Robustness versus Performance in Sorting and Tournament Algorithms

Wilfried Elmenreich, Tobias Ibounig, István Fehérvári

Mobile Systems/Lakeside Labs
University of Klagenfurt, Austria
wilfried.elmenreich@uni-klu.ac.at
ibounig@lakeside-labs.com
istvan.fehervari@uni-klu.ac.at

Abstract: In this paper we analyze the robustness of sorting and tournament algorithms against faulty comparisons. Sorting algorithms are differently affected by faulty comparisons depending on how comparison errors can affect the overall result. In general, there exists a tradeoff between the number of comparisons and the accuracy of the result, but some algorithms like Merge Sort are Pareto-dominant over others. For applications, where the accuracy of the top rankings is of higher importance than the lower rankings, tournament algorithms such as the Swiss System are an option. Additionally, we propose a new tournament algorithm named Iterated Knockout Systems which is less exact but more efficient than the Swiss Systems.

Keywords: sorting algorithms, robustness, tournaments, iterated knockout system

1 Introduction

Sorting is a fundamental and often applied algorithm in computer science. There has been put much attention on the efficiency of a sorting algorithm in terms of number of comparisons or number of element switches. In some applications, like sports tournaments or comparative evolutionary algorithms, the comparison function is a complex function that involves either a match between two players or a simulation of two teams trying to achieve a given goal [1]. In such cases, especially when the opponents are of similar strenght, the outcome of a comparison can become indeterministic (e.g., due to a sports team winning over a stronger team by being lucky).

Our motivation is thus to research sorting algorithms for these potentially failing comparisons. We expect to have faulty comparisons due to random fluctuations in the evaluation function that compares two elements. With respect to randomly occurring errors, we do not assume a hard limit on the number of occurring faulty

comparisons. Instead we are looking for a method that is *robust* against such problems, that is we allow a deviation from the perfect result, but the result should be gracefully degrading based on the number of faults.

The related work on this topic goes back to the 1960s (Section 2) and shows that the problem has been identified in different fields such as mathematics, computer science, and organizers of social studies or (chess) tournaments.

In our work we examine the robustness of typical sorting and tournament algorithms with respect to faulty comparisons. A key hypothesis was that a very efficient (i.e., low complexity order) sorting algorithm might be more susceptible to errors from imprecise comparisons than the more inefficient sorting algorithms which might implement a lot of implicitly redundant comparisons. While our results from an experimental validation of several standard sorting algorithms in general support this hypothesis, there are still some intrinsic factors in the way of sorting that make an algorithm more or less robust to these faults. We show that therea is a tradeoff between accuracy and number of comparisons and place the results for Bubble Sort, Selection Sort, Heap Sort, Quick Sort, Merge Sort and Insertion Sort on a two-dimensional map of both criteria. As shown in Secion 6, Merge Sort provides a good accuracy for a reasonable comparion overhead. Additionally, we have examined tournament systems such as Round Robin, Swiss System and propose an Iterated Knockout System (IKOS). These algorithms are especially of interest for sorting tasks where the accuracy of the topmost places is the most important while errors in the lower ranks do not play a role. For this case, IKOS has shown the highest efficiency.

The insights gained from this work (Section 7) may be a helping guideline for selecting a sorting algorithm under noisy conditions. In particular they are useful for implementing a fair but time-efficient tournament that determines the best teams. Another application can be in evolutionary algorithms with comparative fitness functions, as for example in [1]. Since the fitness comparison often requires a time-consuming simulation, cutting down on the number of comparisons (i.e., simulation runs) while keeping the accuracy for the upper part of the population is an important issue.

2 Related Work

There exists a vast amount of literature on sorting algorithms [2, 3]. In the following we review work where the problem on robustness and fault tolerance is particularly treated.

Binary search with faulty information was formulated as game theoretic problem by Rényi [4] (a player must guess an object based on yes/no answers from another player that sometimes may answer incorrectly) and by Ulam [5] (a very similar

game where a player must guess a number in a given range). A more comprehensive overview on the historical development of research on this topic can be found in [6].

