DebugAdvisor: A Recommender System for Debugging

B. Ashok
Microsoft Research India
bash@microsoft.com

Joseph Joy
Microsoft Research India
josephj@microsoft.com

Hongkang Liang
Microsoft, Redmond
hliang@microsoft.com

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

Gopal Srinivasa
Microsoft Research India
gopalsr@microsoft.com

Vipindeep Vangala
Microsoft India, Hyderabad
vipinv@microsoft.com

ABSTRACT

In large software development projects, when a programmer is assigned a bug to fix, she typically spends a lot of time searching (in an ad-hoc manner) for instances from the past where similar bugs have been debugged, analyzed and resolved. Systematic search tools that allow the programmer to express the context of the current bug, and search through diverse data repositories associated with large projects can greatly improve the productivity of debugging. This paper presents the design, implementation and experience from such a search tool called DebugAdvisor.

The context of a bug includes all the information a programmer has about the bug, including natural language text, textual rendering of core dumps, debugger output etc. Our key insight is to allow the programmer to collate this entire context as a query to search for related information. Thus, DebugAdvisor allows the programmer to search using a *fat query*, which could be kilobytes of structured and unstructured data describing the contextual information for the current bug. Information retrieval in the presence of fat queries and variegated data repositories, all of which contain a mix of structured and unstructured data is a challenging problem. We present novel ideas to solve this problem.

We have deployed DebugAdvisor to over 100 users inside Microsoft. In addition to standard metrics such as precision and recall, we present extensive qualitative and quantitative feedback from our users.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Experimentation

Keywords

Recommendation systems, Search, Debugging

1. INTRODUCTION

Debugging large software is difficult. Large systems have several tens of millions of lines of code. No single person understands all the code in such systems, and often times new hires, who have little or no context about the code, are given the job of debugging failures. Our recent study [6] with Microsoft's Windows Serviceability group revealed that developers and testers spend significant time during diagnosis looking for similar issues that have been resolved in the past. They search through bug databases, articles from the on-line Microsoft Developer Network (known as MSDN), email threads, and talk to colleagues to find this information. For example, the same bug or a very similar bug may have been encountered and fixed in another code branch, and the programmer would greatly benefit from knowing this information. Consequently, we decided to build a recommender system to improve productivity of debugging by automating the search for similar issues from the past. Prior work in mining software repositories such as Hipikat [9, 10], eRose [24], and Fran [19] provides us inspiration and very useful ideas. However, there are important challenges in building a recommender system for debugging.

Challenges. The first challenge involves understanding what constitutes a query. In principle, we would like to leverage all of the context the user has on the current problem, including the state of the machine being debugged, information in the current bug report, information obtained from the user's interaction with the debugger, etc.

The second challenge involves dealing with the diversity of information sources that contain potentially useful information for the user. These include past bug reports, logs of interactive debugger sessions, information on related source code changes, and information about people who can be consulted. These information sources are variegated, and the data is of varied type —a mixture of structured and unstructured data. One approach is to ignore all the structure in the data, and use full text search to index and search through the data. This approach has the advantage that we can reuse existing indexing and searching tools. However, as our empirical evidence shows, there is much room for improvement in the quality of retrieved results.

Approach. We approach the first challenge (capturing the context of the user) by allowing a *fat query* —a query which could be kilobytes of structured and unstructured data containing all contextual information for the issue being debugged, including natural language text, textual rendering of core dumps, debugger output etc. We have built a tool

