OPTIMAL ALGORITHMS FOR EXACT, INEXACT, AND APPROVAL VOTING

Behrooz Parhami

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

Abstract

Voting is an important operation in the realization of ultrareliable systems that are based on the multi-channel computation paradigm. In this paper, the design of optimal n-way voting algorithms based on the structure of the input object space is considered. The design techniques are then extended to inexact and approval voting schemes. It is shown that efficient $\theta(n)$-time voting algorithms can be designed when the input object space is small. Next in the hierarchy is the case of a totally-ordered object space that supports worst-case $\theta(n \log n)$ algorithms for both exact and inexact voting as well as for certain approval-voting schemes. An unordered input object space leads to worst-case $\Omega(n^2)$ algorithms, even when a distance metric can be defined on the input object space. Some observations on the relationship of voting to other well-studied problems, particularly sorting, are also included.

Keywords: Approximate voting, design diversity, inexact voting, multi-channel computation, *n*-modular redundancy.

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 possible combination of hardware/program/data redundancy and/or diversity. Depending on the data volume and the frequency of voting, hardware or software voting schemes may be appropriate. Low-level voting with high frequency requires the use of 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 or less reliable versions was first suggested in the mid 1950s. Since then, the concept has been practically utilized in several fault-tolerant computer systems and has been extended and refined in many different ways. Hardware voters that have been described in the literature are essentially "bit-voters" that compute a majority function on *n* input bits [JOHN89], [SIEW82]. Combined bit voting and disagreement detection has also been discussed [CHEN90].

Proposed software voters are quite varied and possess a wide range of features. The earliest software voters are found in the design of modular multiprocessors with replicated software. For example, the voter routine in SRI's SIFT design [WENS78] is invoked by any task which requires inputs for a new iteration. In the Space Shuttle's 4-way software voting scheme [SKLA76], selected data items are computationally combined to form "compare words" that are periodically exchanged and compared in 4 out of the 5 onboard computers. Another example is the "Stepwise Negotiating Voting" scheme of [KANE89] which essentially amounts to a 2-out-of-*n* voting strategy. The advantages of such "relaxed" (non-majority) voting schemes have been discussed by others as well [AGRA88], [BLOU90], [PARI86]. Researchers in the field of software diversity have designed voters that are suitable for processing the results obtained by multiple versions of a program and have contributed techniques for handling approximate results [AVIZ85], [BRIL87], [KNIG86], [LORC89], [VOGE88]. Software voters have also been designed or proposed in connection with the management of replicated data in distributed systems [BABA87], [BARB87], [GARC85], [GIFF79], [JAJO90] to assure database reliability and/or consistency.

2. The Problem and Its Variations

I define the weighted voting problem as follows:

2.1. Definition: The Weighted Voting Problem — Given n input data objects $x_1, x_2, \ldots, x_n$, with associated non-negative real votes (weights) $v_1, v_2, \ldots, v_n$, compute the output y and its vote w such that y is "supported by" a number of input data objects with votes totalling w, where w satisfies a condition associated with the desired voting subscheme:

Threshold voting subschemes:

- Unanimity voting: $w = \Sigma v_i$
- Byzantine voting: $w > \frac{2}{3}\Sigma v_i$
- Majority voting: $w > \frac{1}{2}\Sigma v_i$
- 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)

Plurality voting subscheme:

- No other y' is supported by inputs having more votes (if $w \leq \frac{1}{2}\Sigma v_i$, y may be non-unique)

The term "supported by" can be defined in several ways, leading to different voting schemes, each of which has the subschemes listed above. With exact voting, an input object x_i supports y iff $x_i = y$. With inexact voting, approximate inequality ($\approx$) is defined in some suitable way (e.g. by providing a comparison threshold ε in the case of numerical values or, more generally, a distance measure d in a metric space) and x_i supports y iff $x_i \approx y$. Finally with approval voting, y must be a member of the approved set of values that x_i defines (more on this later). ▮

The reason I have chosen to start with weighted voting is threefold. First, it is more general than simple voting and thus useful in a wider context. Setting all weights to 1 yields simple voting as a special case. Second, it turns out that in most cases, weighted voting isn't 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 will be pointed out. Third, some fixed-weight schemes are essentially adaptive over the long run (e.g., disabling faulty units in some n-way voting schemes may be equivalent to setting their initial votes of 1 to 0).

