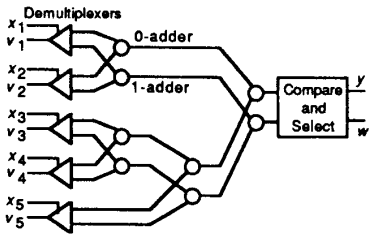

Figure 3. Hardware Realization of 5-Way Weighted Bit-Voting [The triangular demultiplexers use the x_i data bits as *control* or *selection* signals and have widths equal to the number of bits in the binary representations of the v_i. Each circular adder attaches its carry-out signal to its output, which is then fed to a wider higher-level adder. If each of the n input votes can be as large as v_{max}, then w has gilb($\log_2(n \cdot v_{max})$) + 1 bits.]

Algorithm 3.2.1: Exact Plurality Voting — Totally Ordered Object Space

Sort (ascending order) in place the set of records (x_i, v_i) with x_i as key. Use the end-marker $(x_{n+1}, v_{n+1}) = (\infty, 0)$.

$y:=z:=x_1$; $u:=w:=v_1$

for $i = 2$ to $n+1$ do

  while $x_i = z$ do $u:=u + v_i$; $i:=i + 1$ endwhile

  if $u > w$ then $w:=u$; $y:=z$ endif

  $z:=x_i$; $u:=v_i$

endfor {next i} ◄

This algorithm can be easily modified for other voting schemes. For example, majority voting requires appending, at the end, the statement:

if $w \leq \frac{1}{2}V$ then $w:=0$ (no-quorum indicator).

Weighted median voting is performed as follows.

Algorithm 3.2.2: Weighted Median Voting — Totally Ordered Object Space

Sort (ascending order) in place the set of records (x_i, v_i) with x_i as key.

$u:=v_1$; $i:=1$

while $u < \frac{1}{2}V$ do $i:=i + 1$; $u:=u + v_i$ endwhile

$y:=x_i$ ◄

Since sorting is asymptotically an $\Omega(n \cdot \log(n))$ operation (it takes quadratic time for small values of n that are of practical interest in voting), with highly reliable computation channels it is quite advantageous to add the following test to the beginning of all algorithms in section 3:

if $x_1 = x_2 = \ldots = x_n$ then $(y, w) = (x_1, V)$ stop endif

For example, if each computation channel produces a correct result with probability 0.999 and we use 5-way nonweighted voting to obtain a result with even higher reliability, then the effect of this additional voting test is to make: a) the running time of the algorithm linear in at least 99.5 percent of the cases, and b) the average running time almost linear. It might 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 appreciably.

Hardware implementation of algorithm 3.2.1 or its variants can also be contemplated. The straightforward hardware design consists of a sorting network followed by a combining and a max-selection network. The cells in a sorting network can be modified to convert it into a vote-tallier [35] that requires very little additional hardware to become a voting network.

Theorem 3.2.3 shows that algorithm 3.2.1 is optimal in the sense that voting with a large object space has $\Omega(n \cdot \log(n))$ complexity.

Theorem 3.2.3. The complexity of n-way plurality voting as specified in definition 2.1 with large object space is $\Omega(n \cdot \log(n))$. ◄

3.3 Unordered Object Space

Algorithm 3.3.1 for plurality voting runs in $O(n^2)$ time.

Algorithm 3.3.1: Exact Plurality Voting — Unordered Object Space

Notation

z_j distinct input j encountered
u_j vote tally for z_j.

$k:=1$; $y:=z_1:=x_1$; $w:=u_1:=v_1$

for $i = 2$ to n do

  if there 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 this algorithm results from the $(n-1)$-iteration loop and the $O(n)$ linear search required in each iteration. The average performance is again almost linear because k remains 1 with very high probability. Because of excellent average-case performance, there is no need to resort to more efficient search schemes (such as those based on hashing) to find j inside the *for* loop.

Intuitively, in an unordered space, the equality of two objects can be established only 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 in the worst case, a) $\frac{1}{2}n \cdot (n-1)$ comparisons are needed, and b) the last comparison might provide the only non-singleton equivalence class of size 2. Thus, working storage for $n-1$ objects and their associated vote tallies is needed. Theorem 3.3.2 formalizes these intuitive notions.

Theorem 3.3.2. With an unordered input object space, n-way plurality voting has time complexity $\Omega(n^2)$. ◄

One can devise simpler threshold voting algorithms in this case. Consensus voting clearly has linear complexity. Less obvious is theorem 3.3.3.

Theorem 3.3.3. Let $p \equiv \text{gilb}(V/t)$. With an unordered input space, t-out-of-V voting can be performed with time complexity $O(n \cdot p)$ and space complexity $O(p)$. ◄

Theorem 3.3.3 is a generalization of a published result for m-out-of-n voting [12], although my proof appears to be simpler as well as more general.

Corollary 3.3.4. Weighted majority voting (t-out-of-V, with $t > \frac{1}{2}V$) can be performed in $O(n)$ time using working storage for a single object. ◄

Corollary 3.3.5. Unweighted m-out-of-n voting can be performed in $O(n^2/m)$ time. Unweighted majority, (gilb($\frac{1}{2}n$)+1)-out-of-n, voting can be performed in $O(n)$ time using working storage for a single object. ◄

Example 3.3.6

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 inputs (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 $(A, 3)$, $(A, 1)$, $(B, 1)$, (-,-), $(C, 3)$, $(A, 1)$ as we proceed through the steps of the algorithm in the proof of theorem 3.3.3. Therefore, A is a candidate value for the voting result and a second pass through the input yields its actual vote tally of 8 for comparison with the threshold of 8. ◄

4. ALGORITHMS FOR INEXACT VOTING

Algorithms for inexact voting are generally more complex than their exact-voting counterparts, primarily due to the non-transitivity of approximate equality. Although the notion of approximate or inexact voting and its applications have been discussed in the literature, only one published inexact voting algorithm is known to me [28] and that algorithm happens to be incorrect (more on this in section 4.1). Inexact voting with a small object space, though theoretically possible, is impractical.

