Advanced Code Coverage Analysis Using Substring Holes

Yoram Adler Eitan Farchi Moshe Klausner Dan Pelleg Orna Raz
Moran Shochat Shmuel Ur Aviad Zlotnick
IBM Haifa Research Lab, Israel
[adler,farchi,klausner,dpelleg,ornar,morans,ur,aviad]@il.ibm.com

ABSTRACT

Code coverage is a common aid in the testing process. It is generally used for marking the source code segments that were executed and, more importantly, those that were not executed.

Many code coverage tools exist, supporting a variety of languages and operating systems. Unfortunately, these tools provide little or no assistance when code coverage data is voluminous. Such quantities are typical of system tests and even for earlier testing phases. Drill-down capabilities that look at different granularities of the data, starting with directories and going through files to functions and lines of source code, are insufficient. Such capabilities make the assumption that the coverage issues themselves follow the code hierarchy. We argue that this is not the case for much of the uncovered code. Two notable examples are error handling code and platform-specific constructs. Both tend to be spread throughout the source in many files, even though the related coverage, or lack thereof, is highly dependent.

To make the task more manageable, and therefore more likely to be performed by users, we developed a hole analysis algorithm and tool that is based on common substrings in the names of functions. We tested its effectiveness using two large IBM software systems. In both of them, we asked domain experts to judge the results of several hole-ranking heuristics. They found that 57%–87% of the 30 top-ranked holes identified by the effective heuristics are relevant. Moreover, these holes are often unexpected. This is especially impressive because substring hole analysis relies only on the names of functions, whereas domain experts have a broad and deep understanding of the system.

We grounded our results in a theoretical framework that states desirable mathematical properties of hole ranking heuristics. The empirical results show that heuristics with these properties tend to perform better, and do so more consistently, than heuristics lacking them.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability

1. INTRODUCTION

Common code coverage analysis tools rely on the hierarchical structure of the source code. For example, such tools often support drilling down from directories through files and functions to lines of source code. Missing coverage, however, is frequently not aligned with the source code hierarchy. Rather, it may be aligned with a certain functionality that cuts across the predetermined hierarchy. One example is error handling. Error handling code is often evenly spread throughout the source code and typically represents just a small part of the source code. Thus, even if the error handling code is not covered at all, a hierarchical view of code coverage might obscure this fact. In addition, searching through the predetermined hierarchy is often tedious as the hierarchy might not divide the coverage tasks evenly. A **coverage task** consists of an identifier of the element to cover, such as a function name. The set of all coverage tasks indicates the full possible coverage. If all possible coverage tasks are covered, we say that there is 100% coverage.

Substring hole analysis aggregates coverage data across the structural code hierarchy. It looks for common substrings among names of source code elements, typically function names. Substring hole analysis is based on the observation that developers typically give semantically meaningful names to software source code elements. In addition, there often exist coding conventions that enhance the semantic commonality among names. Therefore, source code elements with similar names are often associated with a common topic, context, or functionality.

In a nutshell and very informally, substring hole analysis considers substrings as holes if they are common to multiple element names with poor coverage. However, finding these holes is challenging. First, identifying relevant substrings is tricky. There are many possible substrings, but they are often meaningless. Second, users are typically willing to examine only a few results, so effective ranking of holes is crucial. However, multiple factors influence the risk that a hole represents. The main factors we have identified are the number of coverage tasks in a hole and the percentage of coverage of a hole. The substring hole analysis algorithm and tool are challenged with collapsing these multiple dimensions into a linear order so a user can comprehend the data. Aggregating and presenting coverage information per substring provides the user with insight on aspects of the source code that lack coverage.

Substring hole analysis may be viewed as a post-processing stage that requires only generic coverage data. Therefore, it is independent of the software's programming language, platform, and coverage data collection tool. The substring hole analysis algorithm and tool discussed here is part of our coverage analysis tool FoCuS [5]. Further background and related work are presented in Section 2.

This paper provides the following main contributions:

- Novel algorithm and tool for substring hole analysis
- Methodology for effective code coverage analysis
- Mathematical framework for describing desired properties of hole-ranking heuristics
- Observations from running the tool on two IBM software systems and especially from comparing the hole-ranking heuristics

We empirically validate the effectiveness of our tool and hole-ranking heuristics. We show that there are effective ranking heuristics on two very different large IBM software systems. The effectiveness of the heuristics was determined by asking domain experts to judge which of the 30 top-ranked holes of each heuristic were indeed holes. There was generally an agreement among the experts, even across the software systems, regarding the effective heuristics. Moreover, the effective heuristics suggested holes that the experts were previously unaware of. We were able to empirically validate the importance of the main problem dimensions that we identified. Further, the effective heuristics hold the mathematical properties that we defined.

Section 3 describes the methodology we recommend for effective code coverage analysis. Section 4 details our substring hole analysis algorithm. The substring hole analysis algorithm has three major stages: identifying holes, ranking holes for display, and eliminating duplicate holes. This paper concentrates on the second stage, ranking holes, which is the most challenging stage. We compare different heuristics that we implemented for ranking holes, both mathematically and empirically. The mathematical framework we developed is described in Section 5. Section 6 details the empirical comparison of the heuristics. We conclude and discuss future work in Section 7.

2. BACKGROUND AND RELATED WORK

Our experience in code coverage and functional coverage [6] identifies the need to glean interesting information from large amounts of data. We have also shown [3] how holes are found in coverage data from functional coverage. Though the motivation is similar for data from code coverage, the input is different. In addition to information about coverage, functional coverage models are usually a high-dimensional cross product of attributes. On the other hand, code coverage data relies mainly on strings, typically function names. The simple methodology of looking at all the tasks and seeing which are uncovered works only if the number of tasks is manageable. In the software systems that we work with, this is often not the case even as early as extensive unit tests, and is certainly not the case during system tests. A different methodology is needed in system tests, where there may be millions of coverage tasks. One methodology of drilling down code hierarchies, already exists. However, functionality is not always distributed along this hierarchy. For example, code that handles specific platforms may be distributed over the entire application. Substring hole analysis provides a means to gain additional coverage information out of the long list of covered and uncovered tasks.