Several important voting schemes, such as median and mean voting, are not covered by Definition 2.1 because they are specific to certain input types and do not apply to a general object space. I will deal with such voting schemes where applicable.

3. Algorithms for Exact Voting

Although exact voting has been studied and 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 more complex data types, the situation changes. I divide my discussion of algorithms according to the size and structure of the input object space. Throughout this section, which deals with "exact" comparisons, equality is assumed to be transitive.

3.1. Small Object Space

For a small object space, an efficient linear-time algorithm can be devised. Let the input object space be of size δ and let the set of possible values or classes of objects be encoded by integers $\{1, 2, 3, \ldots, \delta\}$. Then the algorithm consists of tallying the votes for each of the δ possible values/classes and then selecting the appropriate output.

3.1.1. Algorithm: Exact Voting — Small Object Space of size δ

```
Let $u$ be the vote-tally vector, with $u_i$ ($1 \le i \le \delta$) holding the vote for the object class $i$
for $i = 1$ to $\delta$ do $u_i := 0$ endfor
for $i = 1$ to $n$ do $j := class(x_i)$; $u_j := u_j + v_i$ endfor
$(y, w) := select(u)$ ▮
```

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 worst-case execution time of Algorithm 3.1.1 is $O(n + \delta)$.

Algorithm 3.1.1 can be easily implemented in hardware also. Direct hardware realization for the case of bit-voting ($\delta = 2$) is depicted in Figure 1. Actually with bit voting, both hardware and software implementations can be simplified by tallying the vote for only one of the two values. With this simplification, the design of Figure 1 becomes the "arithmetic-based" design technique for bit-voters that I have previously discussed in the context of voting networks [PARH91]. Despite its seemingly low $O(n\delta)$ complexity and $O(\log n)$ delay, hardware implementation of this algorithms is only practical 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:

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

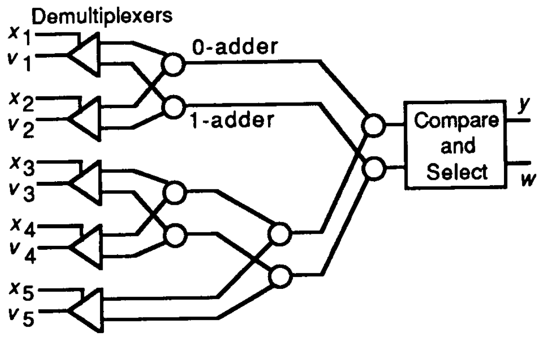

Figure 1. Hardware realization of five-way weighted bit-voting: The triangular demultiplexers use the x_i data bits as "control" or "selection" signals and have widths equal to the number of bits in the binary representations of the input votes v_i. Each circular adder attaches its carry-out signal to 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 will have $\lfloor \log_2 v_{\max} + \log_2 n \rfloor + 1$ bits.

3.2. Totally Ordered Object Space

With a large object space (e.g. 32-bit integers), Algorithm 3.1.1 is inapplicable. In this subsection, I show that if the input object space is totally ordered, then the following algorithm based on sorting is optimal. All sorting is assumed to be in ascending order.

3.2.1. Algorithm: Exact Plurality Voting — Totally Ordered Object Space

```
Sort 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 $i$} ▮
```

The algorithm can be easily modified for other voting schemes. For example, majority voting requires appending, at the end, the statement: if $w \le \frac{1}{2}\Sigma v_i$ then $w := 0$ (no-quorum indicator). weighted median voting follows.