Assumption (section 4 only)

2. The object space is large.

4.1 General Weighted Inexact Voting

Notation

X	input object space
R	set of real numbers
ϵ	a suitably small comparison threshold.

Dealing with approximate equality requires defining a real-valued distance function $d: X^2 \rightarrow R$ on pairs of objects in the input object space. Then two objects, x_i & x_j, are approximately equal if $d(x_i, x_j) \leq \epsilon$. Definition 4.1.1 captures some properties of d consistent with the intuitive notion of approximate equality.

Definition 4.1.1.

The function $d: X^2 \rightarrow R$ is a distance function for the object space X iff for all $x_i, x_j \in X$:

1. $d(x_i, x_j) \geq 0$

2. $d(x_i, x_j) = 0$ iff $x_i = x_j$

3. $d(x_i, x_j) = d(x_j, x_i)$. ◄

Excluded from definition 4.1.1 is the *triangle inequality*, $d(x_i, x_k) \leq d(x_i, x_j) + d(x_j, x_k)$, which would make (X, d) a metric space. This is done because the *triangle* restriction would not simplify the voting algorithm while it needlessly eliminates some potentially useful definitions of distance.

Example 4.1.2

Let our objects be pairs of integers (i, j) denoting points on a 2-dimensional grid. Let,

$$d((i_1, j_1), (i_2, j_2)) \equiv \min(|i_1 - i_2|, |j_1 - j_2|) \text{ with } \epsilon = 0.$$

Accordingly, two objects are considered *approximately equal* iff they have matching i or j coordinates. Then, sets of *approximately equal* objects contain points that are horizontally or vertically aligned. ◄

While distance functions such as the one in example 4.1.2 might not seem particularly useful for common applications, there is no compelling reason to exclude them (by restricting the discussion to metric spaces) when the exclusion does not lead to simpler algorithms.

Given input objects x_i with associated votes v_i, $i=1,2,\ldots,n$ and the distance function $d: X^2 \rightarrow R$ satisfying definition 4.1.1, algorithm 4.1.3 can be used for inexact voting.

Algorithm 4.1.3: Weighted Inexact Voting

1. Determine a maximal-vote subset S of the set of n input objects such that for all $x_i, x_j \in S$, $d(x_i, x_j) \leq \epsilon$ (this subset can be non-unique).

2. In general, S can contain identical elements; *viz*, objects x_i & x_j with $d(x_i, x_j) = 0$. Combine the votes for identical objects in S, getting the set $S' \equiv \{z_1, z_2, \ldots, z_m\}$ of distinct objects and associated vote tallies $u_1, u_2, \ldots, u_m$, with $m \leq n$ and $d(z_i, z_j) > 0$.

3. y:=select(z, u); w:= $\sum_{i=1}^{m} u_i$. ◄

Algorithm 4.1.3 is at a high level, and each step must be further clarified & analyzed with respect to complexity. Step 1 requires $O(n^2)$ time and can be carried out using well-known procedures for minimizing the number of states in an incompletely specified sequential machine [27].

Brief Overview of a Procedure for Step 1

An $(n-1)\times(n-1)$ triangular table is constructed in which entry (i, j) indicates the *compatibility* (approximate equality) of x_i & x_j. Compatibility classes of size 2 are read directly from the table. Larger compatibility classes are built in a stepwise fashion by adding, to an existing class, a new object which is compatible with every member of that class. As illustrated by the example of figure 4, the maximal compatibility classes thus obtained form a *cover* on the set of objects (and not a partition as asserted in [28: section 2.1]).

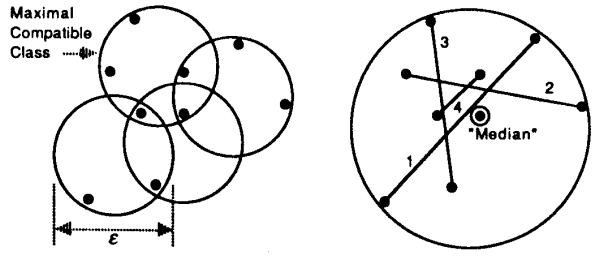

Figure 4. Maximal Compatible Subsets in Inexact Voting and the Generalized *Median* Selection Rule

It is also possible to define the computation in step 1 (of algorithm 4.1.3) recursively.

Notation

S	cluster($\{x_1, x_2, \ldots, x_n\}$): the computed function in step 1
M	$\{z_1, z_2, \ldots, z_k\}$: any set of objects.

Given M, let $z_i, z_j \in M$ be a pair of objects that are furthest apart. Then,

$$\text{cluster}(M) \equiv \begin{cases} M, \text{ if } d(z_i, z_j) \leq \epsilon \\ \max[\text{cluster}(M - \{z_i\}), \text{cluster}(M - \{z_j\})], \text{ otherwise.} \end{cases}$$

The resulting implementation is fairly efficient because in practice recursion stops after 0 or 1 level with very high probability.

Step 2 of algorithm 4.1.3 is simple and can be merged with step 1 (reducing the compatibility table by combining objects pairs having 0 distance).