Substring hole analysis was done manually before we could automate the process. The automation required a deep understanding in order to identify the relevant features and come up with effective heuristics. This paper is about the new feature that was added to FoCuS as a result of our insight and heuristics. The paper concentrates on the algorithm and heuristics, and provides both theoretical and experimental results. We demonstrated the tool and its usage in previous work [1]. The novel automated analysis is superior to the manual one in two major aspects. First, it is much faster; instead of several expert hours, the analysis now takes minutes. Second, the quality of the report is higher and it is more complete. What was once "an art" has now became an engineering practice. Section 3 describes our methodology and Section 4 provides the algorithm we use for automated substring hole analysis.

Kim [7] looked at coverage results of very large systems with about 20 million lines of code. However, the reports are statistics on patterns of coverage and relationship to bugs, not hierarchical drill down reports with coverage information.

Our technique falls under the generic category of mining code coverage data. Unfortunately we have not been able to find previous work that falls under this category. Given that looking at system level code coverage data is very hard to do without efficient techniques, it is not surprising that code coverage is rarely used in practice.

From a theoretical viewpoint, the problem can be cast in the well-studied framework of *rank aggregation*. The rank aggregation problem, originally posed in social choice theory, seeks to reconcile opinions of multiple voters who cast votes on a finite set of candidates. In our case, the candidates are the holes, and the voters are either the different scoring functions or the problem dimensions. In particular, we focus on the rankings induced by the problem dimensions. These dimensions are the number of coverage tasks in a hole and the percentage of coverage of a hole. These are similar to the support and confidence measures that are used in creating association rules [2].

Formally, let there be n candidates, and let each "voter" be a permutation σ of all of the candidates, such that $\sigma(1)$ is the top (most important) candidate, and $\sigma(n)$ is the least important. Given several such permutations, the goal of the rank aggregation algorithm is to produce a reconciliation permutation τ that is optimal in some sense with respect to the input permutations. Several optimization criteria have been proposed[4]. Among them are the Kendall distance and Spearman's Footrule. The Kendall distance seeks to minimize the total number of pairwise swaps needed to convert the input permutations into the output permutation. Optimizing this measure is known to be NP-hard for four or more voters. In our case, where there are only two voters, the set of solutions degenerates to include the two input permutations, as well as any intermediate permutation in the bubble-sort path between them. Therefore, the Kendall distance is not useful for our problem. *Spearman's Footrule* seeks to minimize the absolute difference in ranks between the input and output permutations. This formulation can be solved in polynomial time, for any number of voters, by reducing the problem to minimum weight maximal matching in a bipartite graph. It can be shown that this is a factor-2 approximation to the Kendall optimal permutation. In addition, several heuristics have been proposed. One of the most popular is *Borda's* method. For each voter σ and candidate i, let $B_\sigma(i)$ be the number of candidates ranked below i. The Borda score for candidate i is the sum of the $B(i)$ over all of the voters. The candidates are then sorted by their Borda scores. Our Ranks heuristic (Section 4.4.3) is a generalization of the Borda score.

3. METHODOLOGY

The main goal of code coverage analysis is to increase the probability of finding bugs by improving the coverage of the system's test suite. Using the raw code coverage data to achieve this goal can be a difficult task, especially when the size of the data is very large. In many cases, the newly added tests improve the coverage but do not increase the probability of detecting new bugs. Substring hole analysis provides a list of uncovered holes that relate to a certain functionality of the system (e.g., error handling). Therefore, it helps in adding new tests or changing existing tests in a way that increases the coverage of the most significant code areas. To get the most out of substring hole analysis, we found it useful to make multiple rounds of analysis.

1. In the first round, we recommend finding holes on the program hierarchy. Consider the case where most of the coverage tasks in a substring hole belong to an uncovered node in the hierarchy. For example, we may have a hole with 50 files, all of which belong to a single uncovered directory. Such a hole is less informative than other holes because there is already information about the directory in the hierarchical coverage analysis. Now consider a substring hole that contains tasks belonging to many different hierarchical nodes, most of which are partially covered. This is an interesting cross concern hole. In our experience, cross concern holes identify holes in the coverage that cannot be found with the standard hierarchical coverage techniques and are, therefore, of added value. We recommend listing the hierarchy holes and then removing the coverage tasks that correspond to these holes.

2. In the next round, substring hole analysis is run on the remaining data. This typically reveals some holes that are well known to the user. For example, holes that are due to running on only a subset of the platforms supported by the software. Coverage tasks that belong to such "non-interesting" holes are then removed. This is straightforward to do with FoCuS.

3. A final round of hole analysis is then done over the remaining data. This often triggers less obvious holes to surface.

The tool has various thresholds. We believe that the default parameter values work with the exception of one parameter. Section 4.3.1 discusses the tool parameters and their recommended values.

4. HOLE ANALYSIS ALGORITHM

This section describes the substring hole analysis algorithm in its entirety. Section 4.1 defines basic hole analysis terms. Section 4.2 provides a high level description of the algorithm. Section 4.3 provides further details about our implementation. Section 4.4 describes the different ranking heuristics that we support.

4.1 Definitions

A **coverage task** is identified by a string. A coverage task is covered if the code segment that it represents is executed. The input to the substring hole analysis algorithm is a set of coverage tasks presented as character strings and the number of times that each task was covered. The coverage tasks are usually names of functions. For initial reports, it is sufficient to have a Boolean value for each task indicating whether it was covered. An example of a single coverage task is "Exception.firm.io, 0". A **hole** is a set of coverage tasks, of which at least one is not covered, that have a common substring. This substring identifies the hole.

