Data Debugging

Daniel W. Barowy Dimitar Gochev Emery D. Berger

Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{dbarowy,gochev,emery}@cs.umass.edu

Abstract

Testing and static analysis tools can help root out bugs in programs, but not bugs in data. Checking data for errors is arguably as important as finding program errors, but lacks effective tool support. Previous approaches like data cleaning and statistical outlier analysis require either ground truth data for cross-validation, or that the data follow a known statistical distribution.

This paper introduces *data debugging*, an approach that combines data dependence analysis with statistical analysis to find and rank potential data errors. Since it is impossible to know *a priori* whether data are erroneous or not, data debugging instead reveals data whose impact on the computation is unusual. Data debugging is particularly promising in the context of data-intensive programming environments that intertwine data with programs (in the form of queries or formulas).

This paper presents the first data debugging tool, CHECKCELL, an add-in for Microsoft Excel. CHECKCELL highlights values in shades proportional to the unusualness of their impact on the spreadsheet's computation, which includes charts and formulas. CHECKCELL is efficient; its algorithms are asymptotically optimal, and the current prototype runs in seconds for most spreadsheets we examine. We perform a case study by employing workers via a crowdsourcing platform, and show that CHECKCELL is effective at finding actual data entry errors.

1. Introduction

In many computational tasks, correctness is a primary concern. Most work in the programming language community has focused on ways to discover whether the program performing the computation is correct. Techniques to reduce program errors range from testing and runtime assertions, to dynamic and static analysis tools that can discover a wide range of bugs. These tools and approaches enable programmers to find errors and reduce their impact, contributing to improving overall code quality.

However, a program is just one part of a computation. Existing tools ignore the correctness of program *inputs*. If the input contains errors, the result of the computation is likely to not be correct. Unlike programs, data cannot easily be tested or analyzed for correctness.

Input data errors can arise in a variety of ways [19]:

- **Data entry errors**, including typographical errors and transcription errors from illegible text.
- **Measurement errors**, when the data source itself, such as a disk or a sensor, is faulty or corrupted (unintentionally or not).
- **Data integration errors**, where inconsistencies arise due to the mixing of different data, including unit of measurement mismatches.

While data errors pose a threat to the correctness of any computation, they are especially problematic in data-intensive programming environments like databases, spreadsheets, and certain scientific computations. In these settings, data correctness can be as important as program correctness ("garbage in, garbage out"). The results produced by the computations—queries, formulas, charts, and other analyses—may be rendered invalid by data errors. These errors can be costly: errors in spreadsheet data have led to losses of millions of dollars [29, 30].

By contrast with the proliferation of tools at a programmer's disposal to find program errors, few tools exist to help find data errors. Part of the problem is that it can be difficult to decide whether any given data element is an error or not. For example, the number `1234` might be correct, or the correct value might be `12.34`. Typographical errors can change data items by orders of magnitude. Unfortunately, finding this kind of mistake via manual data auditing is onerous, unscalable, and error-prone.

Existing approaches to finding data errors include *data cleaning* and *statistical outlier detection*. Data cleaning primarily copes with errors via cross-validation with ground truth data, which may not be present. Statistical outlier detection typically reports data as outliers based on their relationship to a given distribution (e.g., Gaussian). Automatic identification of data distributions is error-prone and can give rise to excessive false positives.

Contributions

This paper presents **data debugging**, an approach for locating potential data errors. Since it is impossible to know *a priori* whether data are erroneous or not, data debugging does the next best thing: *locating data that has an unusual impact on the computation*. Intuitively, data that has an inordinate impact on the final result is either very important, or it is wrong. By contrast, wrong data whose presence has no particularly unusual effect on the final result does not merit special attention. Data debugging combines data dependence analysis and statistical analysis to find and rank data based on the unusualness of its impact on the results of a computation.

Data debugging works by first building a data dependence graph of the computations. It then measures data impact by replacing data items with data chosen from the same group (e.g., a range in a spreadsheet formula) and observing the resulting changes

in computations that depend on that data. This non-parametric approach allows data debugging to find errors in both numeric and non-numeric data, without any requirement that data follow any particular statistical distribution.

By calling attention to data with unusual impact, data debugging can provide insights into both the data and the computation and reveal errors. We believe data debugging is broadly applicable, though it is especially well-suited for data-intensive programming that intertwine data and programs (e.g., with queries and formulas).

This paper presents the first data debugging tool in the form of CHECKCELL, an add-in for Microsoft Excel. Spreadsheets are one of the most widely-used programming environments, and this domain has recently attracted renewed academic attention [16, 18, 31]. In addition, spreadsheet errors are a well known risk and have led to significant monetary losses in the past, making them an excellent first target for data debugging.

CHECKCELL highlights all data whose impact crosses a threshold of unusualness that is more than two standard deviations away from the mean impact. In the user interface, CHECKCELL ranks these cells by coloring them in shades proportionally to their impact: the brighter a cell is highlighted, the more unusual impact it has. CHECKCELL is empirically and analytically efficient and effective, as we show in Sections 4 and 5. The current prototype is untuned but analysis time is generally low, taking seconds to run on most of the spreadsheets we examine. We perform a case study by employing human workers via a crowdsourcing platform (Amazon's Mechanical Turk), and show that CHECKCELL is effective at finding actual data entry errors.

Outline

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the algorithms that data debugging employs. Section 4 derives analytical results that demonstrate data debugging's runtime efficiency and effectiveness. Section 5 presents an empirical evaluation of data debugging in the form of CHECKCELL, measuring its runtime performance and its effectiveness at finding errors. Section 6 describes directions for future work, and Section 7 concludes.

2. Related Work

Data Cleaning

Most past work on locating or removing errors in data has focused on *data cleaning* or *scrubbing* in database systems [17, 24]. Standard approaches include statistical outlier analysis for removing noisy data [32], interpolation to fill in missing data (e.g., with averages), and using cross-correlation with other data sources to correct or locate errors [20].