3.2.2. Algorithm: Weighted Median Voting — Totally Ordered Object Space

```
Sort in place the set of records $(x_i, v_i)$ by the $x_i$ keys.
$q = \frac{1}{2}\Sigma v_i$; $u := v_1$; $i := 1$
while $u < q$ do $i := i + 1$; $u := u + v_i$ endwhile
$y := x_i$ ▮
```

Since sorting is asymptotically an $\Omega(n \log n)$ operation (it in fact 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 exact voting algorithms:

```
if $x_1 = x_2 = ... = x_n$ then $(y, w) := (x_1, \Sigma v_i)$ stop endif
```

For example, if each computation channel produces a correct result with probability 0.999 and 5-way non-weighted voting is used to obtain a result with even higher reliability, the effect of this additional test is to make the running time of the algorithm linear in at least 99.5 percent of the cases and the average running time becomes almost linear. It may 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.

Hardware realization of Algorithm 3.2.1 or its variants may also be contemplated. The straightforward hardware design will consist of a sorting network followed by a combining and a max-selection network. It is also possible to modify the cells in a sorting network in order to convert it into a vote-tallier [PARH91] that requires very little additional hardware to become a voting network.

I now show that Algorithm 3.2.1 is optimal in the sense that voting with a large object space has $\Omega(n \log n)$ time complexity.

3.2.3. Theorem: The complexity of n-way plurality voting, as specified in Definition 2.1, with large object space is $\Omega(n \log n)$. ▮

Proof: Any voting algorithm will be based on comparison and combining: comparing two input objects and combining their votes if equal. I prove the result in two phases. First I show that any voting algorithm must tally the votes for all distinct input values. Then, I prove that the complexity of vote-tallying is $\Omega(n \log n)$. The first part is easy to prove. If the vote for a particular value is not tallied, one 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 will be produced if the original algorithm did not tally all the votes. For the second part, I use 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 $S_n^{(m)}$, or Stirling number of the second kind [KNUT73]. Hence:

$$L(n) = \sum_{m=1}^{n} S_n^{(m)}$$

Using the Szekeres-Binet approximation to the above sum of Stirling numbers [DAVI62], one gets for large n

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

where β is defined by $\beta e^{\beta} = n$. For large n, we have $\beta \approx \log n - \log\log n = O(\log n)$ and:

$$L(n) \approx (\log n)^{-1/2}\, e^{n \log n}$$

The minimum number of levels in the decision tree, and thus the number of compare-combine operations is the worst case, is at least $\lg(L(n)) \approx n \log n$. ▮

3.3. Unordered Object Space

An $O(n^2)$ algorithm for plurality voting can be devised in this case. Here is a high-level description of the algorithm.