Approximate voting is also supported by several algorithms supporting anytime behavior, i.e., a process generations intermediate results which increase in their accuracy over time. An example for such an algorithm is Comb Sort [7], which iteratively "combs" the elements similar to a bubble-sort approach. However, the comb sort provides adjustment for specifically focussing on a specific part (e.g. the first few positions) of the list to be sorted.

Ravikumar, Ganesan, and Lakshmanan discuss the problem of finding the largest element of a set using imperfect comparisons [8]. The approach is extended in [9] to an algorithm for sorting elements with a comparison function that may sometimes fail, i.e., yielding the incorrect results. The algorithm has a worst-case complexity of $\Omega(n\,log(n) + e\,n)$, where e is an upper bound for the total number of errors. Based on these results, Long [10] presents an algorithm for searching and sorting with a faulty comparison oracle. Given that the assumption on e does hold, these algorithms provide a perfect ranking. However, there is no assessment on the sorting quality if this assumption is invalidated. Thus, these algorithms are fault-tolerant, but not necessarily robust.

Bagchi presents a similar approach in [11]. His fault-tolerant algorithm is basically a binary insertion sort modified to cope with errors also with a worst-case complexity of $\Omega(n\,log(n) + e\,n)$.

Ajtai et al. [12] assume a different model for imprecise comparisons, where the outcome of a comparison is considered unpredictable if the elements differ by less than a given threshold δ. They present an algorithm that provides a correct sorting of all elements which differ at least by δ.

Giesen et al. [13] present a worst-case bound for the necessary comparisons of any approximate sorting algorithm (however without cosidering faulty comparisons) that ranks n items within an expected Spearman's Footrule distance.

3 Why Robustness

In contrast to the well-established and well-defined field of fault tolarence [14, 15], the notion of robustness differs by the field of research [16]: *"A biologist will understand robustness in terms like adaptation, stability, diversity, survivability, and perturbations. A control theorist will express robustness in terms of uncertainty of mathematical models. A software developer might focus on a programs ability to deal with unusual usage or users input."*

Robustness differs from fault tolerance in the way that robustness is not implemented against a rigid fault hypothesis [17].

Instead, robustness (against a particular property, such as noisy sensor data, etc.) points out that the system is capable of maintaining its function (at least in a degraded, but acceptable way) despite various unexpected perturbations.

When applying the concept of robustness for sorting, we are looking for algorithms that might degrade in its result, i.e., the sorting order, but provide an approximate result which is acceptable. Although we can use standard measures to define if a result is more or less deviating from the correct sorting, the level of acceptability heavily depends on the application. The application we had in mind was performing a sorting of candidates in a genetic algorithm where the fitness function is inaccurate [1]. In genetic algorithms, an inaccurate sorting is likely to have only a degrading effect on the runtime of the algorithm, i.e., slowing down the convergence of the gene pool towards a solution with high fitness. Hence, there exist mutual tradeoffs between sorting speed/sorting accuracy and sorting accuracy/speed of genetic algorithms.

For the sake of generality, we will analyze several sorting approaches yielding different combinations of sorting performance and accuracy.

4 Algorithms under Consideration

4.1 Sorting Algorithms

As a first step we will evaluate a number of standard sorting algorithms. We have selected Bubble Sort[1], Selection Sort[2], Insertion Sort[3], Heap Sort[4], Quick Sort (using a simple randomized function to define the pivot) [5], and Merge Sort[6] for our test. The first three sorting algorithms are in the complexity order of $O(n^2)$, i.e., they are typically very inefficient for a high number of elements. The other three algorithms are in the complexity order $O(n \log n)$, thus more efficient.