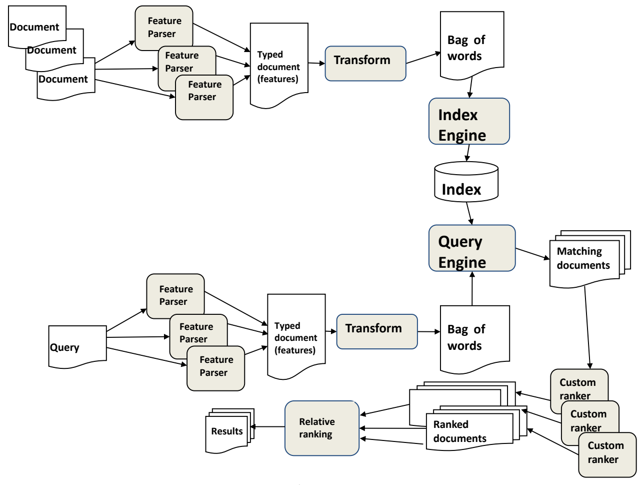

Figure 1: DebugAdvisor's first phase

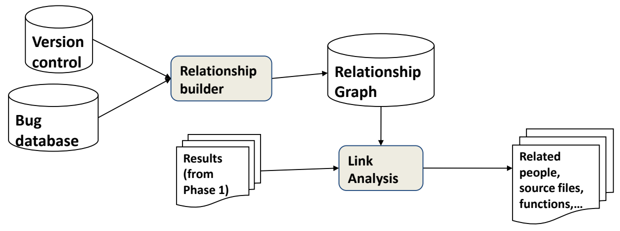

Figure 2: DebugAdvisor's second phase

called DebugAdvisor, which allows users to search through all our software repositories (version control, bug database, logs of debugger sessions, etc) using a fat query interface.

The fat query interface is quite different when compared with short query strings commonly used in information retrieval systems. Previously, our users had to search through each data repository separately using several queries, each of which was restricted to a short query string, and they had no automatic mechanism to combine these search results. DebugAdvisor's fat query interface allows users to query all these diverse data repositories with a single query.[1]

We approach the second challenge (diversity of information sources) by partitioning the search problem into two phases. The first "search" phase (see Figure 1) takes a fat query as input and returns a ranked list of bug descriptions that match the query. These bug descriptions (referred to as "documents" in Figure 1) contain a mix of structured and unstructured data (see example in Figure 3). The second "related-information" phase (see Figure 2) uses the output of the first phase to retrieve a set or related recommendations such as people, source files and functions.

The key idea in the first phase is to uniformly represent both queries and information sources as a collection of *features*, which are formalized as *typed documents*. Typed documents have a recursive type structure with four type constructors:[2](1) unordered bag of terms, (2) ordered list of terms, (3) weighted terms, and (4) key-value pairs. Features with arbitrary structure are expressed by combining these type constructors. Representing features as typed documents leads to an important advantage —it separates the process of defining and extracting domain specific structure from the process of indexing and searching. The former needs to be done by a domain expert, but the latter can be done generically from the type structure. As shown in Figure 1 the domain expert writes one feature parser for each feature (such as call stack), and the output of each feature parser is a typed document. The typed document is processed by a transform (explained in Section 3), which converts it into a bag of words. We define the transform in such a way that the semantics of the type structure is encoded during the transformation. The query is processed in a similar manner —the same feature parsers are used to extract a typed document from the query, and the same transform is used to convert the typed document into a bag of words. We then use an existing search engine based on TF-IDF (Term Frequency and Inverse Document Frequency, see [16]) to index and search these bags of words.

The second phase of DebugAdvisor (see Figure 2) retrieves recommendations about people, source files, binaries, and source functions that are relevant to the current query, by analyzing relationships between these entities. We are inspired by link-analysis algorithms such as Page Rank [5] and HITS [14] to compute these recommendations. Unlike the world-wide web, where explicit URL pointers between pages provide the link structure, there are no explicit links between these data sources. We use prior work in mining software repositories to establish these relationships [9, 10, 22, 20]. For fixed bugs, it is possible to discover the version control revision that was made to fix the bug. We can then find out which lines of code were changed to fix the bug, which functions and which binaries were changed, and who made the change. The output of such an analysis is a relationship graph, which relates elements in bug descriptions, source files, functions, binaries and people. Starting with a "seed" set of bug descriptions from the first phase, the second phase performs link analysis using probabilistic inference (see Section 4) and retrieves a ranked list of people, files, binaries and functions related to the seed set. As we show in our empirical evaluation (see Section 5), using the first phase to "expand" the query greatly improves the quality of the second phase.

Evaluation. We have collected empirical data for 4 weeks of usage from deploying the latest version of DebugAdvisor to a community of over 100 users in Microsoft's Windows Serviceability group. We present usage statistics, and anecdotal feedback from users. Feedback from our users has been very positive —the tool returned useful results for **78%** of respondents, and queries about open and active bugs returned useful results for **75%** of the cases we tried. In **15%** of the cases we tried with open and active bugs, the tool returned a duplicate bug which led to immediate resolution of the issue. Anecdotal feedback suggests that the tool provides value, and that the fat query interface is very useful.