The output of the substring hole analysis algorithm is a ranked set of holes along with coverage data for each hole. For example, if a line of output in the coverage report for functions is "Exception, 912, 907", it means that there are 912 functions whose names contain the Exception substring, 907 of which were not covered.

We use the following terms in our description of the ranking heuristics (Section 4.4). **Uncovered**: number of not covered tasks in the hole. **Covered**: number of covered tasks in the hole. **Total**: Covered plus uncovered. Total is the number of tasks in the hole. **Length**: length of the common substring.

4.2 High Level Algorithm

Our algorithm has three major stages. The first stage finds all the relevant holes that conform to the requirements on the string length, on the hole's size, and on the percentage of uncovered functions in a hole (See Section 4.3.1 below for a description of these requirements). The names of the tasks are analyzed and relevant substrings are determined. Section 4.2.1 describes the current implementation of determining substrings. Each selected string corresponds to a set of tasks whose names contain that string.

In the second stage, the strings are ordered according to quality or concern about leaving the hole uncovered. There are a number of ordering functions that were suggested but in all of them the quality is a function of the size of the set (larger is better), the percentage of uncovered tasks (larger is better), and the properties of the string (some prefer long, some short, and some delimited by capital letters). The result of this step is an ordered list of strings, each representing a hole.

In the third stage, the holes are visited according to the order determined in the second step. The tasks in every hole are compared to those in the holes before it. If too many of the tasks (depending on a threshold) are contained in the previous holes then the hole is discarded; otherwise it is added to the holes for display. Once the algorithm reaches the desired number of holes for display, it is done.

When the algorithm completes, the relevant holes are displayed to the user. The display contains the names of each hole and, depending on the user's choice, may also show the relevant tasks.

4.2.1 Identifying Holes

Identifying holes means defining which substrings to examine. Naively, all possible substrings would be considered. However, we find it effective to prefer strings that have a semantic meaning. This heuristic eliminates noise for the user and improves the tool's performance. Fortunately, most programming styles make a distinction between words in names, either by introducing a delimiter such as '_', or by changing the letter case. The following rules define substrings to consider:

1. Characters are classified as digits, delimiters, uppercase letters, and lowercase letters.
2. Scanning a string from the left, a new substring starts whenever the characters change class.
3. An exception to item 2 is that the last character of a sequence of uppercase letters moves to the beginning of the next substring if it starts with a lowercase letter.

Compared to the naive approach, this approach has the advantage of reducing computation time by avoiding overlapping substrings. The tool also supports a single wildcard in the substring. We found this useful for detecting holes in operations that were uncovered by one component but not by another. For example, "Cache*Callback" functions may be covered, but "Disk*Callback" functions may be uncovered. If "Disk", "Cache", and "Callback" by themselves are not holes that will be ranked highly, two substrings are needed. Our experiments did not show the need to use more than a single wildcard.

4.3 Detailed Description

The actual implementation of our algorithm is slightly different due to performance consideration. Section 4.3.1 provides details about the parameters of our algorithm implementation, including their default values.

1. Identify holes—find substrings and populate the appropriate data structure. For all coverage tasks (tuples of function name and coverage) DO:
   (a) Get the next function name
   (b) Break the name into semantic substrings
   (c) Cull all substrings that are less than the minimal length parameter and greater than the maximal length parameter
   (d) Add substrings with a single wildcard
   (e) For each of the remaining substrings DO
      i. Add a hole identified by the substring to a collective data structure
      ii. Scan all the coverage task strings and update the covered and uncovered values of this hole
2. Rank holes
   (a) Cull all holes whose total value (number of functions in a hole) is less than the minimal size parameter or whose uncovered percentage is less than the hole percentage parameter
   (b) Cull all holes whose substring is a substring of other holes and contain exactly the same coverage tasks
   (c) Cull all holes whose coverage tasks are contained in another hole and have equal or larger coverage percentage compared to the other hole
   (d) Sort the holes according to the ranking heuristic chosen by the user (Section 4.4 details the ranking heuristics we compared in our experiments)
3. Eliminate duplicate holes
   (a) Cull all holes that are different from the union of all previous holes by less than the similarity percentage parameter. If B is the current hole and Union is the union of previous holes, then the difference $|B_{nc}| - Union$ is the uncovered occurrences and the percentage is
   $$100 \cdot (|B_{nc}| - Union)/|B_{nc}|$$
4. Display the top holes by order to the user

4.3.1 Hole Analysis Parameters

Our hole analysis tool defines parameters that serve as thresholds throughout the hole analysis process. In our experiments, we use the default parameter values and change only the ranking methods. In general, a user may choose to change parameter values. Following is a short description of these parameters.

- Minimal length and maximal length. Upper and lower limits on the length of the substring that embodies the hole. Usually the minimum is between four and six, as lower than that may not be a meaningful name. If the naming conventions include acronyms, it may be the case that fewer characters are meaningful. In our experiments, the lower limit is 4 and the upper limit is 30.
- Minimal size. Minimal size of a hole, i.e., the number of functions that contain this substring. In our experiments this threshold is ten. Therefore, the tool does not consider any substring that belongs to less than ten functions.
- Hole percentage. Percentage of the uncovered functions. We have different opinions and the number chosen is between 70% and 95%. The lower the percentage, the more holes the tool displays. When the overall coverage is high, the percentage of uncovered functions should be low (70%) or there are likely to be very few holes. When the overall coverage is low, it is advisable to start with a high value (95%) for the percentage of uncovered functions. In our experiments, the percentage of uncovered functions is 95% for the dataset that has low coverage (BinaryManip) and 70% for the dataset that has high coverage (Driver). In general, one option is to set the percentage of uncovered functions in a hole to be higher than the overall coverage percentage. The rational behind this option is to surface areas that are poorly covered compared to the coverage of the environment. However, there are merits to choosing relatively low numbers, as you get larger holes, and to choosing relatively high numbers,

as you get "purer" holes. There are different preference in our team and among our users so we do not have firm guidelines.