3.3.1. Algorithm: Exact Plurality Voting — Unordered Object Space

```
Let $z$ and $u$ be the distinct-input and vote-tally vectors ($z_j$ the $j$th distinct input encountered, $u_j$ 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 $\exists\, j \le k$ 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 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 due to the fact that k remains 1 with very high probability. Because of excellent average-case performance, there is no need to resort to more efficient search schemes to find j inside the **for** loop. Since 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), the following lower bound is established.

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

It turns out that one can devise simpler threshold voting algorithms in this case. Consensus voting clearly has linear complexity. Less obvious is the following result.

3.3.3. Theorem: With an unordered input object space, t-out-of-Σv_i voting can be performed with time complexity $O(np)$, where $p = (\Sigma v_i)/t$. ▮

Proof: The algorithm to be described needs working storage space for $p = \lfloor(\Sigma v_i)/t\rfloor$ different objects $z_1, z_2, ...,$

z_p, each of which has an associated vote tally u_j (all u_js are initialized to 0). The next object x_i to be considered is compared to the stored z_js. If $x_i = z_j$, 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 thus far, then (x_i, v_i) is stored also. If p objects are stored, then the minimum vote tally u_k for the stored objects is found. If $v_i \le u_k$, then x_i is discarded and all stored vote tallies are decremented by v_i. If $v_i > u_k$, then all vote tallies are decremented by u_k and $(x_i, v_i - u_k)$ replaces one of the objects with 0 vote tally. One can prove that any object with total vote tally of t or more will be among the final p objects. The proof (a generalization of a recently published proof for m-out-of-n voting [CAMP91]) is based on the observation that every time an object loses votes (due to the vote decrementation), p other distinct objects also lose the same vote. Thus if an object loses t votes in the process, a total of $(p + 1)t$ votes must have been lost. But this is impossible since $(p + 1)t > \Sigma v_i$. 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 will complete the algorithm. Each pass requires $O(np)$ time, thus proving the desired result. ∎

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

3.3.5. 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. ∎

3.3.6. 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 inputs $(A, 3)$, $(B, 2)$, $(B, 2)$, $(A, 1)$, $(C, 3)$, $(A, 4)$ in presentation order. A single working storage (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 algorithm steps. 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. ∎

4. Algorithms for Inexact Voting

Algorithms for inexact voting are in general more complex than their exact-voting counterparts because of the non-transitivity of approximate voting. Although the notion of approximate voting and its applications have been discussed in the literature, only one published inexact voting algorithm is known to me [LORC89] and that algorithm happens to be incorrect (more on this in Subsection 4.1). Inexact voting with a small object space, though theoretically possible, is not of practical interest. Thus in this section, a large object space is assumed.

4.1. General Weighted Inexact Voting

Dealing with approximate equality requires defining a real-valued distance function $d: X^2 \to R$ on pairs of objects in the input object space. Then two objects x_i and x_j are approximately equal if $d(x_i, x_j) \le \varepsilon$ where ε is a suitably small comparison threshold. The following definition captures some properties of d consistent with the intuitive notion of approximate equality.

4.1.1. Definition: The function $d: X^2 \to R$ is a distance function for the object space X if for all $x_i, x_j \in X$:

1. $d(x_i, x_j) \ge 0$
2. $d(x_i, x_j) = 0$ iff $x_i = x_j$
3. $d(x_i, x_j) = d(x_j, x_i)$ ∎

Note that I have excluded from Definition 4.1.1 the "triangle inequality", viz $d(x_i, x_k) \le d(x_i, x_j) + d(x_j, x_k)$, which would make (X, d) a metric space. This is done in view of the fact that the "triangle" restriction would not simplify the voting algorithm while it needlessly eliminates some potentially useful definitions of distance.

4.1.2. Example: Let our objects be pairs of integers (i, j) denoting points on a two-dimensional grid. Define $d((i_1, j_1), (i_2, j_2))$ as $\min(|i_1 - i_2|, |j_1 - j_2|)$. Accordingly, two objects will be 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 that defined in Example 4.1.2 may not seem particularly useful for common applications, there is no compelling reason to exclude them (by restricting our discussion to metric spaces) when the exclusion does not lead to simpler voting algorithms.

Given input objects $x_1, x_2, \ldots, x_n$ with associated votes $v_1, v_2, \ldots, v_n$ and the distance function $d: X^2 \to R$ satisfying Definition 4.1.1, the following algorithm can be used for inexact voting.

4.1.3. Algorithm: Weighted Inexact Voting

1. Determine a maximal-vote subset S of the set of n input objects such that for all $x_i, x_j \in S$, $d(x_i, x_j) \le \varepsilon$ (this subset may not be unique).
2. In general, S may contain identical elements; i.e., objects x_i and x_j with $d(x_i, x_j) = 0$. Combine the votes for identical objects in S, getting the set of distinct objects $S' = \{z_1, z_2, \ldots, z_m\}$ and associated vote tallies $u_1, u_2, \ldots, u_m$, with $m \le n$ and $d(z_i, z_j) > 0$.
3. $y := select(z, u)$; $w := \sum_{i=1}^{m} u_i$. ∎

The above description is at a high level and each step must be further clarified and 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 [KOHA78]. Here is a brief overview of the procedure. An $(n-1)$-by-$(n-1)$ triangular table is constructed in which entry (i, j) indicates the "compatibility" (approximate equality) of x_i and 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. The maximal compatibility classes thus obtained form a "cover" on the set of objects (and not a partition as asserted in Subsection 2.1 of [LORC89]).

It is also possible to define the computation in Step 1 recursively. Let the computed function in Step 1 be denoted by $S = cluster(\{x_1, x_2, \ldots, x_n\})$. Given any set $M = \{z_1, z_2, \ldots, z_k\}$ of objects, let $z_i, z_j \in M$ be a pair of objects that are furthest apart. Then $cluster(M)$ is M if $d(z_i, z_j) \le \varepsilon$. Otherwise, $cluster(M)$ is the larger of

$cluster(M - \{z_i\})$ and $cluster(M - \{z_j\})$. The resulting implementation will be fairly efficient in view of the fact that in practice recursion stops after zero or one level with very high probability. Step 2 is simple and can in fact 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_i having maximal vote. For a metric space, generalized median voting (recursively removing an object pair with the largest distance until only a single object or two objects are left, then picking one at will) can be used as suggested in [LORC89]. For numerical values, the mean selection rule can also be applied. In all the above cases, 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 vote total 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" between two strings (the minimal number of symbol insertions, deletions, or substitutions that would convert one string into the other) and the voting result defined as a string for which the 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 is $\Omega(n^2)$.

4.1.4. Example: 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. The distance function is defined as $d(x_i, x_j) = |x_i - x_j|$.

Objects	Votes
$x_1 = 1.300$	$v_1 = 2$
$x_2 = 1.310$	$v_2 = 3$
$x_3 = 1.330$	$v_3 = 4$
$x_4 = 1.340$	$v_4 = 3$
$x_5 = 1.350$	$v_5 = 1$

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, and 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. ▮

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

4.1.5. Theorem: The 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 based on sorting can be devised. First, I formalize the notion of a totally ordered object space for inexact voting.

4.2.1. Definition: An object space X is totally ordered with respect to the distance function d if for any three distinct objects $x_i, x_j, x_k \in X$, we have $d(x_i, x_k) = |d(x_i, x_j) \pm d(x_j, x_k)|$. ▮

Definition 4.2.1 establishes a total order as follows. Pick two distinct elements x_i and 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 and x_k by these 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 utilized to obtain a worst-case $O(n \log n)$ inexact voting algorithm.

4.2.2. Algorithm: 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 $(x_{n+1}, v_{n+1}) = (\infty, 0)$
$i := j := 1$; $u := w := 0$
while $j \le n$ do
    while $d(x_i, x_j) \le \varepsilon$ 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
$y := select(first, last)$ {$w$ has already been computed} ▮
```

