The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

When is A=B?*

Anja Gruünheid
ETH Zürich
ganja@inf.ethz.ch

Donald Kossmann
ETH Zürich
donaldk@ethz.ch

Besmira Nushi
ETH Zürich
nushib@inf.ethz.ch

Abstract

Most database operations such as sorting, grouping and computing joins are based on comparisons between two values. Traditional algorithms assume that machines do not make mistakes. This assumption holds in traditional computing environments; however, it does not hold in several new emerging computing environments. In this write-up, we argue the need for new resilient algorithms that take into account that the result of a comparison might be wrong. The goal is to design algorithms that have low cost (make few comparisons) yet produce high-quality results in the presence of errors.

*This write-up is based on a talk given at the University of Washington and Microsoft Research, Redmond, in August 2013. The slides of that talk are online at http://systems.ethz.ch/talks.

1 Introduction

When we think about computers, we typically assume that they are dumb and make no mistakes. Our software methodology, complexity theory, and algorithmic design are based on these two assumptions. What happens if we drop one of these assumptions? What happens if computers start making mistakes occasionally; even simple mistakes such as getting a comparison between two integers wrong? Will we need new algorithms or are the existing algorithms good enough?

There are a number of research trends that make it worthwhile to think about error-prone computers. The first trend is the emergence of crowdsourcing and the development of hybrid systems that involve machines and humans to compute tasks that neither machines nor humans are capable of computing alone. [4] gives an overview of such systems. Since these systems rely on human input, some of the computations carried out by these systems may be error-prone and algorithms that are designed for these systems need to take this fact into account.

A second trend is the development of new, low-energy processors that trade power for accuracy. That is, these processors might occasionally get an operation wrong in exchange for much lower power consumption. Examples for such designs are [1].

Third, with the advent of Big Data technologies, we are automating an increasing number of tasks based on previous experience. Recommendation systems such as those deployed by Amazon as part of their online shop improve with an increasing amount of data. Putting it differently, these systems might make poor recommendations if only little data is available.

Based on these observations, we believe that it is worthwhile to revisit existing algorithms and start thinking about how to design algorithms for computer systems that occasionally do make errors. It turns out that an algorithm that is optimal in the traditional (error-free) computational model may perform poorly in the presence of error. As an example, this paper reports on some simple observations that we made when studying QuickSort. Furthermore, this paper reports on some observations on how to group objects in a robust way if the machine occasionally misclassifies two objects. These two examples indicate that we might have to rethink complexity theory and algorithm design. As of now, the results of this paper are anecdotal, and we have not been yet able to develop a new theory. The main purpose of this paper is to raise the issue.

The remainder of this paper is organized as follows: Section 2 studies sorting. Section 3 gives an example of how errors impact algorithms for grouping or clustering objects. Section 4 contains conclusions and related work.

2 Example 1: QuickSort

To show how the presence of errors may impact algorithm design, let us start with a discussion of QuickSort. [7] gives a more general discussion of sorting algorithms in the presence of errors. Here and in the remainder of this paper, we assume a computational model in which the computer system might make an error when executing a comparison; however, the logic of the algorithm is executed correctly. Furthermore, we assume that comparisons are the most expensive operation. This computational model matches nicely hybrid systems in which comparisons are crowdsourced (e.g., [10]).

QuickSort is generally perceived as one of the best algorithms for sorting. However, what makes QuickSort great for traditional, error-free computing scenarios, hurts QuickSort in the presence of mistakes. The following example shows why. The task is to sort the following sequence of numbers:

$$7, 24, 2, 13, 51$$

Let us assume that QuickSort chooses 7 as the pivot element of the first partitioning phase and let us furthermore assume that the machine gets all comparisons right in the first partitioning, except for the comparison $7 < 51$. As a consequence, the result of the first iteration of QuickSort are the following two partitions:

$$2, 51$$

$$24, 13$$

Even if the machine is perfect and makes no further mistakes, the best possible outcome to sort the sequence of numbers is:

$$2, 51, 7, 13, 24$$

The key observation is that one wrong comparison (misclassifying $7 < 51$) resulted in three errors in the final result (misclassifying $13 < 51$ and $24 < 51$ in addition to $7 < 51$). The reason is that the QuickSort algorithm aggressively exploits the transitivity of the $<$ relation so that errors propagate. There are many different notions of error and the most appropriate definition depends on the utility function of the application. We use the number of misclassified comparisons in the final result here and in [7] because it is easy to formalize and it is a metric that is highly relevant for many applications that involve sorting or ranking data.