A number of approaches have been developed that allow data cleaning to be expressed programmatically or applied interactively. Programmatic approaches include AJAX, which expresses a data cleaning program as a DAG of transformations from input to output [14]. Data Auditor applies rules and target relations entered by a programmer [15]. A similar domain-specific approach has been employed for data streams to smooth data temporally and isolate it spatially [22]. Potter's Wheel, by Raman and Hellerstein, is an interactive tool that lets users visualize and apply data cleansing transformations [25].

To identify errors, Luebbers et al. describe an interactive data mining approach based on machine learning that builds decision trees from databases. It derives logical rules (e.g., "BRV $= 404 \Rightarrow$ GBM $= 901$") that hold for most of the database, and marks deviations as errors to be examined by a data quality engineer [23]. Raz et al. describe an approach aimed at arbitrary software that uses Daikon [9] to infer invariants about numerical input data and then report discrepancies as "semantic anomalies" [26]. Data debugging is orthogonal to these approaches: rather than searching for latent relationships in or across data, it measures the interaction of data with the programs that operate on them.

	A	B	C
1	**MONTHLY BUDGET**		
2		**Projected Cost**	**Actual Cost**
3	Rent	1150	1150
4	Phone	3675	36.75
5	Gas & Electricity	80	87.23
6	Waste removal	11.25	11.25
7	Groceries	200	187.81
8	Car payment	225	225
9	Gasoline	50	62.3
10	Clothing	100	59.99
11	Total	5491.25	1820.33
12	Fancy dinner tonight?	**Yes**	

Figure 1. A sample spreadsheet showing a personal budget, with an unfortunate typographical error (see Section 3).

Spreadsheet Errors

Spreadsheets have been one of the most prominent computer applications since their creation in 1979. The most widely used spreadsheet application today is Microsoft Excel. Excel includes rudimentary error detection including errors in formula entry like division by zero, a reference to a non-existient formula or cell, invalid numerical arguments, or accidental mixing of text and numbers. Excel also checks for inconsistency with adjacent formulas and other structural errors, which it highlights with a "squiggly" underline. In addition, Excel provides a formula auditor, which lets users view dependencies flowing into and out of particular formulas.

Past work on detecting errors in spreadsheets has focused on inferring units and relationships (has-a, is-a) from information like structural clues and column headers, and then checking for inconsistencies [1, 2, 7, 10, 11]. For example, XeLda checks if formulas process values with incorrect units or if derived units clash with unit annotations. There also has been considerable work on testing tools for spreadsheets [6, 13, 27, 28].

This work is complementary and orthogonal to CHECKCELL, which works with standard, unannotated spreadsheets and focuses on unusual interactions of data with formulas.

Statistical Outlier Analysis

Techniques to locate outliers date to the earliest days of statistics, when they were developed to make nautical measurements more robust. Widely-used approaches include Peirce's criterion, Chauvenet's criterion, and Grubb's test for outliers [5]. All of these techniques require that data belong to a known distribution, primarily the normal (Gaussian). Unfortunately, input data does not necessarily fit any statistical distribution. Moreover, identifying outliers leads to false positives when they do not materially contribute to the result of a computation (i.e., have no impact). By contrast, data debugging only reports data items with a substantial impact on a computation.

3. Data Debugging: Algorithms

This section describes in detail the algorithms that data debugging employs. Section 4 includes formal analysis of various aspects of the data debugging algorithms described here, including asymptotic performance and statistical effectiveness.

Throughout this section, we illustrate these algorithms with a running example of a budget shown in Figure 1, which is a reduced version of a sample template included with Microsoft Excel.

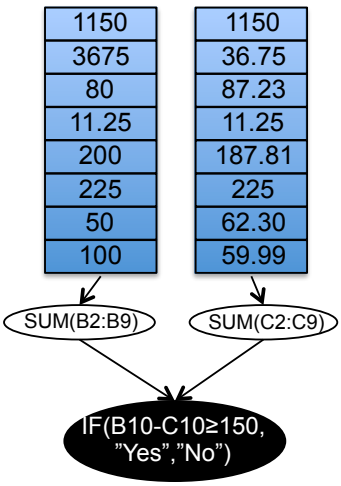


Figure 2. The dependence graph that CHECKCELL extracts from the spreadsheet in Figure 1.

This spreadsheet tracks expected versus real spending on monthly expenses. The spreadsheet indicates whether the user can afford to splurge and go out to a fancy restaurant, which is when real spending is at least $150 less than expected. However, this example contains an unfortunate typographical error that could lead the user to spend money that he or she does not actually have.

3.1 Dependence Analysis

The first step in data debugging is to identify the relationship of data (inputs) to computations (outputs). In a spreadsheet, inputs are data in cells, while computations are either charts or formulas that are not used by other formulas. CHECKCELL uses techniques similar to past work to identify dependencies in spreadsheets [13], but adds support for charts.

Charts are often a key "result" of a spreadsheet computation, so it is important to detect data that triggers dramatic changes in charts. To handle charts, CHECKCELL treats them as if they were formulas that compute the average over their inputs. A data value that significantly alters the average is considered to have an unusual impact on the chart.

Figure 2 shows the dependence graph that CHECKCELL extracts from this spreadsheet. The final formula, the only output in this spreadsheet (shown in black), depends on two formulas, which depend on two disjoint data ranges.

3.2 Impact Analysis

The next step is to iterate through the data itself to test the impact of each data item on all computations. For each item, data debugging repeatedly chooses another item from the same "distribution", e.g., another tuple in the same table, or another cell in the same range, and replaces the item being tested with the selected one.

The selection process is exhaustive for small ranges (less than 30 elements); that is, for each item, impact analysis tests the effect of replacing it with every other item in the range. For larger ranges, we employ random sampling.