The selection function in step 3 can be specified in different ways depending on the structure of the input object space. A general selection rule would be to a pick an object z_j having maximal vote (plurality voting). For a metric space, generalized median[2] voting can be used as suggested in [28]. Figure 4 shows an example of finding a generalized *median* within a maximal compatible class. For numerical values, the mean selection rule can also be applied. In all the above cases (plurality, generalized median, mean), the worst-case complexity of step 3 is $O(n)$, resulting in overall $O(n^2)$ complexity for algorithm 4.1.3. Whether one of the objects z_i is picked as the output, or a *compromise* object is constructed based on the set S', the output vote w is the sum of the votes for the entire subset S.

There are also situations in which the complexity of the final selection dominates that of the rest of the algorithm. A good example is voting on strings with d defined as the *edit distance*[3] between two strings, and the voting result defined as a string for which the weighted sum of distances from $z_1, z_2, \ldots, z_m$ is minimal. Since in the worst case $m = n$, any selection rule that would require a search in the large input space for a value that optimizes an objective function would dominate the algorithm's time complexity. In such cases, the time complexity of the algorithm is $\Omega(n^2)$.

Example 4.1.4

Consider the following object-vote pairs ($x_i \in R$, $v_i \in N$) as inputs to an inexact 8-out-of-13 voting algorithm with the comparison threshold of 0.02; $d(x_i, x_j) = |x_i - x_j|$.

i	Objects (x_i)	Votes (v_i)
1	1.300	2
2	1.310	3
3	1.330	4
4	1.340	3
5	1.350	1

[2]Recursively removing an object pair with the largest distance until only a single object or two objects are left, then picking one at will.
[3]The minimal number of symbol insertions, deletions, or substitutions that would convert one string into the other.

The compatible pairs are: (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_3, x_5), (x_4, x_5). The maximal-compatible cover is thus $(x_1, x_2)(x_2, x_3)(x_3, x_4, x_5)$ with class vote tallies being 5, 7, 8, respectively. Thus $w = 8$. As for y, the maximal-vote selection rule yields 1.330 while the weighted median and weighted mean rules result in 1.340 and 1.336, respectively. ◄

Clearly, distances between all object pairs must be obtained in any algorithm for inexact voting. This leads to theorem 4.1.5 that establishes the optimality of algorithm 4.1.3.

Theorem 4.1.5. The time complexity of weighted inexact voting with a general object space is $\Omega(n^2)$. ◄

4.2 Totally Ordered Object Space

In the special case of a totally ordered object space, a simpler inexact voting algorithm can be devised based on sorting. The notion of a totally ordered object space for inexact voting is formalized in definition 4.2.1.

Definition 4.2.1. An object space X is totally ordered with respect to the distance metric d if for any 3 distinct objects x_i, x_j, $x_k \in X$, $d(x_i, x_k) = |d(x_i, x_j) \pm d(x_j, x_k)|$. ◄

Definition 4.2.1 establishes a total order as follows. Pick 2 distinct elements x_i, x_k and order them arbitrarily; say x_i precedes x_k, denoted by $x_i \rightarrow x_k$. Given an element x_j, it can be ordered relative to x_i & x_k by the rules:

$$x_j \rightarrow x_i \rightarrow x_k \text{ iff } d(x_i, x_k) = -[d(x_i, x_j) - d(x_j, x_k)]$$

$$x_i \rightarrow x_j \rightarrow x_k \text{ iff } d(x_i, x_k) = d(x_i, x_j) + d(x_j, x_k)$$

$$x_i \rightarrow x_k \rightarrow x_j \text{ iff } d(x_i, x_k) = d(x_i, x_j) - d(x_j, x_k).$$

Continued application of these rules will order all elements in X. Assuming that the relative order of any two objects can be determined by simply comparing the object pair, then any fast sorting algorithm can be used to obtain a worst-case $O(n \cdot \log(n))$-time inexact voting algorithm.

Algorithm 4.2.2: Inexact Plurality Voting — Totally Ordered Object Space

Sort in place the set of records (x_i, v_i) with x_i as key; use special end-marker $(\infty, 0)$

$i := j := 1$; $u := w := 0$

while $j \leq n$ do

  while $d(x_i, x_j) \leq \epsilon$ do $u := u + v_j$; $j := j + 1$ endwhile

  if $u > w$ then $w := u$; first$:= i$; last$:= j - 1$ endif

  $u := u - v_i$; $i := i + 1$

endwhile

$y :=$ select(first, last) /* the vote w has already been computed */ ◄

The selection function here takes the indices of 2 elements in the sorted list and returns a value selected from x_{first} to x_{last} or computed from all elements in that interval. The complexity of algorithm 4.2.2 is dominated by the $O(n \cdot \log(n))$-time sorting phase, as the rest runs in linear time. Since inexact voting is at least as hard as exact voting, corollary 4.2.3 to theorem 3.2.3 establishes the optimality of algorithm 4.2.2.

Corollary 4.2.3. The time complexity of n-way inexact plurality voting as specified in definition 2.1 is $\Omega(n \cdot \log(n))$. ◄

Again, as was the case for exact voting, a simple initial test, establishing if the distance between the end (minimal and maximal) objects is no more than ϵ, can produce almost linear average-case running time for the algorithm.

Here, also, a recursive formulation is possible. Instead of sorting, we recursively remove from the set under consideration, the smallest or the largest element. Thus we want to compute:

$$S = \text{cluster}(\{x_1, x_2, \ldots, x_n\}),$$

$$\text{cluster}(M) \equiv$$

$$\begin{cases} M, \text{ if } \max(M) - \min(M) \leq \epsilon \\ \text{larger of}[M - \{\max(M)\}, M - \{\min(M)\}], \text{ otherwise.} \end{cases}$$

The advantage of this method is that it achieves average-case linear running time in most practical situations.