The selection function here takes the indices of two 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 time complexity of Algorithm 4.2.2 is dominated by the $O(n \log n)$ sorting phase, as the rest of the algorithm runs in linear time. Since inexact voting is at least as hard as exact voting, one can state as a corollary to Theorem 3.2.3 the following result which establishes the optimality of Algorithm 4.2.2.

4.2.3. Corollary: The complexity of n-way inexact plurality voting is $\Omega(n \log n)$. ▮

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 $S = cluster(\{x_1, x_2, \ldots, x_n\})$ where $cluster(M)$ is M if $\max(M) - \min(M) \le \varepsilon$ or else it is obtained from $M - \{\max(M)\}$ or $M - \{\min(M)\}$, whichever is larger. The advantage of this method is that it achieves average-case linear running time.

4.2.4. Example: Consider the inputs of Example 4.1.4, which are already in sorted order, and the same distance function. In executing 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 x_3, x_4, x_5 and the output vote is $w = 8$. ▮

5. Algorithms for Approval Voting

In ordinary sociopolitical context, approval voting is an election process whereby each participant, rather than picking just the one "best" candidate, votes for a subset of candidates who are qualified for the office or position under consideration. The candidate with the highest approval vote tally wins the election. Some restrictions may apply to the subset that each voter can approve (e.g., there may be a maximum size). Such details, as well as certain disadvantages of approval voting, are not relevant to our discussion here. An important advantage of approval voting (and one that is also relevant in the context of dependable computing) is that a lesser qualified candidate will not get the highest vote total because of votes splitting among several better, but almost equally qualified, candidates. For our purposes, approval voting means that each input to the voting process consists of a set (finite or infinite) of approved values. Multiple approved values may result from non-unique answer to a given problem or from uncertainties in the solution process. Either way, the value or set of values with the highest approval vote will emerge as output.