The majority of the sorting algorithms considered in this paper are symmetric towards sorting the whole set in similar quality. An exception is the Heap Sort

1 http://en.wikipedia.org/wiki/Quicksort
2 http://en.wikipedia.org/wiki/Selection_sort
3 http://de.wikipedia.org/wiki/Insertionsort
4 http://en.wikipedia.org/wiki/Heapsort
5 http://en.wikipedia.org/wiki/Quicksort
6 http://en.wikipedia.org/wiki/Merge_sort

algorithm and the Swiss system. Under the presence of faulty comparions, Heap Sort turned out to make more sorting errors in the top ranks rather than in the last ranks. Therefore, we inverted the Heap Sort algorithm in order to have the better sorting in the top ranks.

4.2 Tournament Algorithms

The sorting problem is very similar to the task of organizing a tournament among a number of participants. Participants are paired into matches deciding which participant is stronger and should therefore be sorted "above" the other one.

In contrast to sorting, tournament organizers usually consider that comparisons are neither deterministic nor consistently yielding the stronger participant. In the simplest form, the round robin tournament, every participant is paired against every other participant. The results of each match give points to the participants, which are sorted according to their points in the end. Note that for example, a participant could win a tournament even though he or she lost to the second ranked participant.

A round-robin approach is usually very robust against random influences on the comparison function, since the pairing does not depend on the outcome of previous comparisons. However, this approach requires $\frac{n(n-1)}{2}$ comparisons and is thus as inefficient as the sorting algorithms in the complexity order of $O(n^2)$.

The Swiss Systems style tournament [18] is more efficient than the round robin tournament. The Swiss System is extensively used in chess tournaments. When there is no *a priori* knowledge of the participants' strenght, the first round of s Swiss System tournament contains random pairings. In each game the winner gets two points, loser gets zero, in case of a draw both get one point. After this round players are placed in groups according to their score (winners in the group "2", those who drew go in the group "1" and losers go into the group "0"). The aim is to ensure that players with the same score are paired against each other. Since the number of perfect scores is cut in half each round it does not take long until there is only one player remaining with a perfect score. In chess tournaments there are usually many draws, so more players can be handled (a 5 round event can usually determine a clear winner for a section of at least 40 players, possible more).

The drawback of the Swiss system is that it is only designed to determine a clear winner in just a few rounds. Likewise, the worst performing participant is also determined. The more a position differs from the first or last position, the less likely this position is correctly ranked. In other words, the Swiss system has an increasing exactness towards the first few and last few ranks.

4.3 Iterated Knockout System

Some of the applications we had in mind only need a sorting of top half of the elements (e.g., a genetic algorithm that drops all candidates below a threshold). Therefore, we developed a specific algorithm to sort a given number of ranks starting from the "first place":

1 start with an empty ranking list;

2 start a single-elimination tournament: each candidate takes place in exactly one pairing per round. The winners of each pairing promote to the next round. If the number of candidates is uneven, one candidate not being paired passes on to the next round.

3 iterate 2 until there is only one candidate (the winner of this tournament) left;

4 append the winner to the overall ranking list;

5 build the list of candidates (except the ones already ranked) that have not lost to anyone except for the already ranked candidates;

6 go to step 2. Results from already played pairings are kept.

Thus, we subsequently pick players from the list until the ranking list contains all the ranks of interest.

5 Evaluation Method

For an evaluation, we test these algorithms on a set with randomly generated numbers in a range between 0 and 100. Array sizes have been varied between 10 and 200 according to typical target applications. For each comparison operation, a random factor (the "noise") is applied to both values before the comparison operation is performed. A 5% noise means for example that the value used for comparison may vary up to ±5% of the value range (100). Thus, the probability that a comparison may yield an incorrect result is the higher the closer the two values are. The average result of a sorting algorithm under test is compared to the correct sorting, that is without applying the random fluctuation before comparison. For the comparison we apply two metrics: The first one is based on Spearman's footrule as a measure of disarray [19], which is calculated as the sum of absolute differences between the resulting and correct ranks. The results are normalized by the number of elements, thus, the deviation in our results always gives the average distance in ranks of an element to its correct position. In order to account for applications where the correct ranking of the lower ranks is not important, we apply also a different metric that apply a weight of 2 for deviations in the top half, while ranking deviations in the lower half have a weight of zero, thus do not contribute to the metric.