Example 4.2.4

Consider the inputs of example 4.1.4, which are already in sorted order, the same distance function $d(x_i, x_j) = |x_i - x_j|$, and the same comparison threshold of $\epsilon = 0.02$. In performing inexact voting by means of algorithm 4.2.2, the variables involved assume the following values after each iteration of the outer while-loop:

Iteration	i	j	u	w	first	last
0	1	1	0	0	—	—
1	2	3	3	5	1	2
2	3	4	4	7	2	3
3	4	6	4	8	3	5

Therefore, the output y must be computed based on the set $\{x_3, x_4, x_5\}$ of inputs; the output vote is $w = 8$.

5. ALGORITHMS FOR APPROVAL VOTING

In ordinary socio-political context, approval voting is an election process whereby each participant votes for a subset of candidates who meet that participant's criteria for the position rather than picking just the one "best" person. The candidate with the highest approval vote tally wins the election. Some

restrictions might apply to the subset that each voter can approve (*eg*, there is a maximum size). Such details, as well as certain disadvantages of approval voting, are not relevant to this discussion. An important advantage of approval voting (and relevant to dependable computing) is that a lesser qualified candidate does not get the highest vote because of vote-splitting among several better, but almost equally qualified, candidates. In this paper, approval voting means that each input to the voting consists of a set (finite or infinite) of approved values. Multiple approved values can result from: a) non-unique answer to a given problem, or b) uncertainties in the solution process. Either way, the set of values with the highest approval vote emerges as output.

Example of Useful Approval Voting

A process-control requires the periodic determination of a safe setting for a particular system variable. In general, there might be more than one safe setting or a range of safe values. If multiple redundant versions of the control program present their sets of "approved" values in a suitable format, an approval-voting routine can pick the required setting based on these values. ◄

5.1 Small Object Space

Let the input object space be of size δ; let the set of possible values or classes of objects be encoded by integers $\{1, 2, \ldots, \delta\}$. Since δ is small, the approval-set i can be represented by the bit vector x_i with components $x_{i,j}$ ($x_{i,j} = 1$ means that input i approves the value/class j). Then the algorithm consists of tallying the votes for each of the δ possible values/classes and then selecting the appropriate output.

Algorithm 5.1.1: Approval Voting — Small Object Space

u_i $(1 \leq i \leq \delta)$ holds the vote tally for object class i

for $i = 1$ *to* δ do $u_i := 0$ endfor

for $i = 1$ to n do

for $j = 1$ to δ do if $x_{i,j} = 1$ then $u_j := u_j + v_i$ endfor

endfor

$(y, w) := \text{select}(u)$ ◄

The pair of nested loops in algorithm 5.1.1 involves n & δ iterations, respectively. For all varieties of voting covered by definition 2.1, the selection function for output (last step) can be computed in time $O(\delta)$. Therefore, the execution time of algorithm 5.1.1 is $O(n \cdot \delta)$.

A slightly modified version of the circuit in figure 3 can be used for hardware implementation of this algorithm if: a) the bits $x_{i,j}$ are applied serially to the x_i inputs, and b) pipelining is used. It is fairly easy to show the optimality of this algorithm via:

Theorem 5.1.2. Algorithm 5.1.1 is an optimal n-way approval voting scheme. ◄

5.2 Totally Ordered Object Space

Just as for exact voting, a large object space renders the approach in section 5.1 impractical. The set of approved values associated with each input can be represented in various ways. When all approval sets are relatively small, they can be represented by lists. In this case, let the approval lists be sorted in ascending order and terminate each list by the special marker "∞". This marker makes each set nonempty, facilitates the termination check, and obviates the need for special handling of empty sets at input & output. For large approval sets, This paper considers only *interval representation*; *ie*, all values from some lower bound l_i to an upper bound h_i are approved; denote this approval interval by $[l_i, h_i]$.

Algorithm 5.2.1: Approval Voting — List Representation

Initialize the working list N of value-vote pairs to $\emptyset$;

$S := \{1, 2, \ldots, n\}$; $z := -\infty$

while $z \neq \infty$ do

Read the next element of L_i into z_i for all $i \in S$;

$z := \min(z_1, z_2, \ldots, z_n)$.

$S := \{j | z_j = z\}$; $u = \sum_{i \in S} v_i$.

Append the value-vote pair (z, u) to the working list N.

endwhile

Compute the output list $M := \text{select}(N)$. ◄

Example 5.2.2

Consider the approval lists:

$L_1 = (1, 2, 3, 4, \infty)$,

$L_2 = (2, 3, 4, 5, \infty)$,

$L_3 = (3, 4, 5, \infty)$.

Just before executing the final selection step of algorithm 5.2.1, N is:

$((1, 1), (2, 2), (3, 3), (4, 3), (5, 2), (\infty, 3))$.

Depending on the selection rule applied, the output list M is:

Unanimity/Consensus	$(3, 4, \infty)$
Majority	$(2, 3, 4, 5, \infty)$
Compilation	$(1, 2, 3, 4, 5, \infty)$
Plurality	$((3, 4, \infty), 3)$

Compilation is useful to obtain all possible safety risks (*eg*, pairs of aircraft that are dangerously close to each other, or radar targets that are judged to be hostile, even though only one of the n computation channels "thinks" so) for subsequent in-depth analysis. For larger values of n, other schemes such as 2/3 *majority* and *second opinions* (values approved by at least two inputs) can be considered.

Notation

n	number of input lists
q	number of distinct elements in the n input lists
k	size of the longest input list.

The complexity of algorithm 5.2.1 is $O(n \cdot q)$, because an $O(n)$-time search for the minimal element is needed q times. Since it is possible that no two lists contain common elements, q can be as large as $n \cdot k$; the worst-case running time is $O(n^2 \cdot k)$. Like all other voting algorithms, this worst-case performance is almost never encountered in practice.