As an example of where approval voting might be useful, consider a process control application that requires the periodic determination of a safe setting for a particular system variable. In general, there may 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 and other system considerations.

5.1. Small Object Space

Let the input object space be of size δ and let the set of possible values or classes of objects be encoded by integers $\{1, 2, 3, \ldots, \delta\}$. Since δ is small, the ith approval set can be represented by the bit vector x_i with components x_{ij} ($x_{ij} = 1$ 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.

5.1.1. Algorithm: Approval Voting — Small Object Space of size δ

```
Let $u$ be the vote-tally vector, with $u_i$ ($1 \le i \le \delta$) holding the vote for the object class $i$
for $i = 1$ to $\delta$ do $u_i := 0$ endfor
for $i = 1$ to $n$ do
    for $j = 1$ to $\delta$ do if $x_{ij} = 1$ then $u_j := u_j + v_i$ endfor
endfor
$(y, w) := select(u)$ ▮
```

The pair of nested loops above involve n and 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 $O(n\delta)$.

A slightly modified version of the circuit of Figure 1 can be used for hardware implementation of this algorithm if the bits x_{ij} are applied serially to the x_i inputs and pipelining is used. It is fairly easy to show the optimality of this algorithm leading to:

5.1.2. Theorem: Algorithm 5.1.1 represents an optimal n-way voting scheme. ▮

5.2. Totally Ordered Object Space

Just as in the case of exact voting, a large object space renders the above approach impractical. The set of approved values associated with each input can be represented in different ways. When all approval sets are relatively small, they can be represented by lists. In this case, I assume that the approval lists are sorted in ascending order and that each list is terminated 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 and output. For large approval sets, I consider here only the case of interval representation; i.e., all values from some lower bound l_i to an upper bound h_i are approved and I denote this approval interval by $[l_i, h_i]$.