- Similarity percentage. Determines what holes will be presented to the user. We have this parameter fixed at 50. This means that holes are presented to the user only if they contain more than 50% new uncovered coverage tasks compared to the union of all holes that were already chosen for display.

Holes that pass the above thresholds are sorted according to one of the ranking methods described in Section 4.4.

4.4 Methods for Ranking Holes

Typically, users only consider the first few holes, so the order of presentation is very important. This is non trivial: Is a hole of 900 out of 1000 more important than 100 out of 100? When "Exception" defines a hole of 500 out of 520 and "Exception.firm" defines a hole of 450 out of 450, which of them do you show to the user (further, assume that the string "Exception.soft" has coverage of 50 out of 70)?

The following parameters should be taken into account when ranking holes:

- The number of tasks the hole represents
- The percentage of uncovered tasks
- The existence of similar holes
- Whether the hole cuts across the structured hierarchy of the program
- The length of the substring that represents the hole
- Whether it is possible to detect semantically meaningful substrings (see Section 4.2.1)

Different users seem to have different preferences. Therefore, we have come up with a number of ranking heuristics. The emphasis that these heuristics put on the different problem dimensions varies. We describe each of these heuristics in the subsequent section. In Sections 5 and 6 we compare these heuristics mathematically and experimentally, respectively.

4.4.1 lgCov

The lgCov measure is based on the ratio between covered and uncovered.

$$lgCov = \begin{cases} -\infty & \text{if Covered=0} \\ \infty & \text{if Uncovered=0} \\ \log(covered/uncovered) & \text{Otherwise} \end{cases}$$

The range of the lgCov score is $-\infty \leq lgCov \leq \infty$. As lgCov approaches infinity/-infinity the coverage of the hole is higher/lower, respectively. Intuitively, holes with large negative lgCov are poorly covered, hence lgCov represents the coverage of the hole. Notice that lgCov does not take into account the size of the hole.

4.4.2 sqCov

The sqCov measure is based on a combined measure between the absolute number of uncovered tasks and the percentage of uncovered tasks in a hole.

$$sqCov = uncovered/\sqrt{total}$$

The range of the sqCov score is $0 \leq sqCov \leq \infty$. We intend to explain the rationale behind sqCov in future work. Intuitively, sqCov takes into account both the size of the hole and its coverage by multiplying uncovered (relates to size) and *uncovered/total* (relates to the coverage). However, since uncovered is not limited while *uncovered/total* is limited to $[0, 1]$ sqCov "favors" large holes (e.g., $1/\sqrt{1} = 1$ is smaller than $2/\sqrt{3} = 1.15$).

4.4.3 Ranks

The Ranks heuristic performs rank aggregation, inspired by social choice theory, between Uncovered/Covered and Uncovered (see Section 2 for a discussion of rank aggregation). This kind of treatment is scale-free, in the sense that the actual values of the coverage counters are irrelevant—only their relative order is important. This is a convenient way to avoid using scale factors and to reduce the effect of extreme outliers.

Think of Uncovered and Uncovered/Covered as distinct "voters", each ranking the holes. We now try to produce some kind of merged ranking. The gold standard of such an aggregation is optimization of the Kendall distance. But it can be shown that in the case of two voters, the optimum is achieved at either constituent ranking, as well as at any point in the bubble-sort path between them. Therefore we tried an adaptation of the popular Borda method. For two rankings, r and s, this means sorting each element x by the value of $r(x)+s(x)$ (or, equivalently, by the negation of this, depending on the desired direction of sort).

Specifically, if there is a hole that ranks third on both measures, then its combined score is equivalent to one that ranks first on one measure and fifth on the other. We felt this inappropriate and fixed the score to give preference to holes that ranked very high on either measure. Consequently, the contour lines of this function, when drawn on a plane, would be curved, rather than straight. See Figure 1.

We also added a weight parameter to formalize the intuitive sentiment that one of the measures is considered more important than the other.

Precisely, let the rank be an integer in the range $[1, N]$ where 1 is the "best" item and N is the "worst". Let $r_{nc}(i)$ be the rank of the i^{th} element when the elements are sorted by Uncovered in descending order, and let $r_{ratio}(i)$ be the rank of the i^{th} element when the elements are sorted by Uncovered/Covered in descending order. Then sort by