Even though algorithm 5.2.1 is most likely to be implemented in software, hardware realization is possible. One can envision n fixed-size dedicated queues, each associated with one of the n computation channels. The elements at the queue heads are the z_i $(i = 1, 2, \ldots, n)$. In each step, the smallest of these values is removed from the queues, a decision is made whether to output this value, and the process is repeated until all the head elements are "∞".

Algorithm 5.2.3: Approval Voting — Interval Representation

Sort the $2n$ values, l_i & h_i into ascending order to obtain the list N.

Form the $2n$-element list V such that:

$$V_i = \begin{cases} v_j, \text{ if } N_i = l_j \\ -v_j, \text{ if } N_i = h_j, \end{cases}$$

(this is done by initializing V appropriately and exchanging its elements in tandem with the exchanges required to sort N).

Initialize l, h, u, w to 0.

for $i = 1$ to $2n$ do

while $(i \leq 2n)$ and $(V_i \geq 0)$ do $u \leftarrow u + V_i$; $i \leftarrow i + 1$ endwhile

if $u > w$ then $w \leftarrow u$; $l \leftarrow N_{i-1}$; $h \leftarrow N_i$ endif

while $(i \leq 2n)$ and $(V_i \leq 0)$ do $w \leftarrow w + V_i$; $i \leftarrow i + 1$ endwhile

endfor

Output l, h, w. ◄

The complexity of algorithm 5.2.3 is dominated by the initial sorting since the remainder of the algorithm needs $O(n)$ time. Thus the complexity is $O(n \cdot \log(n))$ overall.

Algorithm 5.2.3 selects the lowest subinterval in the case of tie votes for several subintervals. The selection rule can be modified with knowledge about the object space. For example, with intervals on the real line, it is reasonable to assume that for tie votes, the widest interval is to be selected since it represents a wider range of approved values. This is easily done by modifying the condition for the if-statement to:

$$u > w \text{ or } [(u = w) \text{ and } (N_i - N_{i-1} > h - l)]$$

Example 5.2.4

Consider the intervals:

$$(l_1, h_1),\ (l_2, h_2),\ (l_3, h_3),\ (l_4, h_4)$$

on the real line as depicted in figure 5. Let,

$$v_1 = 3,\ v_2 = 2,\ v_3 = 2,\ v_4 = 1.$$

The lists N & V (after sorting) are:

$$N = (l_1, l_3, l_4, l_2, h_4, h_3, h_1, h_2)$$

$$V = (3, 2, 1, 2, -1, -2, -3, -2).$$

The output of algorithm 5.2.3 is then,

$$l = l_2,\ h = h_4,\ w = 8,$$

indicating that subinterval (l_2, h_4) has obtained all 8 votes (unanimous approval).

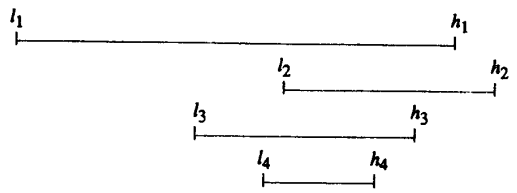

Figure 5. Approval Voting With Intervals On the Real Line

The two while-loops in algorithm 5.2.3 increase the efficiency of the algorithm by avoiding the additional work in the statement between them, in a great majority of cases. For example, with the 4 intervals of example 5.2.4, step 4 of the algorithm is executed only once. If efficiency were of no concern, a somewhat simpler algorithm could be used.

Theorem 5.2.5 establishes an interesting relationship between inexact and approval voting.

Theorem 5.2.5. With the real line as the input space, approximate voting with values $x_1, x_2, \ldots, x_n$ and comparison threshold ϵ is equivalent to approval voting with the input intervals $[x_i - \frac{1}{2}\epsilon, x_i + \frac{1}{2}\epsilon]$, $i = 1, 2, \ldots, n$. ◄

Example 5.2.6

Figure 6 depicts the intervals corresponding to the approval-voting formulation of an inexact plurality voting problem with:

$$x_1 = 5.3,\ x_2 = 5.4,\ x_3 = 5.2,\ x_4 = 5.6,\ x_5 = 5.4,$$

assuming the error threshold of $\epsilon = 0.1$ in establishing approximate equality. ◄

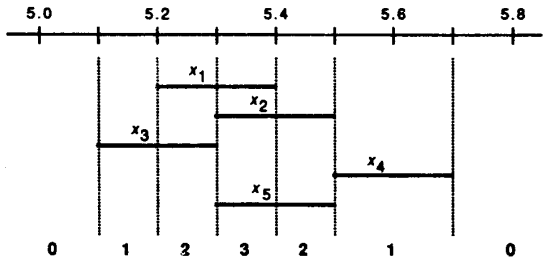

Figure 6. Approval-Voting Formulation Of an Inexact Voting-Problem

6. IMPLEMENTATION & EXTENSIONS

The algorithms for exact, inexact, and approval voting have demonstrated that the complexity of voting algorithms depends on the structure of the input object space. The algorithms were presented in an abstract form, with little attention to the efficiency of data structures & operations. Many application-dependent factors affect the hardware or software implementation details for such algorithms. However, tools & techniques to facilitate such implementations do exist. When the input object space is small, techniques for multiple-operand addition and parallel counting (well-studied problems in computer arithmetic [54]) can be used for hardware realization of the voting algorithms. For an ordered input object space, one can choose from an extensive library of sorting algorithms and networks to realize the most time-consuming phase of the algorithms. In other cases, particularly for the new approval voting schemes, more effort is needed to develop efficient designs.