In addition, we rigorously measure the effectiveness of algorithms implemented using DebugAdvisor in comparison

[1]Every query does not *have* to be kilobytes long. DebugAdvisor can also answer short queries, and some users issue short queries.

[2]Here, we are using the term *type constructor* in the sense used by functional programming languages like ML.

with base-line analyses such as full-text search. We found that DebugAdvisor improves precision and recall (see Section 5 for definition of precision and recall) significantly when compared with base-line analyses. DebugAdvisor's first phase (using features) improves recall by **33%** for challenging queries from our test suite (see Section 5 for definition of challenging queries), and by **14%** for all queries from our test suite. When we consider actual queries posed by users, full-text search is able to retrieve only **65%** of the desired results, which are returned by features and certified as useful by users.

Summary. In summary, DebugAdvisor has the following distinguishing characteristics:

- We allow the user to specify their debugging context as a *fat query*, which collates all the contextual information they have. We are not aware of any information retrieval system that supports such a querying interface to the user.

- We use *typed documents* to represent structure in fat queries and documents (bug descriptions and associated data). We show how typed documents can be used to encode domain specific similarity information. We also show how typed documents can be generically transformed into bags of words so that they can be indexed and searched using existing search engines. We show how probabilistic inference can be used to generate a ranked list of people, source files, functions and binaries related to the query.

- We present an evaluation of DebugAdvisor from our experience deploying it to over 100 users in Microsoft's Windows servicing group. The feedback from users (empirical as well as anecdotal) has been positive. We present precision and recall numbers to demonstrate that individual algorithms in DebugAdvisor are effective.

The remainder of the paper is organized as follows. Section 2 provides some examples to motivate various techniques used in DebugAdvisor. Sections 3 and 4 describe the two phases of DebugAdvisor. Section 5 presents data and experience from deploying DebugAdvisor, and Section 6 covers related work.

2. EXAMPLES

We give some examples to motivate algorithms in the two phases of DebugAdvisor.

To motivate the need for features in the first phase, consider the bug description shown in Figure 3. This bug description contains two parts. The first part is natural language text. It mentions that the problem is a "deadlock", and names an object `ObpInitKillMutant`, which is a semaphore on which threads are blocked. The second part is an extract from a call stack of one thread. The first part is unstructured and the second part has some structure. Suppose we want to search for similar bug descriptions in a database. For this purpose, the first and second parts need to be treated differently. The first part can be handled using full text search, since it has no structure. For the second part, we want to match other bug descriptions whose call stacks are "similar" to the call stack in Figure 3. The definition of a "similarity" between stacks is very particular and

```
The customer experiences some deadlocks on a server. The problem is
random and may occur from several times a week to once a month. The
system looks hung because the global resource 'ObpInitKillMutant' is
help by a thread which tries to close a file for ever. So all the
processes having a thread waiting on 'ObpInitKillMutant' stop working
fine. Drivers such as TCP/IP continue to respond normally but it's
impossible to connect to any share.

Problem Impact:
The impact is high since the servers have to be rebooted when the
problem occurs. As no one can connect to the server anymore(net use),
the production is down. The problem was first escalated as a
severity A.


..
0: kd> !thread 82807020
ChildEBP RetAddr Args to Child
80210000 80c7a028 80c7a068 ntkrnlmp!KiSwapThread+0x1b1
80c7a074 00000000 00000000 ntkrnlmp!KeWaitForSingleObject+0x1b8
80c7a028 00000000 00000000 ntkrnlmp!IopAcquireFileObjectLock+0x58
82a6d7a0 80c7a028 00120089 ntkrnlmp!IopCloseFile+0x79
82a6d7a0 80c7a010 80f6da40 ntkrnlmp!ObpDecrementHandleCount+0x112
00000324 7ffdef01 00000000 ntkrnlmp!NtClose+0x170
00000324 7ffdef01 00000000 ntkrnlmp!KiSystemService+0xc9
00000324 80159796 000000c9 ntkrnlmp!ZwClose+0xb
000000c9 e185f648 00000000 ntkrnlmp!ObDestroyHandleProcedure+0xd
809e3008 801388e4 82a6d926 ntkrnlmp!ExDestroyHandleTable+0x48
00000001 82a6d7a0 7ffde000 ntkrnlmp!ObKillProcess+0x44
00000001 82a6d7a0 82a6d7f0 ntkrnlmp!PspExitProcess+0x54
00000000 f0941f04 0012fa70 ntkrnlmp!PspExitThread+0x447
ffffffff 00000000 00002a60 ntkrnlmp!NtTerminateProcess+0x13c
ffffffff 00000000 00002a60 ntkrnlmp!KiSystemService+0xc9
00000000 00000000 00000000 NTDLL!NtTerminateProcess+0xb
```