The computations are then recalculated using this new dataset. Changes in the computations are recorded as their *impact scores* on each data item. The impact score for a computation depends on whether the output is numeric or non-numeric. For numeric data, the impact score is the absolute change from the original value. For non-numeric data, the impact score is an *indicator value*: 1 if the result of the computation changed, and 0 if not.

	A	B	C
1	**MONTHLY BUDGET**		
2		**Projected Cost**	**Actual Cost**
3	Rent	1150	1150
4	Phone	3675	36.75
5	Gas & Electricity	80	87.23
6	Waste removal	11.25	11.25
7	Groceries	200	187.81
8	Car payment	225	225
9	Gasoline	50	62.3
10	Clothing	100	59.99
11	Total	5491.25	1820.33
12	Fancy dinner tonight?	**Yes**	

Figure 3. The same spreadsheet, with the error highlighted by CHECKCELL as the value with the most unusual impact on the computation.

For the spreadsheet in Figure 1, CHECKCELL starts with cell `B3`. It chooses an item from the same range, for example, cell `B6`. Replacing `B3` with `B6` results in a new sum, $4352.50. Since this sum does not change the `Yes` answer in the formula, the impact score for cell `B3` remains 0.

However, the same procedure for cell `B4` is likely to have a different effect. For example, replacing `B4` with `B9` yields a sum of $1866.25. The difference between this sum and actual spending has now dropped below $150, triggering a change in the result to `No`. CHECKCELL then adds 1 to cell `B4`'s impact score.

To minimize sampling error, random selection (for large ranges) is repeated a fixed number of times (at least 30), accumulating the impact scores associated with each datum. The impact score is then divided by the number of iterations. Section 4.1 shows that the runtime efficiency of this approach is asymptotically optimal, and Section 4.2 explains why sampling can be nearly as effective as checking each item's impact exhaustively.

3.3 Impact Scoring

In the final phase, the impacts of each data item are normalized by transforming them into a test statistic computed as the absolute distance of each item from the sample mean, divided by the sample standard deviation. This *normalized impact score* represents the distance from the mean impact, in numbers of standard deviations. A standard approach, which we adopt here, is to only report data whose impact is at least two standard deviations away from the mean. Note that this interpretation does not depend on normality assumptions about the data or its impact (see Section 4).

Each data item's normalized impact scores are then averaged across all the outputs, and data debugging assigns that average as the *overall impact score* of each data item. Intuitively, data with large overall impact scores either have an extremely unusual impact on a small number of computations, or an unusual impact on many computations. The overall impact score is used both for ranking and for displaying the relative anomalousness of the impact of particular data items, e.g., by coloring such values in brighter colors corresponding to their distance from the mean.

Figure 3 shows the effect of running CHECKCELL, which in this case identifies cell `B4` as the only one with an unusual impact on the spreadsheet.

4. Data Debugging: Analysis

This section presents an analysis of several aspects of data debugging: its runtime, its sampling approach to measure impact for large data ranges, and its scoring to perform impact outlier analysis.

4.1 Asymptotic Runtime

Data debugging operates in several phases: computing the dependence graph, performing impact analysis, and then ranking impacts. Runtime depends on the following parameters: the number of data items (n), the number of formulas (f), and the number of inputs (i). In spreadsheets, the number of inputs equals the number of data items, but in other contexts like databases, inputs correspond to fields, so $i \ll n$. Since each formula and data item must be examined at least once to compute the dependence graph and to measure impact, respectively, runtime for data debugging must be $\Omega(n + f)$.

The cost of building the dependence graph varies depending on the structure of the computation. It has a worst-case runtime of $O((i * f)^2)$, quadratic in the total number of inputs and formulas; it is theoretically possible for each formula to depend on every input and other formula. However, this kind of pathological computation structure is atypical. Dependence graphs normally form a tree or a forest of trees. In this case, the cost of constructing the dependence graph becomes linear in the number of inputs and formulas, or $O(i + f)$.

Impact analysis dominates the costs of constructing the dependence graph and ranking impacts, since it requires recalculation of the computations affected by changes in the data.

A naïve implementation of impact analysis that checked the impact of each data item by systematically replacing it with every other item in the same range would require $O(n^2)$ time. Worse, each of these iterations requires potentially costly recalculations. For large ranges, such an approach would make data debugging unusable in practice.

By using a fixed number of random selections once the range exceeds a threshold size, data debugging keeps the total number of recalculations to $O(n)$, linear in the number of data items. Any strategy that visits each data item at least once takes $O(n)$ time, so this bound is tight. Section 4.2 explains why this strategy approximates the effect of a complete examination of other inputs, while minimizing execution time.

Finally, ranking impacts involves only two linear passes over the impacts to compute absolute impact scores, so it also operates in (optimal) linear time in the number of impacts.

4.2 Sampling Effectiveness

A complete strategy for measuring the impact of any data item i is to systematically measure the impact of replacing it with every other item j in the same range:

$$\text{impact}_i = \frac{\sum_{j \neq i}^{N} |\text{result}_i - \text{result}_j|}{N} \tag{1}$$

This approach, as mentioned above, would take $O(n^2)$ time and so is inefficient for large data sets, though it is reasonable to do for small ones.

For large data sets, data debugging employs a sampling-based strategy that randomly chooses (with replacement) a fixed number of samples K (e.g., $K = 30$).

$$\text{estimated\_impact}_i = \frac{\sum_{j \in \{\text{sample}\}}^{N} |\text{result}_i - \text{result}_j|}{K} \tag{2}$$

The following theorem establishes that using the estimated impact is likely to result in minimal error.