$$x(i) = a \cdot r_{nc}(i)^{1/curv} + r_{ratio}(i)^{1/curv}$$

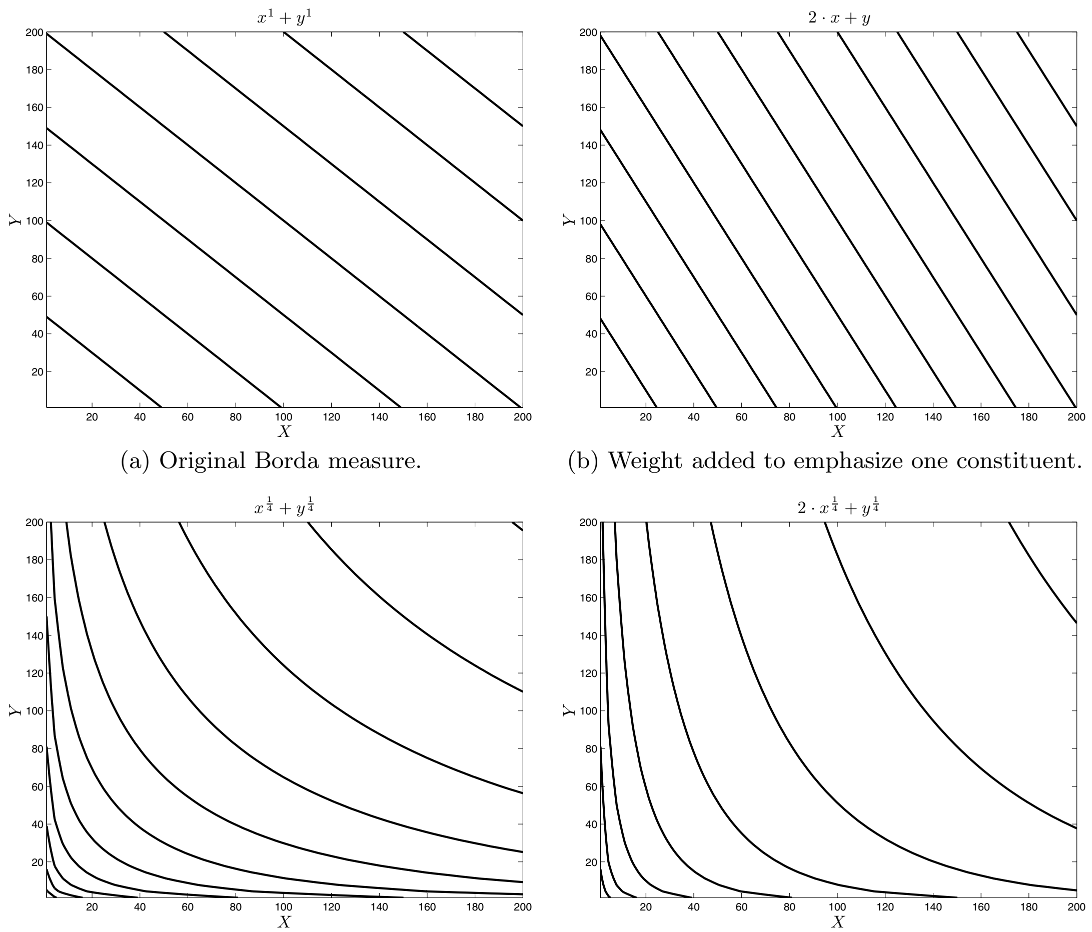
where $a > 0$ and $curv > 0$ are the respective weight and curvature parameters. Note that the smaller $x(i)$ is, the higher the concern of the hole.

4.4.4 Strawman

A naive heuristic can take into account only Uncovered. Uncovered favors the hole with the highest number of uncovered functions, regardless of the total number of functions in the hole. We add this heuristic as a strawman in our experiments.

5. A FORMAL ANALYSIS OF THE CONCERN ABOUT HOLES

A set of coverage tasks C is given and a coverage function $cov : C \rightarrow \{0, 1\}$, such that, $\forall c \in C$, $cov(c) = 1$ iff c is



(a) Original Borda measure.

(b) Weight added to emphasize one constituent.

(c) Curvature added to emphasize top ranking in either constituent.

(d) Curvature and weighing combined.

Figure 1: Contour lines for the modified Borda measure used for rank aggregation. For two constituent orders, each element is placed on the plane by treating its two respective ranks as two dimensions. Under this formulation, the contour lines show a set of pairs of ranks that will receive the same combined score.

covered. For a given subset $A \subseteq C$, define

$$A_c = \{c \in A | cov(c) = 1\}$$

and

$$A_{nc} = \{c \in A | cov(c) = 0\}$$

. Clearly, the larger A_{nc} is, the more concern one should have about the test coverage of A.

In the following paragraphs we elaborate on the relative ranking of coverage task subsets with respect to the concern about their test coverage. The following definitions are used:

1. A set of coverage tasks, that have a common substring, is called a **hole** if it contains at least one uncovered coverage task.
2. The term **concern** is used to reflect our estimate of the danger in leaving the hole uncovered.
3. Two or more holes are called **disjoint holes** if they have no common coverage task.
4. Given two subsets $A, B \subseteq C$, we define the relation $A > B$ to mean that we are more concerned about A than about B.

There are several commonsense observations regarding the concern about a hole. For a given hole A, $A \subset C$:

1. Adding an uncovered coverage task $c \in C$ to A increases our concern about the hole.
2. Adding a covered task $c \in C$ to A decreases our concern.
3. A hole that can be constructed from several disjoint holes, has a higher concern than its least concerning constituent. In other words, if $A = B \cup C \cup D$ and B, C, D are disjoint, then the concern about A is higher than at least one of the concerns about B, C, or D.
4. If $A > B$ then $B > A$ cannot hold (anti-symmetry).

These observations imply a binary anti-symmetric relation on holes $>$. Next, we give a more formal definition of the observations above. These observations can serve as restrictions on the implementation of $>$. A discussion of the anti-symmetry property is beyond the scope of this paper.

While it is hard to provide a complete definition of $>$, these rules capture most of the common sense intuition stated above:

1. If $|A| = |B|$, and $|A_{nc}| = |B_{nc}|$, then $A \sim B$, where $\sim$ denotes equal concern.
2. If $|A| = |B|$, and $|A_{nc}| > |B_{nc}|$, then $A > B$.
3. If $|A| > |B|$, and $|A_{nc}| = |B_{nc}|$, then $A < B$.
4. If there exist $d > 1$, such that $|A| = d \cdot |B|$, and $|A_{nc}| = d \cdot |B_{nc}|$, then $A > B$. Notes:
   (a) Rule 4 is a weaker case of the third common sense rule. It is derived from the special case in which all the constituents have the same concern measure.
   (b) Combining 2 and 4 results in
5. If there exist $d > 1$, such that $|A| = d \cdot |B|$, and $|A_{nc}| >= d \cdot |B_{nc}|$, then $A > B$.

We now define the binary relation $>$ as the transitive closure of the above restrictions.