Figure 3: A bug description

```
FAILURE_BUCKET_ID:  0x8E_CLASSPNP!TransferPktComplete+1f5
SYMBOL_NAME:  CLASSPNP!TransferPktComplete+1f5
MODULE_NAME: CLASSPNP
IMAGE_NAME:  CLASSPNP.SYS
FAILURE_BUCKET_ID:  0x8E_CLASSPNP!TransferPktComplete+1f5
BUCKET_ID:  0x8E_CLASSPNP!TransferPktComplete+1f5
```

Figure 4: Extracts from another bug description

domain specific. The actual addresses in the call stack are not important for similarity —thus the first three columns in the stack trace can be ignored, and two stacks are similar if they have the same functions in the same order. Thus, for the bug description in in Figure 3, the function names such as `ObpDecrecmentHandleCount`, `ZwClose`, `ObpDestroyHandleProcedure`, `ExDestroyHandleTable`, `ObKillProcess` and the order in which they appear are important for determining similarity with respect to other call stacks.

Sometimes bug descriptions contain several attributes and values, as shown in Figure 4. In such situations, the association between the attribute and value is important while determining similarity between two bug reports. For instance, a bug report with an attribute `IMAGE_NAME` having the value `CLASSPNP.SYS` is more similar to the bug report in Figure 4 than another report that has both the terms `IMAGE_NAME` and `CLASSPNP.SYS` at different parts of the document, but not related together as an attribute-value pair.

Call stacks and image names are just two examples of features in bug descriptions. There are several other features such as semaphores, mutexes, memory dumps, exceptions etc. Each of them have their own domain specific notions of similarity. In Section 3, we propose a notion of similarity between typed documents, which can be implemented using any existing search engine as a black-box. This allows us to

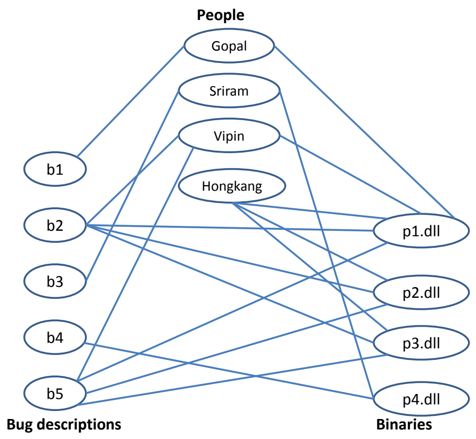

Figure 5: Example relationship graph

```
Type  BaseType   =     String
                 |     Int

Type  NamedDoc   =     Value(BaseType)
                 |     KeyValuePair(BaseType ∗ Doc)

Type  Doc        =     Null
                 |     Base(BaseType)
                 |     Bag(NamedDoc set)
                 |     Ordered(NamedDoc list)

Type  \widehat{Doc}  =     Bag( Base set)
```

Figure 6: Typed documents

leverage enormous amount of progress that has been made in indexing and search engines over the past decade.

To motivate the need for link analysis in the second phase, consider the relationship graph shown in Figure 5. In this graph, there are 5 bug descriptions `b1`, `b2`, `b3`, `b4`, `b5`, and 4 people `Gopal`, `Sriram`, `Vipin` and `Hongkang` and 4 binaries `p1.dll`, `p2.dll`, `p3.dll` and `p4.dll`.