Theorem 4.1. *When the percentage of values with unusual impacts is low, the estimated impact closely approximates the actual impact because it minimizes the risk of false positives and false negatives.*

There are two cases where using the estimated impact will have a different effect than using the actual impact:

- **False negatives.** The item under consideration has an unusual actual impact, but the sampling procedure repeatedly chooses other items with similarly anomalous impacts, so the item under consideration appears to have only average impact (the anomalous values cancel out).
- **False positives.** The item under consideration does not have an unusual actual impact, but the sampling procedure repeatedly chooses items that do have an unusual impact. Now, the item under consideration appears to have an anomalous impact, despite the fact that it actually does not.

The key to bounding the likelihood of either false negatives or false positives is to ensure that the sampling process does not repeatedly sample data items with unusual impacts.

Recall that data debuggging considers a data item to have an unusual impact when that impact is at least two standard deviations away from the mean impact in that range. When impacts follow a normal distribution, the number of items with unusual impact will be less than 5%. Of course, impacts will not necessarily be normally distributed, although they will be when the computations include averages (by virtue of the Central Limit Theorem). Nonetheless, as long as the tails of the impact distribution comprise a small fraction of the total, the theorem holds, as we explain below.

One can view the sampling procedure as a series of Bernoulli trials, repeatedly flipping a biased coin with a probability $p = 0.05$ of choosing a value whose impact is anomalous (heads) and a probability $q = 0.95$ of choosing a value with a non-anomalous impact (tails). For n coin flips and probability p of heads, the expected number of times that the sampling procedure will choose an anomalous value is just np, or 1.5.

Because the Poisson distribution is an excellent approximation as long as n is at least 20 and p is no more than 0.05, it is convenient to use it to approximate the likelihood of choosing a larger number of high-impact values than x, where $x > \lambda$ and $\lambda = np$:

$$Pr[X \geq x] \leq \frac{e^{-\lambda}(e\lambda)^x}{x^x} \tag{3}$$

We can use this equation to show that it is highly unlikely that the sampling procedure would accidentally choose a large number of unusually-impactful values. For example, the probability of choosing one-third of those values from a sample of size $K = 30$ is less than $3/100,000$.

Therefore, the estimated impact computed by sampling is nearly as effective at reducing both false positives and false negatives as computing the actual impact.

4.3 Impact Outlier Analysis

Once all impact scores have been computed, only those data whose impacts cross some threshold of anomalousness should be reported. We currently treat the impact scores as if they fit a normal distribution, and only report data whose scores place them more than two standard deviations away from the mean. In a normal distribution, that is just under 0.05% of the population; in other words, this corresponds to a 95% confidence level that these are anomalies.

In the absence of knowledge of the distribution of impact scores, using the normal distribution is a conservative approach. That is, it is likely to report few false negatives, at the risk of introducing some false positives. This fact derives from two particular characteristics of the normal distribution: its low *skewness* and *kurtosis*.

The normal distribution has zero skewness, where skew is the distribution around the mean; in other words, it is perfectly symmetric. Any asymmetric distribution by definition has a greater number of points either to the left or to the right of the mean. By choosing outliers from the tails of the normal, CHECKCELL also includes one of the skewed tails of any asymmetric distribution.

In addition, the normal has either zero kurtosis, which indicates the peakedness of the curve and heaviness of the tail. Counting outliers from the perspective of the normal distribution is generally conservative, since it includes all distributions with heavier tails (i.e., those with positive kurtosis). Because distributions with negative kurtosis generally have small tails, they also tend to have few outliers, by definition. Thus, failing to report outliers for distributions with negative kurtosis is usually safe.

One exception worth noting and a limitation of this approach is symmetric multimodal distributions, where the outlier values are not only in the tail but also centered around the mean. Using the normal as a reference would not uncover such outliers.

5. Evaluation

We evaluate CHECKCELL across two dimensions: its execution time, and its effectiveness at finding actual errors.

Our experimental platform is a 13" MacBook Air equipped 4GB of RAM and an Intel Core i5-2557M processor running at 1.70GHz. The operating system is Windows 7 Professional (SP1), which executes non-virtualized (via Bootcamp). CHECKCELL was compiled using Microsoft Visual C# 2010, and runs as an add-in in Microsoft Excel 2010.

5.1 Execution Time

To measure the runtime of CHECKCELL, we run it on a random subset of 30 spreadsheets drawn from the EUSES corpus [12], excluding those that do not contain formulas.

Table 1 includes characteristics of these spreadsheets, ordered by the number of formulas each contains. We include two columns that count the number of cells in different ways. *Cells (raw)* indicates the total number of cells that participate in any computation. *Cells (weighted)* indicates the total number of cells *inside ranges*, weighted by the number of times each cell is used in a computation. For example, a cell that is in a range involved in two computations is counted twice. Because the weighted cell count only includes ranges, it is possible for it to be lower than the raw number of cells.

Figure 4 reports the performance of data debugging across our spreadsheet suite, ordered by the weighted number of cells. Table 1 includes the full data.

For 19 of the 30 spreadsheets, CHECKCELL takes 9 seconds or less to complete. Its runtime is less than three minutes for all but two of the spreadsheets: `intresults` and `NEW`, which take 318 seconds and 683 seconds, respectively. The average runtime over all spreadsheets is 61 seconds; without the two outliers, it is 29 seconds. As our analysis in Section 4.1 predicts, the cost of CHECKCELL is generally proportional to the cost of the impact analysis, which is in turn dependent on the weighted number of cells.

The spreadsheets that require the most execution time both have by far the largest number of formulas (1,066 and 2,626), and the latter also has the largest number of weighted cells (2,403). Their relatively high execution time is attributable to the fact that the cost of impact analysis increases as the number of formulas increases, since the Excel recalculation engine must do more work per item tested. The `intresults` spreadsheet also has an extremely highly-connected clique in its dependence graph, which leads to both higher time for dependence analysis and increases the cost of recalculations during impact analysis (see Figure 7).