5.1 Quantifying the Concern about a Hole

For the coverage instance (C, c) where C is a set of coverage tasks and $c :\rightarrow 0, 1$ is the coverage function, a measure of concern is a function $m : 2^C \rightarrow R$. By convention, for a given hole $H \subseteq C$, $m(H)$ is bigger when we have more concern about the hole.

We next consider several possible measures and check if they agree with the partial relation $>$ defined above.

5.1.1 *The Uncovered Tasks Concern Measure*

A simple measure of concern is the number of uncovered tasks in the hole, i.e., $m_{nc}(A) = |A_{nc}|$. This measure is clearly consistent with rules 1, 2, and 4, but not with 3, since the measure value does not change if the number of covered tasks changes. Thus, the order induced by m_{nc} is inconsistent with $>$. This measure corresponds to the Strawman heuristic in Section 4.4.4.

5.1.2 *The Ratio Concern Measure*

Another simple, almost natural, measure of concern is the ratio of uncovered tasks in the hole to the total number of tasks in the hole, i.e., $m_r(A) = |A_{nc}|/|A|$. This measure is clearly consistent with rules 1, 2, and 3, but not with 4, since the measure value does not change if the ratio remains unchanged. Thus, the order induced by m_r is inconsistent with $>$. This measure corresponds to the lgCov heuristic in Section 4.4.1.

5.1.3 *Using the Square Root Function to Decrease the Importance of the Hole Size*

Given a hole $H \subseteq C$, s.t. $|H_{nc}| > 0$, consider the confidence measure $m_{sq}(H) = \frac{|H_{nc}|}{\sqrt{|H|}}$.

This measure is clearly consistent with rule 1. To check if it agrees with $>$, we need to check what happens in the following three cases:

1. A covered task is added to a hole,
2. An uncovered task is added to a hole, and
3. Both $|H|$ and $|H_{nc}|$ are multiplied by some $d > 1$.

In the first case we have $B \setminus A = \{x\}, A, B \subseteq C$ and $x \in C$ and $c(x) = 1$. In this case $m_{sq}(B) = \frac{|B_{nc}|}{\sqrt{|B|}} = \frac{|A_{nc}|}{\sqrt{|A|+1}}$ whereas $m_{sq}(A) = \frac{|A_{nc}|}{\sqrt{|A|}}$. Clearly, $m_{sq}(B) < m_{sq}(A)$, as required.

In the second case we have $B \setminus A = \{x\}, A, B \subseteq C$ and $x \in C$ and $c(x) = 0$. In this case $m_{sq}(B) = \frac{|B_{nc}|}{\sqrt{|B|}} = \frac{|B_{nc}|+1}{\sqrt{|A|+1}}$.

We next show that $m_{sq}(B) > m_{sq}(A)$ in agreement with $>$. To this end we show that $f(x) = \frac{x+1}{\sqrt{n+1}} - \frac{x}{\sqrt{n}}, x \in [0, n]$ is positive. Indeed, $f(0) = \frac{1}{n+1} > 0$ and $f(n) = \frac{n+1}{\sqrt{n+1}} - \frac{n}{\sqrt{n}} = \sqrt{n+1} - \sqrt{n} > 0$. In addition, $f'(x) = \frac{1}{\sqrt{n+1}} - \frac{1}{\sqrt{n}} < 0$, thus $f'(x)$ is a monotonic function in $[0, n]$, and is thus positive in $[0, n]$.

The third requirement is also met. $\frac{d}{\sqrt{d}} > 1$ for $d > 1$, so $\frac{d \cdot |H_c|}{\sqrt{d \cdot |H|}} = \frac{d}{\sqrt{d}} \cdot \frac{|H_c|}{\sqrt{|H|}} > \frac{|H_c|}{\sqrt{|H|}}$

This measure, is, therefore, consistent with $>$. This measure corresponds to the sqCov heuristic in Section 4.4.2.

5.1.4 *Rank Based Measure*

Let S be a collection of S_i s.t. $S_i \subseteq C$.

For any $A \in S$, let $r_{nc}(A)$ be the rank of A when S is sorted by $|S_{i,nc}|$ in descending order and let $r_{ratio}(A)$ be the rank of A when S is sorted by $\frac{|S_{i,nc}|}{|S_{i,c}|}$ in descending order.

Then sort S by $x(A) = a \cdot r_{nc}(A)^{1/curv} + r_{ratio}(A)^{1/curv}$, where $a > 0$ and $curv > 0$ are pre-selected parameters. Notice that the smaller $x(i)$, the higher the concern of the hole.

If $|A| = |B|$ and $|A_{nc}| = |B_{nc}|$ then also $|A_c| = |B_c|$, therefore this measure is clearly consistent with rule 1.

Since $a > 0$, rule 2 is satisfied, in the sense that if a subset B such that $|A| = |B|$, and $|A_{nc}| > |B_{nc}|$ is added to S then $r_{nc}(A) < r_{nc}(B)$ and $r_{ratio}(A) < r_{ratio}(B)$ hence $x(A) < x(B)$ and A's concern is higher than B's.

Rule 3 is satisfied, in the sense that if a subset B such that $|A| > |B|$, and $|A_{nc}| = |B_{nc}|$ is added to S, then $r_{nc}(A) = r_{nc}(B)$ and $r_{ratio}(A) > r_{ratio}(B)$, hence $x(A) > x(B)$ and A's concern is smaller than B's.

Rule 4 is satisfied, in the sense that if a subset B is added to S and there exist $d > 1$, such that $|A| = d \cdot |B|$, and $|A_{nc}| = d \cdot |B_{nc}|$, then also $|A_c| = |A| - |A_{nc}| = d \cdot |B| - d \cdot |B_{nc}| = d \cdot (|B| - |B_{nc}|) = d \cdot |B_c|$. Therefore, $r_{nc}(A) < r_{nc}(B)$ and $r_{ratio}(B) = r_{ratio}(A)$ hence $x(A) < x(B)$ and A's concern is higher than B's.