Suppose we start with a query for which the first phase returns the set of bugs { `b2`, `b5` }. The objective of the second phase is to use these bugs as "seeds" and find people and binaries related to these bugs. We note that `Vipin` is related to `b2` and `b5`, and that the binaries {`p1.dll`, `p2.dll`, `p3.dll`} are related to { `b2`, `b5` }. Further, since `Hongkang` is related to the binaries {`p1.dll`, `p2.dll`, `p3.dll`}, we want to infer that `Hongkang` is related to the bugs { `b2`, `b5` } as well. Thus, we want to infer { `Hongkang`, `Vipin` } as the set of people related to the bugs { `b2`, `b5` }.

Probabilistic inference using factor graphs provides a uniform way to do this analysis over large relationship graphs and obtain a ranked list of people, bugs, binaries, and functions that are most closely related to the seeds. Further details can be found in Section 4.

3. FIRST PHASE

The first phase of DebugAdvisor needs to match documents from a corpus that are similar to a fat query. This is related to the open research problem of similarity detection between documents.[3] A major difficulty in the debugging domain is the variegated mixture of structured and unstructured data in the repositories. We need domain expertise to specify domain specific notions of structure and similarity. At the same time, we desire to reuse existing robust IR infrastructure for full-text search. Our approach is to uniformly represent both queries and information sources as a collection of *features*, which are formalized as *typed documents*. Typed documents help us meet the conflicting goals of representing domain specific structure and using generic index and search infrastructure. Domain experts can represent features as typed documents, and we present generic techniques to transform typed documents to bags of words for reusing index and search infrastructure.

Typed Documents and Similarity. The specification of a typed document is given in Figure 6 in ML-style syntax. A document `Doc` is one of (1) a null document, (2) base type (integer or string), (3) a bag of named documents (which is an unordered set), or (4) an ordered list of named documents. A named document `NamedDoc` is either a value of base type or a key-value pair consisting of a key (of base type) and a value (of document type).

In contrast with typed documents, full-text search can be thought of as implementing a similarity metric between a less expressive type structure, namely a bag of terms. More precisely, a full-text search engine operates over the (impoverished) type structure $\widehat{\texttt{Doc}}$ from Figure 6. A full-text search engine implements retrieval based on a score function $\widehat{Score}$: $(\widehat{\texttt{Doc}} \times \widehat{\texttt{Doc}}) \rightarrow \texttt{Int}$, where $\widehat{Score}(\texttt{Bag}(q), \texttt{Bag}(d))$ is given by the nested sum $\sum_{t \in q} \sum_{s \in d}$ (**if** $(t = s)$ **then** 1 **else** 0). Our goal here is to define and implement a similarity metric between typed documents represented by the type `Doc`.

Note that the grammar for typed documents forces the occurrence of a key at each level of the recursive type structure. In the rest of this paper, we assume that each key occurs only once in a type definition. Thus, each sub-document of a document can be uniquely scoped using a sequence of keys used to reach the sub-document from the root. We define a function *Score*: $\texttt{Doc} \times \texttt{Doc} \rightarrow \texttt{Int}$ that maps a pair of documents to an integer score. Intuitively, $Score(q, d)$ is a measure of the similarity between q and d. The more similar q and d are, the higher the value of $Score(q, d)$. Figure 7 defines *Score* inductively over the type structure of its arguments.

If the documents q and d are base type expressions, then *Score* is defined to be 1 if they are identical and 0 if they are not identical. Suppose the first two arguments to *Score* are bags of documents. Then $Score(\texttt{Bag}(q), \texttt{Bag}(d))$ is defined to be sum of the scores of matching the elements of q and d. The auxiliary function $Match : \texttt{NamedDoc} \times \texttt{NamedDoc} \rightarrow \texttt{Int}$ returns the score of base types if the arguments are base types. If the arguments are key-value pairs, *Match* returns the score between the values of the two documents if the keys match, and 0 otherwise.

Suppose the first two arguments to *Score* are ordered lists of documents. An n-gram of a sequence q is a contiguous sub-sequence of q of length n. $NGram(q, n)$ is set of all n-grams of a sequence q. Then $Score(\texttt{Ordered}(q), \texttt{Ordered}(d))$ is the sum of the scores of matching n-grams for q and d. In practice, we consider n-grams for values of n equal to 1 (unigrams) and 2 (bigrams).