6 Results

Figure 1 visualizes the complexity of the different algorithms with respect to the number of comparions. As expected, we can observe the inefficient ($O(n^2)$) sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort to require by far the most comparisons to create a ranking. The sorting algorithms Heap Sort, Merge Sort, and Quick Sort are more efficient. The Swiss System needs even less number of comparisons, but the Swiss System is no sorting algorithm since it does not yield a perfect sorting of the result even with perfect comparisons.

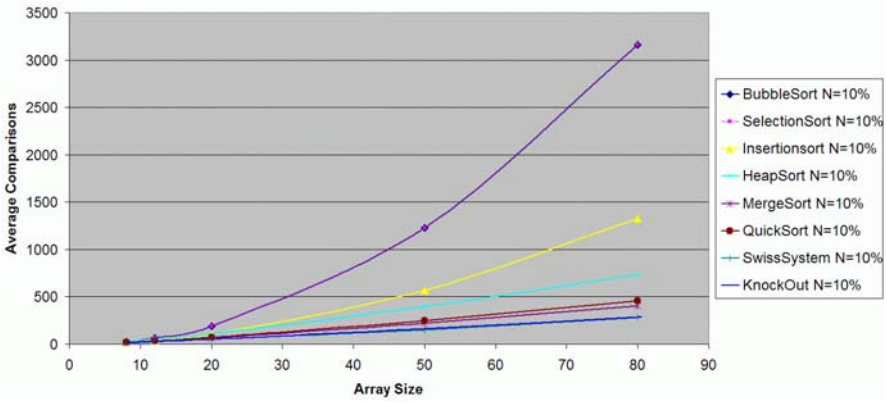

Figure 1
Number of comparisons vs. array size for different algorithms

When considering random fluctuations before comparison, the algorithms show different performances as depicted in Figure 2. The Swiss System, which was the most efficient one in the previous analysis, comes with the cost of hight deviation (disarray according to Spearman's footrule). In this graph, also the performance of a full Round Robin tournament is depicted. In the round robin tournament there are $\frac{n(n-1)}{2}$ comparisons, where each comparison gives a point to the winner. Afterwards, the ranking is established by the number of points. Although not being very efficient, faulty comparisons in the Round Robin tournament approach are likely to cancel out to have their effect limited. Therefore, a Round Robin turns out to be the most robust (but painfully slow) approach. Interestingly, Insertion Sort is both, slow and inaccurate. This is due to the fact that one faulty comparison can affect the ranking of all other elementes and thus leads to subsequent errors.

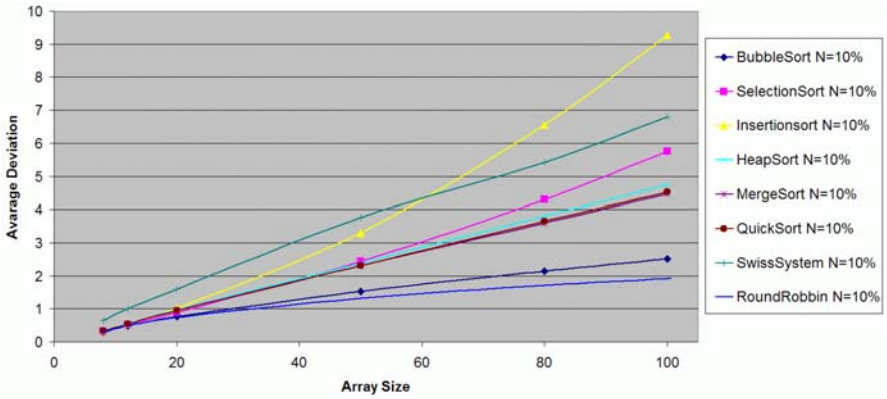

Figure 2
Deviation from the perfect result

Figure 3 examines the robustness of the sorting and tournament approaches for different levels of noise. The array size was chosen to be constant 50. The most robust methods are (sorted according to their robustness): Round Robin, Tournament, Bubble Sort, Merge Sort, Quick Sort, Heap Sort, and Selection Sort. The Swiss System is an interesting case, for low noise levels, it is among the worst methods, however, for noise of 30% and more, the Swiss System is the third best one, since its results degrade slower than the other algorithms.