5.2.1. Algorithm: 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 $i \in S$;
    $z := \min(z_1, z_2, \ldots, z_n)$;
    $S := \{j \mid z_j = z\}$; $u = \sum_{i \in S} v_i$ ;
    Append the value-vote pair $(z, u)$ to $N$
endwhile
Compute the output list $M := select(N)$. ▮
```

5.2.2. Example: Consider the approval lists $L_1 = (1, 2, 3, 4, \infty)$, $L_2 = (2, 3, 4, 5, \infty)$, and $L_3 = (3, 4, 5, \infty)$. Just before final selection step of Algorithm 5.2.1 is executed, the working list N is ((1,1), (2,2), (3,3), (4,3), (5,2), (∞,3)). Depending on the selection rule applied, the output list M will be:

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

Compilation is useful in situations where one wants to compile all possible safety risks (e.g., 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 "seconded opinions" (values approved by at least two inputs) may be considered. ▮

The complexity of Algorithm 5.2.1 is $O(nq)$, where q is the number of distinct elements that appear in the n input lists. This is true since 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 nk and the worst-case running time is $O(n^2k)$, where k is the size of the longest input list. Like all other voting algorithms, this worst-case performance is almost never encountered in practice.

Even though Algorithm 5.2.1 will most likely be implemented in software, hardware realization is also 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_is ($i = 1,2,...,n$). In each step, the smallest of these values are removed from the queues and a decision is made whether to output this value. The process repeats until all head elements are "∞".

5.2.3. Algorithm: Approval Voting — Interval Representation; the ith input object is $[l_i, h_i]$

Sort the $2n$ values l_i and h_i into ascending order to obtain the list N. Also form the $2n$-element list V such that $V_i = v_j$ if $N_i = l_j$ and $V_i = -v_j$ if $N_i = h_j$ (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 $i \le 2n$ and $V_i \ge 0$ 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 $i \le 2n$ and $V_i \le 0$ 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 $O(n \log n)$ 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 in the case of tie votes, the widest interval is to be selected since it represents a wider range of approved values. This can be easily accomplished by modifying the condition for the if-statement to:

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

5.2.4. Example: Consider the intervals (l_1,h_1), (l_2,h_2), (l_3,h_3), and (l_4,h_4) on the real line as depicted in Figure 2 and assume $v_1 = 3$, $v_2 = 2$, $v_3 = 2$, $v_4 = 1$. The lists N and V (after sorting) are $N = (l_1, l_3, l_4, l_2, h_4, h_3, h_1, h_2)$ and $V = (3, 2, 1, 2, -1, -2, -3, -2)$. The output of Algorithm 5.2.3 is then $l = l_2$, $h = h_4$, $w = 8$, indicating that the subinterval (l_2, h_4) has obtained all of the 8 votes or unanimous approval. ∎

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 four intervals of Example 5.2.4, Step 4 of the algorithm is executed only once. If efficiency were of no concern, a simpler algorithm could be used.

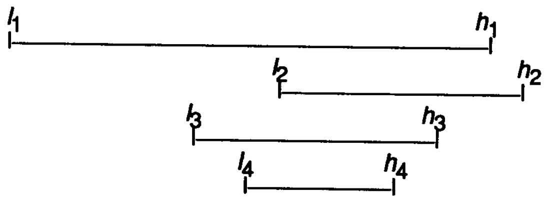

Figure 2. Example intervals on the real line.

The following result establishes an interesting relationship between inexact and approval voting.

5.2.5. Theorem: 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 $[x_i - \varepsilon/2, x_i + \varepsilon/2]$, $i = 1,2,...,n$.

Proof: Immediate upon noting that the intervals $[x_i - \varepsilon/2, x_i + \varepsilon/2]$ and $[x_j - \varepsilon/2, x_j + \varepsilon/2]$ overlap iff $|x_i - x_j| \le \varepsilon$. ∎

6. Conclusions

I have presented algorithms for exact, inexact, and approval voting and have demonstrated that the complexity of voting algorithms differs depending on the structure of the input space. The complexity of n-way voting ranges from $O(n)$ for a small object space to $O(n^2)$ for a large object space having no particular structure, with the $O(n \log n)$ complexity in the case of a totally ordered object space falling between the two extremes.

The results in this paper show that with a totally ordered object space, voting is intimately related to sorting and has the same complexity for both exact and inexact voting as well as for approval voting with intervals. In fact, for hardware voting, sorting networks can be easily modified and augmented to yield voting networks [PARH91]. When the input space is small, techniques for multiple-operand addition and parallel counting (both fairly well-studied problems in the field of computer arithmetic) can be used for hardware realization of the voting algorithms.