Techniques for making the algorithms faster on the average were presented. While reducing the average-case complexity of voting algorithms can be quite important in many contexts, it might be unimportant in real-time applications with hard deadlines. However, even then, lower average-case running times can be used to advantage in a multiprocessing environment if it can be shown that the probability of missing a deadline due to excessive voting delays is comparable to the contribution of other sources of failures such as resource exhaustion or imperfect coverage [40]. Exploring the suitability of various voting strategies and their effects on tradeoffs between the correctness and timeliness of computation results remains a fruitful area of research.

Analyses, in this paper, for the complexity of the voting algorithms have been asymptotic and worst-case. Since in practice the number of input data objects that participate in voting is small, more detailed analyses are needed for comparing & selecting algorithms. Such analyses and associated algorithm selection guidelines remain to be developed. There are situations, however, when voting with a fairly large number of inputs is actually needed. For example:

- In image-processing filters where during each pass, pixel values can be replaced by values determined from voting on a predefined neighborhood of nearby points [11].
- 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 [50].

APPENDIX

A.1 Proof of Theorem 3.2.3

Any voting algorithm is based on comparison & combining: comparing two input objects and combining their votes if equal. The proof is in 2 parts:

1. Any voting algorithm must tally the votes for all distinct input values.
2. The complexity of vote-tallying is $\Omega(n \cdot \log(n))$.

Part 1 is easy to prove. If the vote for a particular value is not tallied, we can change the vote weights such that this particular value has the maximum total vote. But changing the votes does not change the decision structure of the voting algorithm and the instances of vote combining. Thus, incorrect output is produced if the original algorithm did not tally all the votes.

Part 2 uses a decision-tree argument similar to the one used for establishing a lower-bound for sorting. The number $L(n)$ of leaves in this binary decision tree is equal to the number of ways combining can occur. This is equal to the number of different partitions of a set of n elements into nonempty disjoint subsets. The number of partitions with m classes is $S_n^{(m)}$ (Stirling number of the second kind [26]). Hence:

$$L(n) = \sum_{m=1}^{n} S_n^{(m)}. \qquad \text{(A-1)}$$

Using the Szekeres-Binet approximation to (A-1) [15], for large n:

$$L(n) \approx (\beta + 1)^{-1/2} \cdot [1 - \beta/(12n)]$$

$$\cdot \exp[n \cdot (\beta - 1 + 1/\beta) - 1]$$

β is defined by $\beta \cdot \exp(\beta) = n$.

For large n,

$$\beta \approx \log(n) - \log(\log(n)) = O(log(n)),$$

$L(n) \approx (\log(n))^{-1/2} \cdot \exp(n \cdot \log(n))$.

The minimum number of levels in the decision tree, and thus the number of compare-combine operations in the worst case, is at least $\log_2(L(n)) \approx n \cdot \log(n)$. *Q.E.D.*

A.2 Proof of Theorem 3.3.2

Consider instances of the weighted plurality voting problem for which:

- input weights satisfy $v_1 < v_2 < \ldots < v_n$,
- the n input objects belong to $n-1$ or n equivalence classes (*ie*, all are distinct except possibly for x_q & x_r that are equal),
- $v_i + v_j > v_k$, for all i, j, 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 if $x_q = x_r$, the inputs x_q & x_r must be compared directly; all other comparisons reveal only 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 are equal, it might in fact sequence its operations such that x_q & x_r are compared after all other comparisons. Hence the need for $\Omega(n^2)$ worst-case comparisons.

To complete the proof for the weighted case, comparing all possible pairs of objects, requires $\Omega(n)$ working storage.

The proof for unweighted plurality voting is similar but slightly more involved. *Q.E.D.*

A.3 Proof of Theorem 3.3.3

The algorithm needs working storage space or "slots" for $p = \text{gilb}(V/t)$ different objects $z_1, z_2, \ldots, z_p$, each with an associated vote tally u_j (the u_j are initialized to 0, thus designating all p slots as empty). The next object x_i to be considered is compared to all the stored z_i. 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 u_k for the stored objects is found. If $v_i \leq u_k$, then x_i is discarded, and all stored vote tallies are decremented by v_i; otherwise all vote tallies are decremented by u_k and $(x_i, v_i - u_k)$ replaces one of the objects which is left with 0 vote tally.

Once all n input objects have been examined in this manner, any object with total vote tally of t or more is among the final p objects stored in the p available slots. Since only objects whose vote tallies are reduced to zero through the decrementation 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 decrementation process) p other distinct objects also lose the same vote. Thus if an object loses t votes in this process, a total of $(p + 1) \cdot t$ votes must have been lost. But this is impossible since $(p + 1) \cdot t > V$, the sum of all input votes. A second pass through the input, comparing each x_i to all remaining z_j, tallying the vote for each z_j, and keeping track of the largest vote tally, completes the algorithm. Each pass requires $O(n \cdot p)$ time. *Q.E.D.*

A.4 Proof of Theorem 5.2.5

The proof is quite simple. The intervals $[x_i - \epsilon/2, x_i + \epsilon/2]$ and $[x_j - \epsilon/2, x_j + \epsilon/2]$ overlap iff $|x_i - x_j| \leq \epsilon$. Hence, any maximal compatible set of input values obtained in step 1 of algorithm 4.1.3 corresponds to a maximal number of overlapping subintervals in algorithm 5.2.3. *Q.E.D.*

REFERENCES

[1] P. Agrawal, "Fault tolerance in multiprocessor systems without dedicated redundancy", *IEEE Trans. Computers*, vol 37, 1988 Mar, pp 358-362.

[2] P.E. Ammann, J.C. Knight, "Data diversity: An approach to software fault tolerance", *IEEE Trans. Computers*, vol 37, 1988 Apr, pp 418-425.