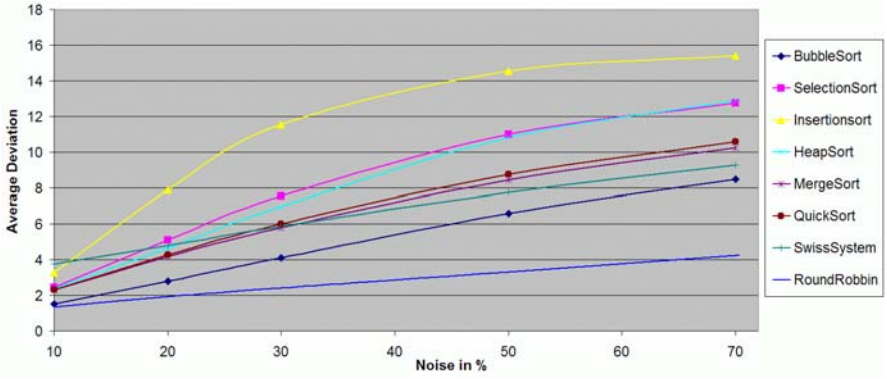

Figure 3
Vulnerability to noise in comparison function

Figure 4 maps the different algorithms according to their average number of comparisons and the average resulting deviation. The noise parameter had been chosen to be 10% and the array size was 50 for that comparison. We observe that Merge Sort dominates Quick Sort, Heap Sort, Selection Sort and Insertion Sort. In other words, Merge Sort is Pareto-optimal among this set. The Round Robin tournament dominates Bubble Sort and Selection Sort. Finally the Swiss system

dominates the IKOS approach (which was set to sort only the upper half). Thus, the algorithms of choice are Round Robin if accuracy is of most importance, Swiss Sort if efficiency is of most importance, and Merge Sort for a combination of both. Quick Sort has only slightly worse results than Merge Sort.

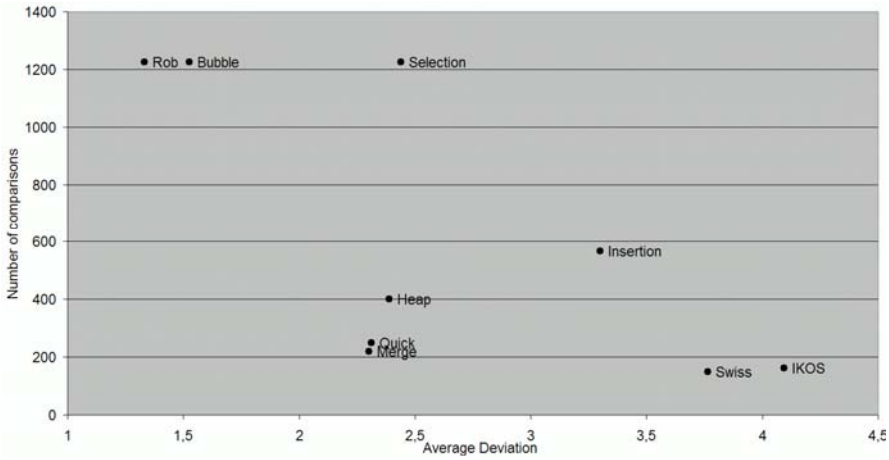

Figure 4
Mapping of different algorithms according to comparison effort and resulting deviation

Figure 5 analyzes the robustness to noise in the comparison function with respect to the top half rank results. Thus, ranking errors in the lower half do not influcence the result. Here, our proposed IKOS algorithm shows a better efficiency, since it was designed for this case.

