Chapter 5

MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.

[1]The work described here was conducted in the Computer Science Laboratory of SRI International and suported in part by NASA Contract NAS1-15528, NSF Grant MCS-7904081, and ONR Contract N00014-75-C-0816 1981. A brief history of this work is given in the concluding section.

5.1 Introduction

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

If the votes can be simply ordered, an algorithm with order n execution time can be coded first to find the median using the Rivest-Tarjan algorithm [7] and then to check whether the median received more than half the votes. The Rivest-Tarjan algorithm is bounded above by $5.43n - 163$ comparisons, when $n > 32$.

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

5.2 The Algorithm

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

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

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

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

Here is a proof by simple induction on i that the delegates polled may always be divided into two such groups after the chairman has processed the first i delegates. After the chairman has processed the first delegate, k and i are both 1: the group of delegates passed has 1 vote for *cand*. So suppose the invariant holds after the i^{th} candidate, and suppose the i delegates processed so far may be divided into two groups, U and P, with the aforementioned properties. If after processing the i^{th} delegate k is 0, then *cand* is reset to the candidate preferred by the $i+1^{\text{st}}$ delegate and k is set to 1. But when k is 0 the invariant tells us that P contains all the first i delegates. Thus the first $i+1$ delegates may be divided into two groups: one containing only the $i+1^{\text{st}}$ delegate and one that is P. If after processing the i^{th} delegate k is not 0, there are two cases: the $i+1^{\text{st}}$ delegate votes for or against *cand*. If the $i+1^{\text{st}}$ delegate votes for *cand*, k is incremented; the first $i+1$ delegates may be divided into two groups: U plus the $i+1^{\text{st}}$ delegate and P. If the $i+1^{\text{st}}$ delegate votes against *cand*, k is decremented; the first $i+1$ delegates may be divided into two groups as follows. Let j be any one of the delegates in U. Let the first group be U minus j, and let the second group be P together with both j and the $i+1^{\text{st}}$ delegate.

5.3 Examples

Suppose there are three candidates, `A`, `B`, and `C`, and suppose that the delegates are polled by the chairman in the following order:

```
A A A C C B B C C C B C C.
```

After the chairman has visited the third delegate, candidate `A` is leading with 3 votes. In the diagram below we put a vertical bar after the third delegate to indicate the position of the chairman.

```
                                          CAND     K
A A A|C C B B C C C B C C                   A      3
```

In processing the next three delegates, the chairman pairs off the three `A` votes against three other votes (two for `C` and one for `B`). After the sixth delegate has been visited, k is 0 and the vote of the seventh delegate makes `B` the leading candidate.

```
                                          CAND     K
A A A C C B B|C C C B C C                   B      1
```

The next delegate, however, cancels out `B`'s short-lived ascendancy and the next two delegates give `C` the lead by two votes.

```
                                          CAND     K
A A A C C B B C C C|B C C                   C      2
```

The next delegate diminishes `C`'s lead by one, but the last two raise it to 3 by the time the pairing phase terminates. The claim is that if any candidate has a majority, it is `C`.

Here is a simple example of the final state of the pairing phase on a ballot in which no candidate has a majority:

```
                                          CAND     K
A A A B B B C|                              C      1
```

The votes for `A` and `B` cancel one another out and `C` wins the pairing phase by default. Had the delegates been polled in a different order, `A` or `B` might have won.

5.4 The Fortran Implementation

Suppose the delegates are in an array `A` of length `N`. The subroutine `MJRTY` shown in Figure 5.1 takes `A` and `N` as input and sets `BOOLE` and `CAND` to communicate the results. `BOOLE` will be set either to `.TRUE.` or to `.FALSE.`. If `BOOLE` is set to `.TRUE.`, there is one (and only one) majority element in `A` and `CAND` is set to that element. If `BOOLE` is set to `.FALSE.`, there is no majority element in `A`.