Summary: For nearly every spreadsheet examined, CHECKCELL's runtime is under three minutes; we believe this overhead is acceptable for an error detection tool.

5.2 Error Detection

While CHECKCELL can be used across the EUSES suite, looking for errors in existing spreadsheets is problematic because we do not know what the ground truth is. To evaluate CHECKCELL's efficacy at finding actual errors, we need errors and ground truth to compare it against.

Rather than artificially inject errors, we designed an experiment that allows us to observe real errors produced by people and use CHECKCELL to find them. We collect human errors by hiring workers to perform data entry tasks (entering known data) via Amazon's Mechanical Turk, a popular crowdsourcing platform, and then check their results with CHECKCELL.

Our ground truth data is drawn from `3q2000.xls`, a spreadsheet from the EUSES repository that contains selected financial information from Fannie Mae. We save the data as a comma-separated value file (.csv). Mechanical Turk workers were paid 3 cents to enter 10 of these numerical values at a time into a web form designed to look like a spreadsheet, shown in Figure 5. To prevent copying and pasting, we generate an image containing the comma-separated values. Each worker had the opportunity to perform up to seven different tasks.

In all, we collected 200 responses from 46 distinct users. Out of these responses, 14 had omitted data and 52 contained errors, for an overall error rate of 33%. The errors can be classified into the following categories:

- **Sign omission**, where a negative sign was dropped;
- **Magnitude errors**, any change in a value (usually a dropped or spurious digit) that results in an order of magnitude increase or decrease;
- **Digit transposition**, where at least two digits are transposed;
- **Typos**, any other typographical error (e.g., a mistyped digit).

We then inserted the erroneous data back into the spreadsheet one at a time and ran CHECKCELL to see whether it found any of these errors. Recall that by design, CHECKCELL reports data with an unusual impact on any of the calculations. For this spreadsheet, CHECKCELL always highlights the values in the top row (the net interest income) because these values have a significant impact on the spreadsheet; most of the income in this spreadsheet comes from this row. We classify CHECKCELL as having correctly found an error if it also highlights an erroneous cell.

For 13 of the 52 erroneous inputs (25%), CHECKCELL correctly marks the cell with the error, supporting our hypothesis that locating data with unusual impact also finds errors. In all but two of these cases, the error was a magnitude error; such errors are likelier to have an unusual impact on a computation than all other errors, since they change the input data dramatically. Even sign omission only causes a factor of two change in a data element. Nonetheless, 20 of the errors that CHECKCELL does not report also involve magnitude errors, but those errors occur in data that do not contribute significantly to any computation.

Figure 6 presents a screenshot of CHECKCELL's results with one of these errors. In addition to the top row, CHECKCELL indicates that cell `G19` has an unusual impact; this is, in fact, the error. The correct value for `G19` is `-379300000`, and the value entered by the worker was `3793000000`: the worker made both a sign error and an order of magnitude error (one too many 0's).

Summary: By searching for data with unusual impacts on the spreadsheet, CHECKCELL is able to successfully find actual human data entry errors.

5.3 Impact Distribution

CHECKCELL treats inputs whose impact scores are more than two standard deviations from the mean impact as outliers. When impact scores are normally distributed, we can strongly claim that scores outside 2 standard deviations are unusual, and that standard outlier

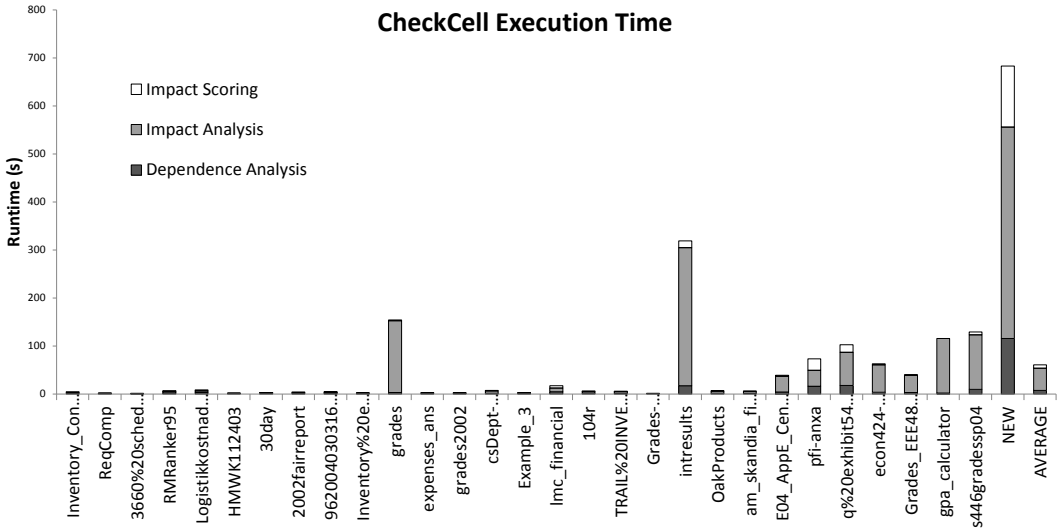


Figure 4. CHECKCELL execution time. For most of the spreadsheets, CHECKCELL completes its analysis in under 9 seconds; for all but two, it completes in under three minutes (see Section 5.1).