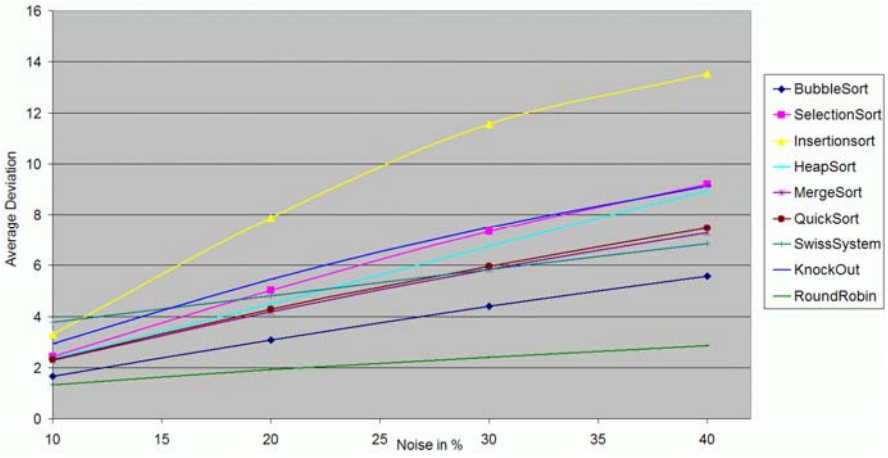

Figure 5
Robustness to noise in top half rank results

Figure 6 depicts the mapping of different algorithms according to comparison effort and resulting deviation in top half rank results. Likewise in the analysis before, the noise parameter had been chosen to be 10% and the array size was 50 for this evaluation.

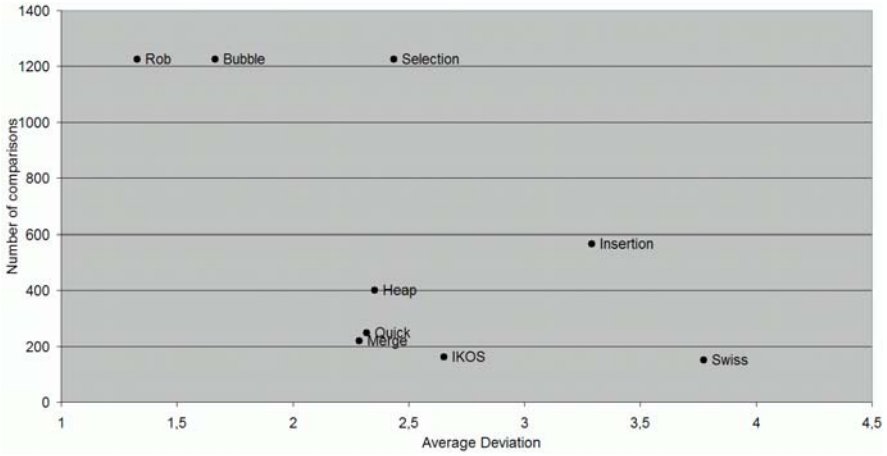

Figure 6
Mapping of different algorithms according to comparison effort and resulting deviation in top half rank results

Again, Round Robin and Merge Sort are Pareto-optimal as before. IKOS and Swiss System, however, switch places. With only slightly more comparisons than Swiss System, IKOS is able to provide a result which less prone to noisy comparions.

Conclusion

This paper contributes in two ways to the state of the art. The first contribution is the analysis of existing sorting algorithms according to their robustness against imprecise or noisy comparisons. In contrast to related work which introduces new algorithms that overcome a defined number of faulty comparisons, our approach did not aim at a fault-free sorting but rather at an approximate sorting with a minimum overhead. This is especially of interest for applications where an expensive and noisy comparison function is used to establish a ranking. If only the ranking of the first few elements is of interest, algorithms designed for (sports) tournaments are an intresting option. Apart from tournaments such a ranking function is of interest for the evaluation phase in genetic algorithms when evolving a comparative fitness function. Therefore, we have also presented a new tournament algorithm that provides an ordering incrementally starting from the top ranks.