It turns out that it is difficult to fix QuickSort. The most natural way to improve the quality of the result is to avoid misclassifications by repeating the computation. That is, recomputing $7 < 51$ several times and then do a majority vote or accept based on a theshold. That will increase the number of comparisons by a

TransactionId	*Customer*	*Purchase*
1	Jane	$ 1000
2	Bob	$ 500
3	Jane	$ 100
4	Jane	$ 50

Table 1: Example Transactions

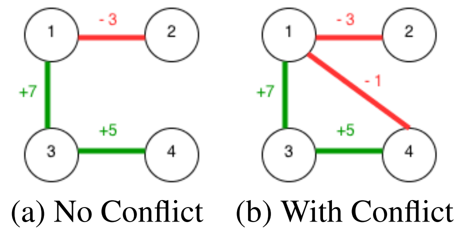
constant factor (i.e., the number of times each comparison is made) so that QuickSort continues to be in the $\mathcal{O}(n*\log(n))$ complexity class. The problem is that even with a high number of attempts, the probability of a misclassification is not zero. So, we can never expect perfection with QuickSort. Also, the impact of a wrong comparison grows with the size of the sequence in our particular error model that counts the misclassifications in the final result: In the worst case, it is $n/2$ with n the length of the sequence. The question then is how to best invest additional comparisons and whether new algorithms are more appropriate than traditional algorithms to achieve high quality for lower cost. [7], for instance, shows that iteratively running BubbleSort might be a better a way to invest additional computation for better quality. That is, do an intial sorting with QuickSort and then run BubbleSort once or several times on the result to improve the quality of the result, thereby exploiting that BubbleSort has $\mathcal{O}(n)$ complexity if the data is sorted already and the affects of wrong comparisons are always local in BubbleSort.

3 Example 2: Grouping

3.1 Vote Graphs

As a second example of how the cost/quality trade-off of error-prone computer systems impacts algorithm design, consider the list of transactions of Table 1.

The task is to compute the total purchase of each customer; i.e., $ 1150 for Jane and $ 500 for Bob. With SQL, this task can be specified using a simple GROUP BY clause. Depending on the number of customers, the number of transactions, and the skew in the distribution of transactions to customers, modern database systems choose one of three alternative ways to compute this grouping of transactions by customer: sorting, hashing, or nested-loops. For the purpose of this example, we will use the nested-loop variant and discuss alternative ways to compare the *customer* fields of two transactions in order to decide whether they belong to the same customer. Note that hashing and sorting are often more efficient variants, but they suffer from the same kind of error propagation as the QuickSort algorithm in the previous section.



(a) No Conflict (b) With Conflict

Figure 1: Example Vote Graphs for Table 1

Similarly to Section 2, we assume that the comparison of two customer names is the only error-prone and costly operation. Thus, the goal is to minimize the number of such comparisons and minimize the impact of mistakes made when computing these comparisons. Figure 1 illustrates one possible approach to do that. It depicts two example *Vote Graphs*. Such Vote Graphs capture the results of all comparisons carried out between the customer names of the four transactions. The nodes of a Vote Graph are transactions. Edges of a Vote Graph represent the results of comparing the customer names of two transactions. The weight of an edge indicates how often the comparison returned that results; the sign of the weights of an edge indicates the result of the comparison (true or false).

The Vote Graph of Figure 1a, for instance, indicates that we compared three times the customer names of Transactions 1 and 2 (i.e., "Jane = Bob") and all three times the answer was "false" (which happens to be the correct answer in this example). Furthermore, it shows that all seven comparisons between the customer names of Transactions 1 and 3 were positive (which happens to be correct, too, in this example).

3.2 Decision Functions

If minimizing comparisons between two customer names is our main objective (e.g., because they need to be crowdsourced or need to be executed repeatedly on an error-prone machine), then it makes sense to exploit the transitivity of the = relation. So, if the grouping algorithm asks whether Transactions 1 and 4 belong to the same customer in Figure 1a, the answer is *true* and can be inferred from Figure 1a without actually looking at the customer names of these transactions.