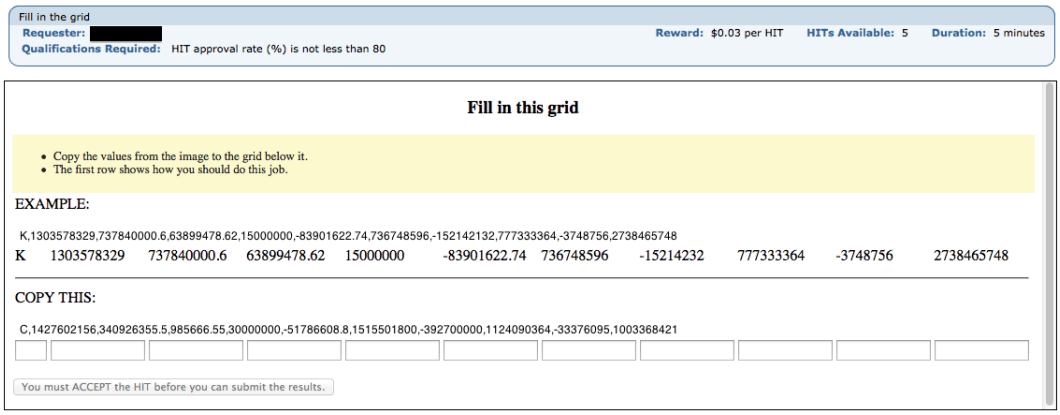


Figure 5. The page presented to Mechanical Turk workers to perform data entry tasks in order to collect actual human data entry errors (see Section 5.2).

rejection techniques are non-controversial. While CHECKCELL does not assume normality for our outlier rejection procedure, we empirically evaluate the distribution of average impacts produced by our suite of benchmarks.

Several of our benchmarks are empty forms and are thus populated only with zero values. We exclude these spreadsheets from our analysis since the standard deviation of their average impacts is, by definition, zero, and they would thus be vacuously normal.

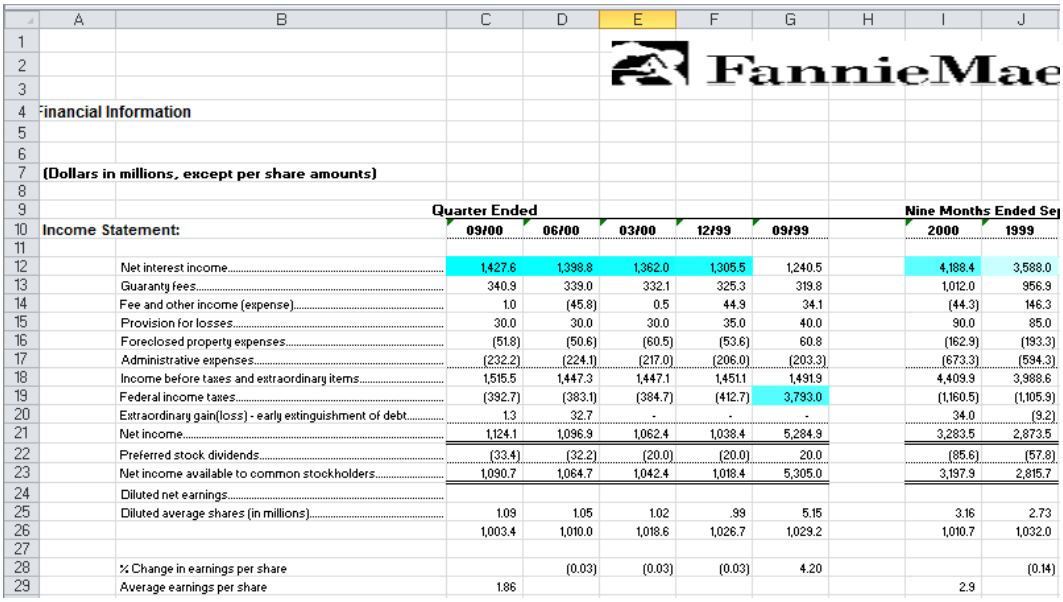
Our analysis of 23 spreadsheets shows that average impact scores which are more than two standard deviations from the mean impact score are in fact rare, thus supporting our decision to reject them. In a standard normal distribution, no more than 5% of values are found outside 2 standard deviations. In our evaluation, on average, 1.45% of impact values fall outside 2 standard deviations from the mean. Limiting our analysis to ranges of size 15 or greater, approximately 3.47% of their impact values fall outside 2 standard deviations. The latter is a stronger evidence, since small distributions tend to have fewer outliers.

6. Future Work

In future work, we plan to explore applying data debugging to other data-intensive domains, including Hadoop/MapReduce tasks [3, 8], scientific computing environments like R [21], and database management systems, especially those with support for "what-if" queries [4].

We expect all of these domains will require some tailoring of the existing algorithms to their particular context. For databases, we plan to treat as computations both stored procedures and cached queries. While it is straightforward to apply data debugging to databases when queries have no side effects, handling queries that do modify the database will take some care in order to avoid an excessive performance penalty due to copying.

A similar performance concern arises with Hadoop, where the key computation is the relatively costly reduction step. Data debugging will also likely need to take into account features of the R language in order to work effectively in that context. Finally, we are interested in exploring the effectiveness of data debugging in conventional programming language settings.



	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4	Financial Information									
5										
6										
7	(Dollars in millions, except per share amounts)									
8										
9			Quarter Ended						Nine Months Ended Se	
10	Income Statement:		09/00	06/00	03/00	12/99	09/99		2000	1999
11										
12		Net interest income	1,427.6	1,398.8	1,362.0	1,305.5	1,240.5		4,188.4	3,588.0
13		Guaranty fees	340.9	339.0	332.1	325.3	319.8		1,012.0	956.9
14		Fee and other income (expense)	1.0	(45.8)	0.5	44.9	34.1		(44.3)	146.3
15		Provision for losses	30.0	30.0	30.0	35.0	40.0		90.0	85.0
16		Foreclosed property expenses	(51.8)	(50.6)	(60.5)	(53.6)	60.8		(162.9)	(193.3)
17		Administrative expenses	(232.2)	(224.1)	(217.0)	(206.0)	(203.3)		(673.3)	(594.3)
18		Income before taxes and extraordinary items	1,515.5	1,447.3	1,447.1	1,451.1	1,491.9		4,409.9	3,988.6
19		Federal income taxes	(392.7)	(383.1)	(384.7)	(412.7)	3,793.0		(1,160.5)	(1,105.9)
20		Extraordinary gain(loss) - early extinguishment of debt	1.3	32.7	-	-	-		34.0	(9.2)
21		Net income	1,124.1	1,096.9	1,062.4	1,038.4	5,284.9		3,283.5	2,873.5
22		Preferred stock dividends	(33.4)	(32.2)	(20.0)	(20.0)	20.0		(85.6)	(57.8)
23		Net income available to common stockholders	1,090.7	1,064.7	1,042.4	1,018.4	5,305.0		3,197.9	2,815.7
24		Diluted net earnings								
25		Diluted average shares (in millions)	1.09	1.05	1.02	.99	5.15		3.16	2.73
26			1,003.4	1,010.0	1,018.6	1,026.7	1,029.2		1,010.7	1,032.0
27										
28		% Change in earnings per share		(0.03)	(0.03)	(0.03)	4.20			(0.14)
29		Average earnings per share	1.86						2.9	