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

```
      SUBROUTINE MJRTY(A, N, BOOLE, CAND)
      INTEGER N
      INTEGER A
      LOGICAL BOOLE
      INTEGER CAND
      INTEGER I
      INTEGER K
      DIMENSION A(N)
      K = 0
C     THE FOLLOWING DO IMPLEMENTS THE PAIRING PHASE. CAND IS
C     THE CURRENTLY LEADING CANDIDATE AND K IS THE NUMBER OF
C     UNPAIRED VOTES FOR CAND.
      DO 100 I = 1, N
      IF ((K .EQ. 0)) GOTO 50
      IF ((CAND .EQ. A(I))) GOTO 75
      K = (K - 1)
      GOTO 100
50    CAND = A(I)
      K = 1
      GOTO 100
75    K = (K + 1)
100   CONTINUE
      IF ((K .EQ. 0)) GOTO 300
      BOOLE = .TRUE.
      IF ((K .GT. (N / 2))) RETURN
C     WE NOW ENTER THE COUNTING PHASE. BOOLE IS SET TO TRUE
C     IN ANTICIPATION OF FINDING CAND IN THE MAJORITY. K IS
C     USED AS THE RUNNING TALLY FOR CAND. WE EXIT AS SOON
C     AS K EXCEEDS N/2.
      K = 0
      DO 200 I = 1, N
      IF ((CAND .NE. A(I))) GOTO 200
      K = (K + 1)
      IF ((K .GT. (N / 2))) RETURN
200   CONTINUE
300   BOOLE = .FALSE.
      RETURN
      END
```

Figure 5.1: Fortran Implementation of the Algorithm

In some applications it may be assumed that a majority candidate exists. For example, in the SIFT aircraft control system [11], where reliability is achieved with redundant processors and software voting, the failure rate of the individual processors is sufficiently low to permit the assumption that on a given flight a majority of the working processors agree on each vote. If it can be assumed that a majority exists, the counting phase may be eliminated. More importantly, the algorithm can then be implemented to poll the delegates in real time (rather than store the votes for batch processing).

The Fortran code above contains one minor improvement not mentioned in the convention floor-fight analogy. After the pairing phase has terminated, we test k against $n/2$. If k is greater than $n/2$, we announce that *cand* is the majority candidate without bothering with the counting phase, because we know there are at least k votes for *cand*. Indeed, one could make such a test every time k is incremented in the first loop. This would sometimes allow the algorithm to avoid making the second pass through the votes. Whether the running time of the algorithm is improved when such a test is inside the loop depends upon the distribution of the votes.

We have failed to find a variation of the first phase of the algorithm that obviates the second phase.

5.5 The Fortran Verification System

The informal proof sketched above may be convincing evidence that the algorithm computes the majority element if one exists. Of more practical importance, however, is whether the Fortran code implements the algorithm correctly and executes without error on all Fortran processors. There are many potential sources of error in the code that are completely ignored by the "proof" above. Is the program really a legal ANSI Fortran program? Does it violate any of the rules about aliasing and second level definition? Have we correctly analyzed the flow of control? Have we considered all the possibilities at run time? For example, ANSI Fortran permits individual elements of an array to be "undefined" (e.g., uninitialized). In such cases, even the meaning of an equality test is left unspecified by ANSI. A more obvious run time worry is that n might be so large that one of the arithmetic operations causes an overflow. Furthermore, the proofs are very informal. Are they correct? Have cases been ignored? Have false or unwarranted properties about "unanimity" and "majority" been assumed?

To permit the reliable verification of many Fortran programs we have implemented a mechanical verification system for Fortran. That system has been used to verify `MJRTY` and other subprograms. Before presenting the formal specifications that were verified, we briefly sketch our verification system.

The system handles a subset of both ANSI Fortran 66 [10] and ANSI Fortran 77 [1]. The subset is described precisely in [4]. Informally stated, the subset includes all the statements of Fortran 66 except the i/o, `DATA`, `BLOCK DATA`, and `EQUIVALENCE` statements. Certain restrictions, however, are placed on some of the remaining statements. For example, we allow only named `COMMON` blocks, we prohibit `REAL` arithmetic because we do not have a machine independent semantics for floating point operations, and we require that all arithmetic statements be fully parenthesized to permit straightforward overflow analysis. (ANSI permits the compiler to associate `A+B+C` to either the left or right. The overflow analysis is different for the two cases. We therefore require the programmer to write `(A+(B+C))` or `((A+B)+C)`, which, according the ANSI standard, determines the run time association. We have implemented this requirement in a simple but conservative way: all arithmetic expressions must be fully parenthesized. Thus the code for `MJRTY` contains unnecessary parentheses, e.g., in `K=(K+1)`. A more elaborate expression grammar could eliminate the unnecessary parentheses.)

Here, expressed informally, is what we mean when we say that our system has established the "correctness" of a subprogram:

> If a Fortran subprogram is accepted and proved by our system and the program can be loaded onto a Fortran processor that meets the ANSI specification of Fortran and certain parameterized constraints on the accuracy of arithmetic, then any invocation of the program in an environment satisfying the input condition of the program will terminate without run time errors and will produce an environment satisfying the output condition of the program.