Transitivity and anti-transitivity can be applied in a straightforward way in the example of Figure 1a. The situation becomes trickier in Figure 1b because in that Vote Graph there is a conflicting edge: The negative edge between Transactions 1 and 4 conflicts with the positive edges, "1-3" and "1-4".

In the presence of error-prone computations, conflicts in the Vote Graph are inevitable. Therefore, it is important to tolerate these errors and make decisions even in conflict situations. In the example of Figure 1b, it is evident that the

system should conclude that the same customer carried out Transactions 1 and 4 because the weight of the edges “1-3” and “3-4” is much higher than the weight of the negative edge “1-4”. In general, we propose the use of a decision function that given a Vote Graph, determines whether two nodes are the same, not the same, or if additional comparisons are needed in order to make the decision.

There are many decisions functions conceivable and [8] contains a more detailed discussion of which properties a decision function should have. For instance, a decision function that always says that two nodes are the same is obviously not good because it will result in poor *quality*. Likewise, a decision function that always says “I do not know” is not good because it will result in high *cost* as it would induce additional comparisons. For the discussion in this paper, let us consider a decision function that is inspired by work on combining scoring functions [5] and that we call the MinMax function.

The MinMax function considers all positive and negative paths between two nodes. A positive path is a path that involves only edges with weight greater than 0. A negative path is a path that has exactly one negative edge. Paths with more than one negative edge are ignored because neither equality nor inequality can be inferred from them. For each path, the MinMax function computes a score: For a positive path, the score is the *minimum* of the weights of the edges of the path. For a negative path, the score is defined as the *minimum* of the *absolute* weights of the edges (i.e., the weight of the only negative edge is multiplied by -1 for this purpose). The intuition behind this scoring function is that a path is as strong as its weakest link. Another way to interpret the *minimum* is that it implements a conjunction (i.e., $\wedge$) along the path, thereby interpreting each edge as a predicate.

Continuing the example of Figure 1b, the score of the positive path ’1-3-4’ is 5 while the score for the negative path ’1-4’ is 1.

After computing the scores for all positive and negative paths, the MinMax decision function aggregates these scores into a single positive score, *pScore*, and a single negative score, *nScore*. *pScore* is the *maximum* of the scores of all positive paths. If there is no positive path, then *pScore* = 0. Analogously, *nScore* is the *maximum* of the scores of all negative paths. If there is no negative path, then *nScore* = 0. These values represent the maximum impact that a positive respectively negative path can have within an entity.

Finally, the MinMax function uses a threshold q in order to form a final decision based on the positive and negative scores; e.g., $q = 3$. That is, if the positive score is 3 or more higher than the negative score then the MinMax function decides that the two nodes are the same. More formally, the decision part of MinMax is defined as follows.

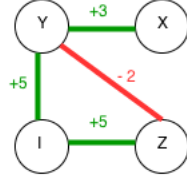


Figure 2: Interesting MinMax Example

$$f(r_1, r_2) = \begin{cases} \text{Yes}, & pScore(r_1, r_2) - nScore(r_1, r_2) \geq q \\ \text{No}, & nScore(r_1, r_2) - pScore(r_1, r_2) \geq q \\ \text{Do-not-know}, & otherwise \end{cases}$$

3.3 Observations

[8] contains a full discussion of this grouping / clustering use case under uncertainty with a series of experiments. The important observation and conclusion of [8] is that maintaining a Vote Graph and doing inference with the MinMax function is much better than doing pairwise comparisons in terms of both quality and cost in order to compute any database operation that is based on equality (e.g., joins, grouping, or clustering). In terms of cost, it is better because of its inference capability; in terms of quality, it is better because it detects inconsistencies and tries to keep the whole graph consistent. The designers of traditional database systems would never consider keeping such a Vote Graph because it is in traditional computing environments it is always cheaper (and as reliable) to recompute a comparison than to infer its result from a Vote Graph.