This measure corresponds to the Ranks heuristic in Section 4.4.3.

6. EXPERIMENTS AND RESULTS

In this section, we empirically compare the effectiveness of the ranking heuristics that we implemented. In Section 6.1 we provide details about the experimental settings. In Section 6.2 we list the experimental results and provide an analysis for these results.

6.1 Experimental Settings

We describe the two software systems that we ran in our experiments and provide details about their coverage data in Section 6.1.1. The experiments were run according to our recommended methodology. The parameter values and heuristics we ran are detailed in Section 6.1.2. The results were determined with the assistance of domain experts. The

	Total	Covered	Coverage Percentage
Directories	164	75	45.7
Files	852	401	47.1
Functions	50032	10268	20.5

Table 1: Code size and coverage for BinaryManip

	Total	Covered	Coverage Percentage
Directories	23	22	95.7
Files	497	442	88.9
Functions	11417	8241	72.2

Table 2: Code size and coverage for Driver

experimental method is discussed in Section 6.1.3. Major threats to the validity of our experiments are discussed in section 6.1.4.

6.1.1 Data

We ran the collection of hole analysis heuristics listed below on coverage data from two large IBM software systems. Both systems have been used extensively by many users. These systems are very different from one another. One system, referred to as **BinaryManip**, is a tool that performs various manipulations on code binaries. It is written in C++ and, as Table 1 lists, has more than 50,000 functions. The software is multi-platform. The test suite that was used for measuring coverage executes only on a subset of these platforms. The code coverage of this test suite is low, as shown in Table 1. The second system, referred to as **Driver**, is the kernel code of a high-end device driver. It is written in C and, as Table 2 lists, has over 11,000 functions. The test suite that was used for measuring coverage executes on a simulator and has relatively high code coverage as shown in Table 2.

6.1.2 Parameters and Heuristics

All the heuristics were run with the default algorithm parameter values detailed in Section 4.3.1, with an exception for the uncovered functions parameter. The uncovered functions parameter was set to 95% for BinaryManip and to 70% for StroageDriver. This is a result of the overall code coverage for these systems. When the code coverage is high we need to decrease this parameter value in order to get a decent number of holes. This parameter is ignored for the Strawman heuristic, since that heuristic takes into account only the number of uncovered tasks. The left column of Table 4 details the ranking heuristics that we compare, including the operation of each heuristic in case of equality. For example, the ranking heuristic in the first row is lgCov. Using lgCov, holes are sorted by lgCov in ascending order. If the lgCov score is equal, holes are sorted by Total in descending order. If Total is also equal, holes are sorted by Length in descending order.

6.1.3 Method

We were fortunate to have domain experts evaluate the ranking of the ranking heuristics. One domain expert evaluated the rankings for BinaryManip and two domain experts evaluated the rankings for Driver. For the evaluation, we took the top 30 substrings (holes according to the heuristics)

BinaryManip: Substring	Total	Uncovered
::PartialCFG::ACyclicSimulator	19	19
::PartialCFG::ProgBased::PPC::	100	100
::Pipeline::ProgBased::DFATest	61	61
::Pipeline::ProgBased::Example	23	23
Driver: Substring	Total	Uncovered
Bifyl	58	17
BuildSense	81	59
Device	222	52
DeviceImage	12	12

Table 3: Example of substrings given to the experts. For each substring, our tool also provides a list of covered and uncovered functions

from each of the ranking heuristics and created a merged list sorted alphabetically. Table 3 shows a small example of input that was given to the experts. The experts then marked the substrings that, in their opinion, are significant holes in this list. They also indicated the substrings they found especially interesting. We then gave scores to each of the ranking heuristics as follows. The initial score for all heuristics is 0. For each substring that the expert marked as a hole, if that substring appears among the top 30 holes of the heuristic, the score of that heuristic increases by one. Therefore, scores are in the range [0–30]. The higher the score, the better the heuristic is according to that expert.

6.1.4 Threats to Validity

Our experiments included only two software systems and three domain experts. Though these are real and complex systems, clearly, this sample is not large enough to make our results statistically valid. We leave the complete statistical validation of the results for future work, after more data is collected. However, we believe that these initial results provide a good indication that our technique is useful. Another threat to the validity of the results might be the fact that the experts were only asked to evaluate the top ranked holes found by the tool. It is possible that significant holes were missed by the tool or got a low rank and therefore were not evaluated by the experts. Although we didn't address this threat directly in our experiments, it is worth mentioning here that when we asked the experts to look at the raw coverage data, they failed to find additional significant holes due to the large size of the data.

6.2 Results and Analysis

Table 4 summarizes the scores that the ranking heuristics got from each of the experts. The BinaryManip column provides the scores that result from the evaluation of the BinaryManip expert. Similarly, the Driver1 and Driver2 columns provide the scores that result from the evaluation of the first and second Driver experts, respectively. Although we only have two data sets, we believe that we can draw representative conclusions from the results on these data sets. This is because these data sets are from very large software systems and because these systems are significantly different from one another. It is especially encouraging to notice that there is generally agreement among the experts regarding the top performing heuristics, even across the two data sets. This suggests that the results are indeed general. The agreement between the two Driver experts is high. They gave the same score for more than 80% of the holes. Therefore, there

Ranking Heuristic	Binary Manip	Driver1	Driver
lgCov $lgCov \rightarrow Total \rightarrow Length$	3	25	20
sqCov $sqCov \rightarrow Total \rightarrow Length$	17	25	23
Ranks0.5a $Ranks \rightarrow Length$ $curv = 0.5, a = 0.5$	3	24	19
Ranks2a $Ranks \rightarrow Length$ $curv = 0.5, a = 2$	12	25	22
Uncovered $Uncovered \rightarrow Length$	16	26	23
Strawman $Uncovered \rightarrow Length$ (percentage ignored)	14	14	10

Table 4: Experts' scores for ranking heuristics. The maximal possible score is 30. The uncovered functions parameter was 95% for BinaryManip and 70% for Driver, with the exception of the Strawman, for which it was zero.

was no need to further resolve their scores to get a better agreement.