This statement is made more precise in [4].

Our Fortran verifier is a standard Floyd-King-style system [5, 6, 2, 8] consisting of two parts: a Fortran analyzer (syntax checker and verification condition generator) and a mechanical theorem-prover. For those readers unfamiliar with Floyd-King-style verification, we briefly describe our system below.

Input to the analyzer consists of the Fortran subprogram (function or subroutine) to be verified, the mathematical specification of the subprogram, and all the subprograms somehow referenced by the candidate program. Each referenced subprogram must have been previously verified by the system. A specification consists of two mathematical formulas, called the "input assertion" and the "output assertion." The first describes those states in which the program may be properly invoked. The second describes the states produced by the program. In addition to the input/output assertions, each loop in the subprogram must be cut by an inductive assertion—a mathematical formula describing the machine state each time execution arrives at the indicated point in the program. All the formulas are written in the formal logical language described in [3].

The analyzer checks that the program satisfies all our syntactic requirements and then generates mathematical formulas called "verification conditions." If these can be proved—i.e., derived symbolically from a certain set of axioms using certain rules of inference—then, whenever the program is invoked in an input state satisfying the input assertion it produces a state satisfying the output assertion.

In general, there is one such formula for each assertion-free path between any two assertions. The formula for such a path requires proving that, if the assertion at the beginning of the path is true and one is led down the path by the tests, then the assertion at the end of the path is true. In addition, formulas are generated to establish that no array bound errors, overflows, or other run time errors occur, and that the program terminates. (See [4].)

To permit consideration of arithmetic overflow, our verification system permits formal talk about the "least inexpressible positive integer" and the "greatest inexpressible negative integer" on the host Fortran processor. Typical input assertions for programs must specify the relations between the input variables and these otherwise unspecified constants. We assume that ANSI Fortran processors compute the correct results and cause no arithmetic overflow on primitive `INTEGER` arithmetic operations (i.e., `+`, `-`, `*`, `/`, and `**`) in which the inputs and the mathematically defined result are all strictly between the least and greatest inexpressible integers.[2]

The second part of the verification system is a mechanical theorem-prover that attempts to prove the formulas generated by the analyzer. The theorem-prover, which is described in [3], is entirely responsible for the correctness of each proof.

5.6 Formal Specification

Below we denote the initial values of the Fortran variables `A` and `N` by A and n respectively. The precise input assertion for `MJRTY` is that n is a positive integer, that $n+1$ is strictly less than the least inexpressible positive integer, and that every element of A is defined. The reason $n+1$, rather than merely n, must be expressible is that the ANSI standard permits `I` to obtain the value $n+1$ immediately before the termination of `DO 100 I = 1, N`.

The output assertion for `MJRTY` is

- The final value of `BOOLE` is `.TRUE.` or `.FALSE.` (that is, `BOOLE` may not be returned "undefined").
- The elements of `A` are not changed.
- If `BOOLE` is set to `.TRUE.`, then `CAND` is defined and the number of occurrences in A of its final value, *cand*, is more than $n/2$. (By "/" we denote the integer "floor" of the real quotient.)

[2]In addition, for division we require that the denominator be nonzero.

- If `BOOLE` is set to `.FALSE.`, then for all x, the number of times x occurs in A is less than or equal to $n/2$.

We phrase these requirements in terms of the mathematical function cnt of four arguments. One may read $\mathrm{cnt}(x, A, i, j)$ as "the number of times x occurs in A from i through j inclusive." The function cnt is a typical example of a concept that must be introduced into one's underlying logical theory to specify a program. We may define cnt recursively for all $i \geq 0$ and $j \geq 0$ as follows:

$$\mathrm{cnt}(x, A, i, j) = \begin{cases} 0 & \text{if } j = 0 \lor j < i \\ 1 + \mathrm{cnt}(x, A, i, j-1) & \text{if } 0 < j \not< i \land x = A(j) \\ \mathrm{cnt}(x, A, i, j-1) & \text{otherwise} \end{cases}$$

Our mechanical theorem-prover verifies that there exists a function satisfying the above equation before the equation is added as a new axiom. Without such a check, the user of a verification system might inadvertently "overspecify" a concept and permit correctness proofs based on contradictions in the underlying specification.