[3] A. Avizienis *et al*, "The STAR (self-testing-and-repairing) computer: An investigation of the theory and practice of fault-tolerant computer design", *IEEE Trans. Computers*, vol C-20, 1971 Nov, pp 1312-1321.

[4] A. Avizienis, "The N-version approach to fault-tolerant software", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1491-1501.

[5] O. Babaoglu, "On the reliability of consensus-based fault-tolerant distributed computing systems", *ACM Trans. Computer Systems*, vol 5, 1987 Nov, pp 394-416.

[6] D. Barbara, H. Garcia-Molina, "The reliability of voting mechanisms", *IEEE Trans. Computers*, vol C-36, 1987 Oct, pp 1197-1208.

[7] D.M. Blough, G.F. Sullivan, "A comparison of voting strategies for fault-tolerant distributed systems", *Proc. 9th Symp. Reliable Distributed Systems*, 1990 Oct, pp 136-145.

[8] D.M. Blough, G.F. Sullivan, "An analysis of the voting problem for fault-tolerant systems", unpublished manuscript, 1992.

[9] R.S. Boyer, J.S. Moore, "MJRTY: A fast majority vote algorithm", *Automated Reasoning: Essays in Honor of Woody Bledsoe* (R.S. Boyer, Ed), 1991; Kluwer.

[10] S.S. Brilliant, J.C. Knight, N.G. Leveson, "The consistent comparison problem in N-version software", *Software Engineering Notes*, ACM SIGSOFT, vol 12, 1987 Jan, pp 29-34.

[11] D.R.K. Brownrigg, "The weighted median filter", *Commun. ACM*, vol 27, 1984 Aug, pp 807-818.

[12] D. Campbell, T. McNeill, "Finding a majority when sorting is not available", *The Computer J.*, vol 34, 1991 Apr, p 186.

[13] L. Chen, A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation", *Proc. Int'l Symp. Fault-Tolerant Computing*, 1978 Jun, pp 3-9; Toulouse, France.

[14] Y. Chen, T. Chen, "Implementing fault tolerance via modular redundancy with comparison", *IEEE Trans. Reliability*, vol 39, 1990 Jun, pp 217-225.

[15] F.N. David, D.E. Barton, *Combinatorial Chance*, 1962; p 315; Hafner.

[16] P.T. DeSousa, F.P. Mathur, "Sift-out modular redundancy", *IEEE Trans. Computers*, vol C-27, 1978 Jul, pp 624-627.

[17] D. Dolev, L. Lamport, M. Pease, R. Shostak, "The Byzantine generals", *Concurrency Control and Reliability in Distributed Systems* (B.K. Bhargava, Ed), 1987, pp 348-369; Van Nostrand Reinhold.

[18] H. Garcia-Molina, D. Barbara, "How to assign votes in a distributed system", *J. ACM*, vol 32, 1985 Oct, pp 841-860.

[19] D.K. Gifford, "Weighted voting for replicated data", *Proc. 7th ACM SIGOPS Symp. Operating System Principles*, 1979 Dec, pp 150-159; Pacific Grove, CA.

[20] D. Gries, "A hands-in-the-pocket presentation of a k-majority vote algorithm", *Formal Development of Programs and Proofs* (E.W. Dijkstra, Ed), 1990, pp 43-45; Addison-Wesley.

[21] A.L. Hopkins, T.B. Smith, J.H. Lala, "FTMP: A highly reliable fault-tolerant multiprocessor for aircraft", *Proc. IEEE*, vol 66, 1978 Oct, pp 1221-1239.

[22] S. Jajodia, D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database", *ACM Trans. Database Systems*, vol 15, 1990 Jun, pp 230-280.

[23] B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, 1989; Addison-Wesley.

[24] N. Kanekawa, H. Maejima, H. Kato, H. Ihara, "Dependable onboard computer systems with a new method stepwise negotiating voting", *Proc. Int'l Symp. Fault-Tolerant Computing*, 1989 Jun, pp 13-19; Chicago.

[25] J.C. Knight, N.G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming", *IEEE Trans. Software Engineering*, vol SE-12, 1986 Jan, pp 96-109.

[26] D.E. Knuth, *The Art of Computer Programming vol 1: Fundamental Algorithms* (2nd ed), 1973, subsection 1.2.6, problem 64, pp 73 & 489; Addison-Wesley.

[27] Z. Kohavi, *Switching and Finite Automata Theory* (2nd ed), 1978, pp 333-347; McGraw-Hill.

[28] P.R. Lorczak, A.K. Caglayan, D.E. Eckhardt, "A theoretical investigation of generalized voters for redundant systems", *Proc. Int'l Symp. Fault-Tolerant Computing*, 1989 Jun, pp 444-451; Chicago.

[29] J. Losq, "A highly efficient redundancy scheme: Self-purging redundancy", *IEEE Trans. Computers*, vol C-25, 1976 Jun, pp 569-578.

[30] F.P. Mathur, A. Avizienis, "Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair", *AFIPS Conf. Proc*, vol 36 (Spring Joint Computer Conf.), 1970, pp 375-383; AFIPS Press.

[31] F.P. Mathur, P. DeSousa, "Reliability modeling and analysis of general modular redundant systems", *IEEE Trans. Reliability*, vol R-24, 1975 Dec, pp 296-299.

[32] D.F. McAllister, C.-E. Sun, M.A. Vouk, "Reliability of voting in fault-tolerant software with small output spaces", *IEEE Trans. Reliability*, vol 39, 1990 Dec, pp 524-534.