It is clear from these scores that the experts prefer large holes. sqCov does well on both data sets. Surprisingly, Uncovered performs similarly well on these data sets. However, we see that Strawman consistently performs poorly. The difference between Uncovered and Strawman is that Uncovered ranks only those holes that have at least a minimal coverage percentage as determined by the uncovered functions parameter. Strawman, on the other hand, ranks all holes. In other words, Uncovered takes into account not only the absolute number of uncovered tasks in a hole but also, indirectly, their percentage. This confirms our initial analysis regarding the importance of these two dimensions for ranking holes.

We also prefer sqCov to Uncovered because sqCov has the mathematical properties we defined in Section 5 while Uncovered does not. The experimental results provide an indication that these mathematical properties are indeed relevant and helpful. Heuristics that possess the desired properties perform better and do so in a more consistent manner (i.e., perform well over multiple data sets) than those that do not. For example, lgCov performs poorly on the BinaryManip data set and performs better but not as well as sqCov on the Driver data set. This is explained by our theoretical framework. LgCov does not hold the desired properties for our problem domain.

It seems that Ranks could be made comparable to sqCov by changing the weight given to the size of the hole. This can easily be done by changing a. We plan to investigate this further.

It is interesting that the deviation of scores is larger for BinaryManip than for Driver. This can be explained by the difference in the overall coverage percentage. Because the overall coverage percentage for BinaryManip is much lower than for Driver there are many more candidate substrings. Naturally, the resulting overlap among the heuristics is smaller.

We asked the experts to not only mark for each substring whether it is a hole, but to indicate holes that are especially interesting to them. Interesting in this context means that the experts were unaware of this aspect of the system being poorly covered. This gave us a qualitative indication for the usefulness and effectiveness of our hole analysis tool. The tool is especially effective because it finds holes that domain experts are unaware of. Notice that the tool uses only function names and coverage data. Domain experts have a much wider and deeper knowledge and understanding of the system, including knowledge about the system's test suite. Humans, however, are not good at analyzing large quantities of data. It is very interesting to note that all domain experts indicated as interesting holes that were detected by sqCov. This is an additional confirmation for the good performance of sqCov. The number of interesting holes varies among experts. This is to be expected, because it is highly subjective and depends on the knowledge and understanding of the system. The BinaryManip expert marked 19 holes as interesting, one Driver expert marked 42 holes as interesting and the other marked 30 as interesting.

7. CONCLUSIONS AND FUTURE WORK

Substring hole analysis is an effective way of viewing and analyzing large quantities of code coverage data. We developed a hole analysis algorithm and tool, as well as a theoretical framework for comparing hole-ranking heuristics. We ran experiments over two large IBM software systems that are very different in nature from one another. For each of these systems, we evaluated our ranking heuristics according to input from domain experts. The domain experts agree on the effective heuristic—sqCov. Other heuristics that have the mathematical properties we defined and the capability to weight the problem dimensions, such as Ranks, are also effective. Furthermore, the heuristics provided information about missing coverage of which the experts were previously unaware of. Our theoretical framework supports the experimental results. Heuristics with the desired mathematical properties performed better and more consistently in practice. The experimental results also provide empirical confirmation regarding the main dimensions of the substring hole analysis problem. These dimensions are the absolute size of the hole and the coverage percentage in the hole. Our experiments show that users prefer big holes, but the coverage percentage must be taken into account as well.

We intend to implement the improvements suggested by the experiments to our substring hole analysis tool. For example, it seems possible to significantly reduce or even eliminate the users' need to select an effective ranking heuristic. The experiments reveal that we should be able to provide an effective single ranking heuristic. By effective we mean that it is flexible and able to provide a ranking of holes that different users find useful.

This paper describes experiments from two pilot projects for substring hole analysis. We are in the process of conducting two additional pilot projects, one on a large Java middleware product and one on a large firmware project. The preliminary results are similar to our conclusions in this paper.

Recently, we added another dimension to our reports using data automatically collected from version control. It is intuitively obvious that a hole representing functions that have been recently modified is more important than one representing functions that have not been changed since the previous version. This is due to the well-known correlation between code changes and the likelihood of bugs [8]. Currently, we simply mark holes as representing changed or unchanged code. However, we would like to factor in the degree to which the code has been modified.

The theoretical framework we defined received initial empirical confirmation from the experiments. We intend to continue to develop this framework, including a possible ranking based on the framework. In addition, the mathematical properties of the framework rules need to be further investigated and proven.

8. ACKNOWLEDGMENTS

We greatly thank Shachar Fienblit for fruitful discussions of various aspects of hole analysis. We thank Shachar and Ran Fashchik for their help in the experimental evaluation of the hole analysis algorithm.

9. REFERENCES

[1] Y. Adler, E. Farchi, M. Klausner, D. Pelleg, O. Raz, M. Shochat, S. Ur, and A. Zlotnick. Automated substring hole analysis. In *ICSE, NIER Track*, 2009.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[3] H. Azatchi, L. Fournier, E. Marcus, S. Ur, A. Ziv, and K. Zohar. Advanced analysis techniques for cross-product coverage. *IEEE Trans. Computers*, 55(11):1367–1379, 2006.

[4] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 613–622, New York, NY, USA, 2001. ACM.

[5] Focus code and functional coverage tool. `http://www.alphaworks.ibm.com/tech/focus.` Accessed December 2008.

[6] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *DAC*, pages 158–163, 1998.

[7] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, 2003.

[8] T. Zimmermann, N. Nagappan, and A. Zeller. *Predicting Bugs from History*, chapter 4, pages 69–88. Springer, March 2008.