[8] discusses some of the properties of the MinMax decision function. It turns out that it is not transitive and an example can be seen in Figure 2 with a threshold of 3. In that example, the MinMax rules that "X = Y" (pScore=3, nScore=0) and "Y = Z" (pScore=5, nScore=2)), but it rules that "X = Z" is *unknown* (pScore=3, nScore=2). There are many conceivable decision functions; many which indeed are transitive. For instance, it would be possible to define a decision function by applying the MinCuts algorithm on every instance of the Vote Graph (i.e., after computing every comparison). This decision would indeed be transitive, but its implementation would have high computational cost. [8] proposes the MinMax function because it can be implemented in a highly efficient way.

For the purpose of designing good and robust algorithms for error-prone computer systems, however, we would like to make another important, somewhat surprising observation. Going back to Figure 2 and using the MinMax function, the best way to conclude that "X = Z" is *not* by comparing "X = Z" directly. Doing so would require, in the best case, five calls to the comparison function. Instead,

investing into the “Y = Z” edge is more promising: In the best case, two comparisons that confirm that indeed “Y = Z” are sufficient to finally conclude with the MinMax function that “X = Z”.

4 Conclusion and Related Work

The two examples showed some phenomena that may occur if computer systems make mistakes. The examples show that an optimal algorithm for the traditonal (error-free) computing model might result in poor quality when run on error-prone computer systems. It is an open question of what the optimal algorithms to sort a sequence of numbers and to group/cluster objects in the presence of errors are. The main message that we would like to illustrate with these examples is that error should be part of the equation. That is, we need to do two things:

- We need to design algorithms that scale (with the problem size) and tolerate errors. (Traditional algorithms were designed only to scale.)
- We need to optimize for both *cost* and *quality*. (Traditional algorithms were designed to minimize cost only.)

In other words, algorithm designers face two kinds of optimizations:

- Given a problem (e.g., sorting), a problem instance (e.g., 1000 integers), an error model (e.g., 1% of the comparisons are wrong uniformly) and a budget (e.g., 1 million comparisons), maximize the quality of the result.
- Given a problem, a problem instance, an error model, and quality requirements, minimize the cost.

At the moment, we do not even have good abstractions to characterize computational error and result quality.

The examples used in this paper were derived from typical database operators (i.e., sorting, joins, and grouping). Recently, there have a number of papers in the database community that studied how to enhance database with crowdsourcing, a special form of uncertain computation; e.g., [9, 11, 6, 3] to name just a few. It turns out that the topic of error-prone computing has been studied in other communities as well and not only in the context of crowdsourcing. For instance, Busse and Buhmann studied the information gain of a comparison in alternative sorting algorithms [2]. Schulze developed a method to carry out elections, called the Schulze method, which is similar to the MinMax decision function [12]. Furthermore, designers of distributed systems have been developing fault-tolerant algorithms for decades. The fact that several communities are looking into fault-tolerant computation makes it even more important to develop a theory that incorporates error and result quality in algorithm design and complexity.

References

[1] L. Avinash, K. K. Muntimadugu, C. C. Enz, R. M. Karp, K. V. Palem, and C. Piguet. Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling. In J. Feo, P. Faraboschi, and O. Villa, editors, *Conf. Computing Frontiers*, pages 3–12. ACM, 2012.

[2] L. M. Busse, M. H. Chehreghani, and J. M. Buhmann. The information content in sorting algorithms. In *ISIT*, pages 2746–2750. IEEE, 2012.

[3] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In W.-C. Tan, G. Guerrini, B. Catania, and A. Gounaris, editors, *ICDT*, pages 225–236. ACM, 2013.

[4] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, 2011.

[5] R. Fagin and E. L. Wimmers. A formula for incorporating weights into scoring rules. *Theor. Comput. Sci.*, 239(2):309–338, 2000.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In Sellis et al. [13], pages 61–72.

[7] A. Gruenheid and D. Kossmann. Cost and quality trade-offs in crowdsourcing. In R. Cheng, A. D. Sarma, S. Maniu, and P. Senellart, editors, *DBCrowd*, volume 1025 of *CEUR Workshop Proceedings*, pages 43–46. CEUR-WS.org, 2013.

[8] A. Gruenheid, D. Kossmann, S. Ramesh, and F. Widmer. Crowdsourcing entity resolution: When is a=b? Technical Report No. 785, Department of Computer Science, ETH Zurich, Sep 2012.

[9] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of qurk: a query processor for humanoperators. In Sellis et al. [13], pages 1315–1318.

[10] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[11] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[12] M. Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.

[13] T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. ACM, 2011.