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 during each pass, pixel values may be replaced by values determined from voting on a predefined neighborhood of nearby points [BROW84]. 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 [SUSY91].

Finally, in several places, I have referred to average-case complexity of voting algorithms. In real-time applications with hard deadlines, the gain over the worst-case performance may seem unimportant. However, even in such cases, the average running time may be used to advantage if the probability of missing the deadline due to excessive voting delays is comparable to other sources of failure (e.g., resource exhaustion or imperfect coverage).

References

[AGRA88] Agrawal, P., "Fault Tolerance in Multiprocessor Systems Without Dedicated Redundancy", *IEEE Transactions on Computers*, Vol. 37, No. 3, pp. 358-362, Mar. 1988.

[AVIZ85] Avizienis, A., "The *N*-version Approach to Fault-Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, pp. 1491-1501, Dec. 1985.

[BABA87] Babaoglu, O., "On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems", *ACM Transactions on Computer Systems*, Vol. 5, pp. 394-416, 1987.

[BARB87] Barbara, D. and H. Garcia-Molina, "The Reliability of Voting Mechanisms", *IEEE Transactions on Computers*, Vol. C-36, No. 10, pp. 1197-1208, Oct. 1987.

[BRIL87] Brilliant, S.S., J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in *N*-version Software", *Software Engineering Notes*, ACM SIGSOFT, Vol. 12, pp. 29-34, Jan. 1987.

[BLOU90] Blough, D.M. and G.F. Sullivan, "A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems", *Proc. of the 9th Symp. on Reliable Distributed Systems*, Oct. 1990.

[BROW84] Brownrigg, D.R.K., "The Weighted Median Filter", *Communications of the ACM*, Vol. 27, No. 8, pp. 807-818, Aug. 1984.

[CAMP91] Campbell D. and T. McNeill, "Finding a Majority When Sorting is Not Available", *The Computer Journal*, Vol. 34, No. 2, p. 186, Apr. 1991.

[CHEN90] Chen, Y. and T. Chen, "Implementing Fault Tolerance via Modular Redundancy with Comparison", *IEEE Transactions on Reliability*, Vol. 39, pp. 217-225, June 1990.

[DAVI62] David F.N. and D.E. Barton, *Combinatorial Chance*, Hafner, 1962, p. 315.

[GARC85] Garcia-Molina, H. and D. Barbara, "How to Assign Votes in a Distributed System", *Journal of the ACM*, Vol. 32, pp. 841-860, Oct. 1985.

[GIFF79] Gifford, D.K., "Weighted Voting for Replicated Data", *Proc. of the 7th ACM SIGOPS Symp. on Operating System Principles* (Pacific Grove, CA), Dec. 1979, pp 150-159.

[JAJO90] Jajodia, S. and D. Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database", *ACM Transactions on Database Systems*, Vol. 15, pp. 230-280, June 1990.

[JOHN89] Johnson, B.W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.

[KANE89] Kanekawa, N., H. Maejima, H. Kato, and H. Ihara, "Dependable Onboard Computer Systems with a New Method — Stepwise Negotiating Voting", *Proc. of the International Symp. on Fault-Tolerant Computing* (Chicago), June 1989, pp 13-19.

[KNIG86] Knight, J.C. and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transactions on Software Engineering*, Vol. SE-12, pp. 96-109, Jan. 1986.

[KNUT73] Knuth, D.E., *The Art of Computer Programming — Vol. 1: Fundamental Algorithms*, Addison-Wesley, 2nd Edition, 1973. (Subsection 1.2.6, Problem 64, p. 73 and p. 489).

[KOHA78] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, 2nd Edition, 1978, pp. 333-347.

[LORC89] Lorczak, P.R., A.K. Caglayan, and D.E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems", *Proc. of the International Symp. on Fault-Tolerant Computing* (Chicago), June 1989, pp. 444-451.

[PARH91] Parhami, B., "Voting Networks", *IEEE Transactions on Reliability*, Vol. 40, No. 3, pp. 380-394, Aug. 1991.

[PARH91a] Parhami, B., "The Parallel Complexity of Weighted Voting", *Proc. of the International Conf. on Parallel and Distributed Systems*, Washington, DC, Oct. 1991, pp. 382-385.

[PARI86] Paris, J.-F., "Voting with a Variable Number of Copies", *Proc. of the International Symp. on Fault-Tolerant Computing* (Vienna, Austria), July 1986, pp. 50-55.

[SIEW82] Siewiorek, D.P. and R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982 (discussion on voting, pp 117-122).

[SKLA76] Sklaroff, J.R., "Redundancy Management Techniques for Space Shuttle Computers", *IBM Journal of Research and Development*, Vol. 20, pp. 20-28, Jan. 1976.

[SUSY91] Su, S.Y.H., M. Cutler, and M. Wang, "Self-Diagnosis of Failures in VLSI Tree Array Processors", *IEEE Transactions on Computers*, Vol. 40, No. 11, pp. 1252-1257, Nov. 1991.

[VOGE88] Voges, U., "Use of Diversity in Experimental Reactor Safety Systems", in *Software Diversity in Computerized Control Systems*, Springer-Verlag, 1988, pp. 29-49.