Figure 6. A screenshot of CHECKCELL's results. In addition to the top row, which has a large impact on the final results, CHECKCELL highlights cell `G19`, a human data entry error.

While CHECKCELL's speed is reasonable in most cases, we are interested in further optimizing it. We are especially interested in developing a version that incrementally updates its impacts on-the-fly. This version would run in the background and detect data with unusual impacts as they are entered, much like modern text entry underlines misspelled words. We believe that having automatic detection of possible data errors on all the time could greatly reduce the risk of data errors.

7. Conclusion

This paper presents data debugging, an approach aimed at finding potential data errors by locating and ranking data items based on their overall impact on a computation. Intuitively, errors that have no impact do not pose a problem, while values that have an unusual impact on the overall computation are either very important or incorrect.

We present the first data debugging tool, CHECKCELL, which operates on spreadsheets. We evaluate CHECKCELL's performance analytically and empirically, showing that it is reasonably efficient and effective at helping to find data errors. CHECKCELL is available for download at `https://github.com/plasma-umass/DataDebug`.

Acknowledgments

Thanks to Charlie Curtsinger for valuable discussions during the evolution of this project.

References

[1] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *ASE*, pages 174–183. IEEE Computer Society, 2003.

[2] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Apache Foundation. Welcome to Apache Hadoop. `http://hadoop.apache.org/`, Nov. 2012.

[4] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 220–231, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[5] V. Barnett and T. Lewis. Outliers in statistical data. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, Chichester: Wiley, 1994, 3rd ed.*, 1, 1994.

[6] J. Carver, M. Fisher, II, and G. Rothermel. An empirical evaluation of a testing and debugging methodology for excel. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 278–287, New York, NY, USA, 2006. ACM.

[7] C. Chambers and M. Erwig. Reasoning about spreadsheets with labels and dimensions. *J. Vis. Lang. Comput.*, 21(5):249–262, Dec. 2010.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[10] M. Erwig. Software engineering for spreadsheets. *IEEE Softw.*, 26(5):25–30, Sept. 2009.

[11] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 136–145, New York, NY, USA, 2005. ACM.

[12] M. Fisher and G. Rothermel. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes*, July 2005.

[13] M. Fisher, G. Rothermel, T. Creelan, and M. Burnett. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 13–22. IEEE, 2006.

[14] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: an extensible data cleaning tool. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, page 590, New York, NY, USA, 2000. ACM.