[33] J. Misra, D. Gries, "Finding repeated elements", *Science of Computer Programming*, vol 2, 1982, pp 143-152. (See also correspondence in *The Computer J.*, vol 35, 1992 Jun, p 298.)

[34] L.E. Moser, V. Kapur, P.M. Melliar-Smith, "Probabilistic language analysis of weighted voting mechanisms", *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, 1990 May, pp 67-73.

[35] B. Parhami, "Voting networks", *IEEE Trans. Reliability*, vol 40, 1991 Aug, pp 380-394.

[36] B. Parhami, "The parallel complexity of weighted voting", *Proc. Int'l Symp. Parallel and Distributed Computing and Systems*, 1991 Oct, pp 382-385; Washington, DC.

[37] B. Parhami, "High-performance parallel pipelined voting networks", *Proc. Int'l Parallel Processing Symp*, 1991 Apr/May, pp 491-494; Anaheim, CA.

[38] B. Parhami, "Design of m-out-of-n bit-voters", *Proc. 25th Asilomar Conf. Signals, Systems, and Computers*, 1991 Nov, pp 1260-1264; Pacific Grove, CA.

[39] B. Parhami, "Optimal algorithms for exact, inexact, and approval voting", *Proc. Int'l Symp. Fault-Tolerant Computing*, 1992 Jun, pp 444-451; Boston.

[40] B. Parhami, C.Y. Hung, "Scheduling of replicated tasks to meet correctness requirements and deadlines", *Proc. 26th Hawaii Int'l Conf. System Sciences*, 1993 Jan, vol 2, pp 506-515.

[41] J.-F. Paris, "Voting with a variable number of copies", *Proc. Int'l Symp. Fault-Tolerant Computing*, 1986 Jul, pp 50-55; Vienna.

[42] W.H. Pierce, "Adaptive decision elements to improve the reliability of redundant systems", *IRE Int'l Conv. Record*, 1962 Mar, pp 124-131.

[43] K.G. Shin, J.W. Dolter, "Alternative majority-voting methods for real-time computing", *IEEE Trans. Reliability*, vol 38, 1989 Apr, pp 58-64.

[44] D.P. Siewiorek, E.J. McCluskey, "Switch complexity in systems with hybrid redundancy", *IEEE Trans. Computers*, vol C-22, 1973 Mar, pp 276-282.

[45] D.P. Siewiorek, E.J. McCluskey, "An iterative cell switch design for hybrid redundancy", *IEEE Trans. Computers*, vol C-22, 1973 Mar, pp 290-297.

[46] D.P. Siewiorek, "Reliability modeling of compensating module failures in majority voted redundancy", *IEEE Trans. Computers*, vol C-24, 1975 May, pp 525-533.

[47] D.P. Siewiorek *et al*, "C.vmp: A voted multiprocessor", *Proc. IEEE*, vol 66, 1978 Oct, pp 1190-1198.

[48] D.P. Siewiorek, R.S. Swarz, *Reliable Computer Systems: Design and Evaluation* (2nd ed), 1992, pp 138-146 (Discussion on voting); Digital Press.

[49] J.R. Sklaroff, "Redundancy management techniques for space shuttle computers", *IBM J. Research and Development*, vol 20, 1976 Jan, pp 20-28.

[50] S.Y.H. Su, M. Cutler, M. Wang, "Self-diagnosis of failures in VLSI tree array processors", *IEEE Trans. Computers*, vol 40, 1991 Nov, pp 1252-1257.

[51] Z. Tong, R.Y. Kain, "Vote assignments in weighted voting mechanisms", *IEEE Trans. Computers*, vol 40, 1991 May, pp 664-667.

[52] U. Voges, "Use of diversity in experimental reactor safety systems", *Software Diversity in Computerized Control Systems*, 1988, pp 29-49; Springer-Verlag.

[53] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components", *Automata Studies* (Annals of Mathematics Studies, num 34), (C.E. Shannon, J. McCarthy, Eds), 1956, pp 43-98; Princeton Univ. Press.

[54] S. Waser, M.J. Flynn, *Introduction to Arithmetic for Digital System Designers*, 1982; Holt, Rinehart, & Winston.

[55] J.H. Wensley, *et al*, "SIFT: design and analysis of a fault-tolerant computer for aircraft control", *Proc. IEEE*, vol 66, 1978 Oct, pp 1240-1255.

[56] J.H. Wensley C.S. Harclerode, "Programmable control of a chemical reactor using a fault tolerant computer", *IEEE Trans. Industrial Electronics*, vol IE-29, 1982 Nov, pp 258-264.

AUTHOR

Dr. Behrooz Parhami, Professor; Dept. of Electrical & Computer Eng'g; Univ. of California; Santa Barbara, California 93106-9560 USA.
Internet (e-mail): parhami@ece.ucsb.edu

Behrooz Parhami (S'70-M'73-SM'78) received his PhD (1973) in Computer Science from the University of California, Los Angeles. From 1974 to 1988 he was with Sharif University of Technology in Tehran, Iran where he was involved in educational planning, curriculum development, standardization efforts, technology transfer, and various editorial responsibilities. His current research deals with parallel architectures and algorithms, high-speed computer arithmetic, and dependable (fault-tolerant) computing. His technical publications include over 100 papers in journals and international conferences, two Farsi-language textbooks, and an English/Farsi glossary of computing terms. He is a member of the Assoc. for Computing Machinery and the British Computer Society, and is a Distinguished Member of the Informatics Society of Iran for which he served as a founding member, Editor-in-Chief, and President during 1979-84. He also served as Chair'n of IEEE Iran Section (1977-86) and received the IEEE Centennial Medal in 1984.

Manuscript received 1994 January 3

IEEE Log Number 94-00317 ◄TR►