We cut the first `DO`-loop in `MJRTY` with an invariant at the bottom of the loop, just before `I` is incremented and tested against `N`. In our informal proof the invariant required that the i delegates processed thus far could be divided into a unanimous group for *cand* of size k and a group that could be paired into disagreeing delegates. Since the algorithm does not explicitly keep track of any such division of the delegates, we reformulated the invariant in a slightly weaker fashion. The reformulation is based on the observation that, if a collection of delegates can be paired in such a way that paired delegates disagree, then the collection has no majority.[3]

Here is the actual invariant used:

(1) $0 < i$ and $0 \leq k \leq i \leq n$.

(2) `CAND` is always defined and has some value *cand*.

(3) The number of times *cand* occurs in A from 1 through i is at least k.

(4) The number of times *cand* occurs in A from 1 through i, minus k, is no greater than $(i - k)/2$.

(5) For all x other than *cand*, the number of times x occurs in A from 1 through i is no greater than $(i - k)/2$.

Although conjuncts (1) and (2) were ignored in our informal proof, they are essential in a careful proof. Conjunct (3) establishes that we have at least k votes for *cand*. Let those k delegates constitute the "unanimous group." The $i - k$ remaining delegates are the "majority-free group." Conjunct (4)

[3] The converse also holds for collections with an even number of members.

says that *cand* does not have a majority in the majority-free group; ignoring the k votes in the unanimous group, the number of votes for *cand* thus far encountered is less than $(i-k)/2$. Conjunct (5) says that no other candidate has a majority in the majority-free group. We count the votes for candidates other than *cand* over the entire interval processed, rather than just over the majority-free group, since we do not really know where the majority-free group is. But we know that the unanimous group contributes nothing to the tally of a candidate other than *cand*. (It is easy to see by the construction of a counterexample that (4) and (5) do not imply (3). Nevertheless, if one modifies the code so that k is not tested against $n/2$ before entering the counting phase, one can omit conjunct (3) of this invariant. That is, unless the program exits early when k exceeds $n/2$, a demon within the first loop is permitted to raise k above the count of *cand* (within the constraint imposed by (5)) without causing the algorithm to perform incorrectly. We do not know how to interpret this lack of constraint.)

As the counting phase is trivial, we shall not discuss it.

5.7 The Formal Proofs

The Fortran analyzer produced 61 verification conditions for `MJRTY`. Most of the conjectures established that array bounds are not violated, that arithmetic operations cause no overflows, and that variables and array elements are defined when required.

The mechanical theorem-prover proved all 61 conjectures. Most of the proofs were immediate either from the axioms and definitions in the "basic Fortran theory" [4] (e.g., the definition of the negative integers in terms of the Peano numbers), from the definition of cnt (e.g., if x is $A(i+1)$ and $i \geq 0$ then $\mathrm{cnt}(x, A, 1, i+1)$ is $1 + \mathrm{cnt}(x, A, 1, i)$), or from elementary arithmetic lemmas (e.g., the theorem that for all naturals m and n, $n/2 < m$ iff $n < 2m$). Several of the paths to the invariants and to the output assertion required that the user help the system.

The user of our system can help the system prove a "hard" theorem by suggesting that it first prove some key lemmas. When the system proves a theorem for the user, it stores the theorem for use in future proofs. Thus, by bringing to the theorem-prover's attention previously unrecognized truths, the well-trained user of our system can get the theorem-prover to prove formulas that would otherwise be beyond the system's competence. The user of our system, however, does not have to be trusted. The machine—not the human—is responsible for the validity of the final proof; the user cannot maliciously or inadvertently cause the system to accept falsehoods, because the system proves for itself every fact used.

To get all 61 theorems proved, we had to instruct the theorem-prover to prove five lemmas about cnt. The two most interesting ones were as follows:

- cnt is monotonic: the number of times x occurs from 1 through i is less than or equal to the number of times it occurs from 1 through j if $0 \leq i \leq j$. Without knowing this, the theorem-prover could not approve our exiting from the counting phase as soon as k exceeds $n/2$ lest subsequent processing of the remaining delegates decrease k.

- The number of times x occurs from 1 through i ($i \geq 0$) is no greater than i. This ensures that k in the second loop will never exceed i (and thus incrementing k will never cause an overflow).

These two lemmas are proved by the system with mathematical induction on the length of the interval scanned.