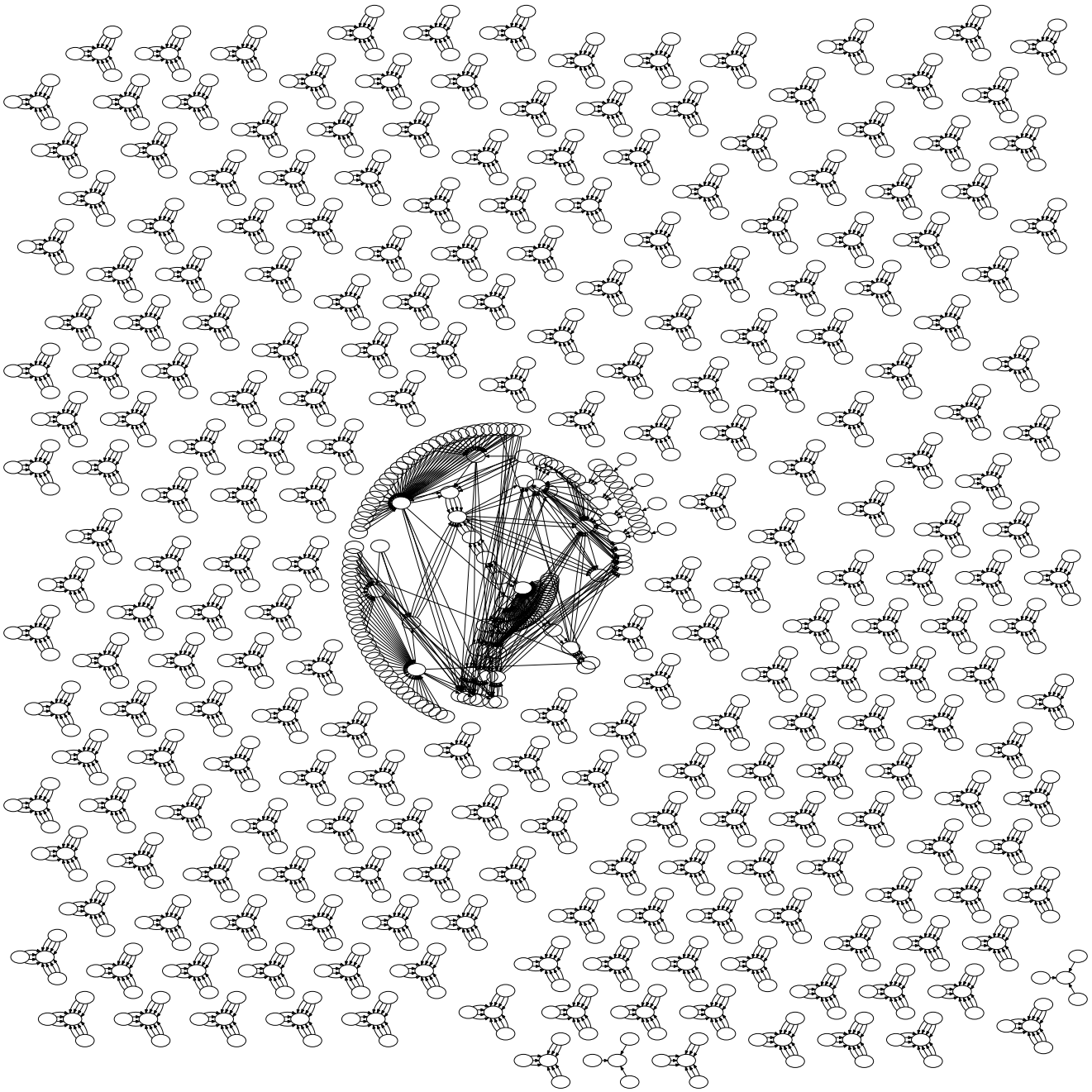


Figure 7. A dataflow graph for the `intresults` spreadsheet, one of the benchmarks with an unusually long runtime. The highly-connected clique in the center causes the impact analysis to dominate CHECKCELL's computation time since a change in a single cell may require the recomputation of a large number of values.

[15] L. Golab, H. Karloff, F. Korn, and D. Srivastava. Data auditor: exploring data quality and semantics using pattern tableaux. *Proc. VLDB Endow.*, 3(1-2):1641–1644, Sept. 2010.

[16] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *POPL*, pages 317–330. ACM, 2011.

[17] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[18] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 317–328. ACM, 2011.

[19] J. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[20] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 127–138, New York, NY, USA, 1995. ACM.

[21] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[22] S. Jeffery, G. Alonso, M. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 140–142, Apr. 2006.

[23] D. Luebbers, U. Grimmer, and M. Jarke. Systematic development of data mining-based data quality tools. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 548–559. VLDB Endowment, 2003.

[24] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[25] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 381–390, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[26] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 302–312, New York, NY, USA, 2002. ACM.

[27] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):110–147, 2001.

[28] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 198–207. IEEE, 1998.

[29] M. Sakal and L. Raković. Errors in building and using electronic tables: Financial consequences and minimisation techiques. *International Journal of Strategic Management and Decision Support Systems in Strategic Management*, 17(3):29–35, 2012.

[30] V. Samar and S. Patni. Controlling the information flow in spreadsheets. *CoRR*, abs/0803.2527, 2008.

[31] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, Apr. 2012.

[32] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, Mar. 2006.

Spreadsheet	**Formulas**	**Cells** *raw*	**Cells** *weighted*	**Runtime** *total (s)*	**Dep. Analysis**	**Impact Analysis**	**Impact Scoring**
3660 schedule S2003	31	1	0	1.54	0.73	0.44	0.34
ReqComp	54	162	0	1.95	0.95	0.52	0.44
Inventory_Control	33	21	0	4.71	1.67	1.59	1.42
RMRanker95	79	54	11	7.07	2.74	2.38	1.91
Logistikkostnader	73	29	26	8.88	3.48	2.97	2.40
HMWK112403	36	41	27	2.31	0.88	0.78	0.63
30day	125	92	30	3.01	1.39	1.31	0.27
2002fairreport	3	39	39	4.30	1.29	1.67	1.30
9620040303160820	42	81	81	4.77	1.19	2.74	0.81
Inventory errors	100	129	90	2.83	1.25	1.02	0.53
grades	227	661	96	154.45	3.06	149.85	1.51
expenses_ans	57	60	120	3.24	0.92	2.15	0.15
grades2002	61	143	123	2.67	1.03	1.11	0.51
csDept-PayrollTimecardEntry	68	204	124	7.37	1.85	4.37	1.10
Example_3	71	130	127	3.15	1.22	1.56	0.28
lmc_financial	72	148	142	17.15	4.69	7.63	4.80
104r	22	146	144	6.66	1.81	3.26	1.54
TRAIL INVENTORY N#A850A	2	156	156	6.15	1.12	3.99	0.99
Grades-6_excerpt	106	168	168	1.83	1.10	0.45	0.25
intresults	1066	3158	239	318.91	17.12	287.63	14.12
OakProducts	69	271	242	6.82	1.67	4.20	0.91
am_skandia_fin_supple#A80EE	56	272	268	6.64	1.53	4.01	1.06
E04_AppE_Census_Database_50	42	300	300	39.04	4.07	32.72	2.22
pfi-anxa	5	310	310	73.56	16.38	33.10	24.05
q exhibit54-OEA	797	1160	365	102.56	18.03	68.70	15.79
econ424-fall2003-publ#A8A23	93	517	384	62.83	3.91	56.96	1.93
Grades_EEE481&581	177	757	756	40.11	3.31	35.74	1.03
gpa_calculator	80	80	819	115.86	1.88	113.67	0.28
s446gradessp04	335	1369	1247	129.36	9.76	113.29	6.27
NEW	2626	2574	2403	683.32	115.75	440.30	127.23

Table 1. The benchmark suite of 30 spreadsheets, a random sample from the EUSES repository [12], ordered by weighted number of cells. The raw number of cells indicates the total number of cells that are used in any formula; the weighted number of cells weighs cells *in ranges* by the number of formulas that depend on it. A breakdown of CHECKCELL execution times (in seconds) appears on the right side.