The sorting and tournament algorithms under consideration have been evaluated according to their sorting complexity and result accuracy. The results show that

round robin tournament, merge sort, and the Swiss system are Pareto-optimal according to the overall ordering and that round robin tournament, merge sort, and the presented IKOS algorithm are Pareto-optimal according to a sorting of the top half of elements.

The presented algorithms have been selected for their best robustness, but, except for IKOS, have not been especially designed for this case. We think that there is potential for further improving the sorting algorithms by adding mechanisms dedicated to implement robustness.

Acknowledgement

This work was supported by the European Regional Development Fund and the Carinthian Economic Promotion Fund (contract KWF 20214-18128-26673) within the Lakeside Labs project DEMESOS.

References

[1] I. Fehérvári, W. Elmenreich: Evolutionary Methods in Self-Organizing System Design. In *Proceedings of the 2009 International Conference on Genetic and Evolutionary Methods*, 2009

[2] D. E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1997

[3] P. Puschner: Real-Time Performance of Sorting Algorithms. *Real-Time Systems*, 16(1):63-79, January 1999

[4] A. Rényi: On a Problem in Information Theory. *Magyar Tudományos Akadémia Matematikai Kutató Intézet Közlemény*, 6:505-516, 1961

[5] S. M. Ulam: *Adventures of a Mathematician*. Scribner, New York, 1976

[6] A. Pelc: Searching Games with Errors – Fifty Years of Coping with Liars. *Theoretical Computer Science*, 270:71-109, 2002

[7] S. Lacy, R. Box: A Fast, Easy Sort. *Byte Magazine*, p. 315 ff., April 1991

[8] B. Ravikumar, K. Ganesan, K. B. Lakshmanan: On Selecting the Largest Element in spite of Erroneous Information. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, Passau, Germany, 1987, pp. 88-99

[9] K. B. Lakshmanan, B. Ravikumar, K. Ganesan: Coping with Erroneous Information while Sorting. *IEEE Transactions on Computers*, 40(9):1081-1991, September 1991

[10] P. M. Long: Sorting and Searching with a Faulty Comparison Oracle. Technical Report UCSC-CRL-92-15, University of California at Santa Cruz, 1992

[11] A. Bagchi: On Sorting in the Presence of Erroneous Information. *Information Processing Letters*, 43(4):213-215, 1992

[12] M. Ajtai, V. Feldman, A. Hassidim, J. Nelson: Sorting and Selection with Imprecise Comparisons. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2009, Vol. Part I, pp. 37-48

[13] J. Giesen, E. Schuberth, M. Stojaković: Approximate Sorting. *Fundamenta Informaticae*, XXI:1001-1006, 1977

[14] A. Avizienis: Fault Tolerance, the Survival Attribute of Digital Systems, *Proceedings of the IEEE*, 66(10):1109-1125, October 1978

[15] J.-C. Laprie, J. Arlat, C. Béounes, K. Kanoun: Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *Computer*, 23(7):39-51, July 1990

[16] V. Mikolasek: Dependability and Robustness: State of the Art and Challenges. In *Workshop on Software Technologies for Future Dependable Distributed Systems*, Tokyo, March 2009

[17] H. Kopetz: On the Fault Hypothesis for a Safety-Critical Real-Time System. In *Keynote Speech at the Automotive Software Workshop San Diego (ASWSD 2004)*, San Diego, CA, USA, January 10-12, 2004

[18] FIDE Swiss Rules. Approved by the General Assembly of 1987. Amended by the 1988 and 1989 General Assemblies.

[19] P. Diaconis, R. L. Graham: Spearman Footrule as a Measure of Disarray. *Journal of the Royal Statistical Society*, Series B 39:262268, 1977