The other three lemmas we proved were required because of inadequacies in the theorem-prover itself. For example, when `MJRTY` exits because k is 0 at the end of the counting phase, the theorem-prover knows that *cand* has no majority and that no x other than *cand* has a majority. It must prove that no x has a majority. The proof is obvious if one merely asks, "Is x equal to *cand* or not?" and considers the two cases. Without an explicit theorem stated by the user, the theorem-prover failed to consider such a case split. The other two lemmas were necessary for similar reasons and indicate inadequacies in our system that we hope to repair in the future.

The entire effort of specifying `MJRTY` and getting the 61 verification conditions proved required about 20 man hours. Most of the time was spent identifying problems caused by incorrectly written invariants, overcoming inadequacies in the theorem-prover by identifying appropriate lemmas, and struggling with the still awkward interface to our Fortran verification condition generator. It requires about 55 minutes of computer time to prove the final list of 66 theorems. The time was measured on a Foonly F2 Computer (about 30 percent as fast as a DEC 2060) running small INTERLISP-10. A total of 42 minutes was required for theorem-proving, 8 minutes for garbage collection, and 5 minutes for printing out the proofs.

Readers interested in obtaining the system's complete English description of its proofs may contact the authors.

5.8 Postscript 1991

In this paper we have described a linear time majority vote algorithm and discussed the mechanically checked correctness proof of a Fortran implementation of it. This work has a rather convoluted history which we would here like to clarify.

The algorithm described here was invented in 1980 while we worked at SRI International. A colleague at SRI, working on fault tolerance, was trying to specify some algorithms using the logic supported by "Boyer-Moore Theorem Prover." He asked us for an elegant definition within that logic of the notion of the majority element of a list. Our answer to this challenge was the recursive expression of the algorithm described here.

In late 1980, we wrote a Fortran version of the algorithm and proved it correct mechanically. In February, 1981, we wrote this paper, describing that work. In our minds the paper was noteworthy because it simultaneously announced an interesting new algorithm and offered a mechanically checked correctness proof. We submitted the paper for publication.

In 1981 we moved to the University of Texas. Jay Misra, a colleague at UT, heard our presentation of the algorithm to an NSF site-visit team. According to Misra (private communication, 1990): "I wondered how to generalize [the algorithm] to detect elements that occur more than n/k times, for all k, $k \geq 2$. I developed algorithm 2 [given in Section 3 of [9]] which is directly inspired by your algorithm. Also, I showed that this algorithm is optimal [Section 5, *op. cit.*]. On a visit to Cornell, I showed all this to David Gries; he was inspired enough to contribute algorithm 1 [Section 2, *op. cit.*]." In 1982, Misra and Gries published their work [9], citing our technical report appropriately as "submitted for publication."

However, our paper was repeatedly rejected for publication, largely because of its emphasis on Fortran and mechanical verification. A rewritten version emphasizing the algorithm itself was rejected on the grounds that the work was superceded by the paper of Misra and Gries!

When we were invited to contribute to the Bledsoe festschrift we decided to use the opportunity to put our original paper into the literature. We still think of this as a minor landmark in the development of formal verification and automated reasoning: here for the first time a new algorithm is presented along with its mechanically checked correctness proof—eleven years after the work.

References

[1] American National Standards Institute, Inc. (1978): American National Standard Programming Language Fortran, ANSI X3.9–1978, 1430 Broadway, New York, N. Y.

[2] R. B. Anderson (1979): Proving Programs Correct. New York: John Wiley and Sons.

[3] R. S. Boyer and J S. Moore (1979): A Computational Logic. New York: Academic Press.

[4] R. S. Boyer and J S. Moore (1981): A Verification Condition Generator for Fortran. In R. S. Boyer and J S. Moore, eds.: The Correctness Problem in Computer Science. London: Academic Press.

[5] R. W. Floyd (1967): Assigning Meanings to Programs. In: Mathematical Aspects of Computer Science, Proc. Symp. Appl. Math., 19. Providence, RI: American Mathematical Society.

[6] J. C. King (1969): A Program Verifier. Ph. D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[7] D. E. Knuth (1973): The Art of Computer Programming, Volume 3: Sorting and Searching. Reading, MA: Addison-Wesley.

[8] Z. Manna (1974): Mathematical Theory of Computation. New York: McGraw-Hill.

[9] J. Misra and D. Gries (1982): Finding Repeated Elements. Science of Computer Programming 2, 143–152.

[10] United States of America Standards Institute (1966): USA Standard Fortran, USAS X3.9–1966, 10 East 40th Street, New York, N. Y.

[11] J. Wensley, et al. (1978): SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control. Proc. IEEE 66(10), 1240–1255.