[3]Please see [21] for a recent approach to the problem, applicable to the domain of on-line news articles.

$$
\begin{array}{lcl}
\mathit{Score}(\texttt{Base}(q), \texttt{Base}(d)) & = & \textbf{if } (q = d) \textbf{ then } 1 \textbf{ else } 0 \\
\mathit{Score}(\texttt{Bag}(q), \texttt{Bag}(d)) & = & \sum_{s \in q} \ \sum_{t \in d} \mathit{Match}(s, t) \\
\mathit{Score}(\texttt{Ordered}(q), \texttt{Ordered}(d)) & = & \sum_{n \in \{1,2\}} \sum_{t \in \mathit{NGram}(q,n)} \sum_{s \in \mathit{NGram}(d,n)} \mathit{NGramMatch}(s, t, n) \\
\mathit{Match}(\texttt{Value}(t), \texttt{Value}(t')) & = & \mathit{Score}(\texttt{Base}(t), \texttt{Base}(t')) \\
\mathit{Match}(\texttt{KeyValuePair}(k_1, v_1), \texttt{KeyValuePair}(k_2, v_2)) & = & \textbf{if } (k_1 = k_2) \textbf{ then } \mathit{Score}(v_1, v_2) \textbf{ else } 0 \\
\mathit{NGramMatch}(s, t, n) & = & \prod_{i=1..n} \mathit{Match}(s_i, t_i)
\end{array}
$$

Figure 7: Scoring Typed Queries. Cases not given (eg. $\mathit{Match}(\texttt{Value}(t), \texttt{KeyValuePair}(k, v), c)$) are defined as 0.

$$
\begin{array}{lcl}
\mathcal{T}(\texttt{Null}) & = & \{\} \\
\mathcal{T}(\texttt{Base}(x)) & = & \{x\} \\
\mathcal{T}(\texttt{Bag}(\{v_1, v_2, \ldots, v_n\})) & = & \biguplus_{1 \le i \le n} \mathcal{N}(v_i) \\
\mathcal{T}(\texttt{Ordered}(\{v_1, v_2, \ldots, v_n\})) & = & \biguplus_{1 \le i \le n} \mathcal{N}(v_i) \ \ \uplus \\
 & & \biguplus_{1 \le i \le (n-1)} \mathit{Join}(\mathcal{N}(v_i), \mathcal{N}(v_{i+1})) \\
 & & \\
\mathcal{N}(\texttt{Value}(x)) & = & \{x\} \\
\mathcal{N}(\texttt{KeyValuePair}(k, x)) & = & \mathit{Prefix}(k, \mathcal{T}(x)) \\
 & & \\
\mathit{Prefix}(s, X) & = & \{s\#x \mid x \in X\} \\
\mathit{Join}(X, Y) & = & \{x\#y \mid x \in X, y \in Y\} \\
 & & \\
x\#y & = & x \circ \text{"\_\$\%\$\_"} \circ y, \\
 & & \text{ where } \circ \text{ denotes string concatenation}
\end{array}
$$

Figure 8: Transformation from typed documents to bags of words

We assume here implicitly that the type structure of the query and document match. For parts of the parse tree that do not match *Score* is implicitly defined to be 0. For instance, $\mathit{Match}(\texttt{Value}(t), \texttt{KeyValuePair}(k, v))$ is 0 since it does not match any of the templates given in Figure 7.

In addition to the constructors shown in Figure 6, our queries also allow annotating arbitrary subtrees with integer weights. That is, we allow queries to contain another constructor $\texttt{WeightedQuery(Int}*\texttt{Doc)}$, and we extend the *Score* function from Figure 7 as: $\mathit{Score}(\texttt{WeightedQuery}(w, q), d) = w \times \mathit{Score}(q, d)$. This allows us to write our features in such a way that more importance is given to matching certain important parts of the query (such as call stacks).

Transformation. A salient feature of our implementation is that it leverages existing technology for full-text search. Thus, we are able to use any full-text search engine as a subroutine, to implement indexing and searching typed documents.

We now show how to implement the score function $\mathit{Score} : (\texttt{Doc} \times \texttt{Doc}) \rightarrow \texttt{Int}$ described in Figure 7 using the score function $\widehat{\mathit{Score}} : (\widehat{\texttt{Doc}} \times \widehat{\texttt{Doc}}) \rightarrow \texttt{Int}$. The key idea is to use a transformation $\mathcal{T} : \texttt{Doc} \rightarrow \widehat{\texttt{Doc}}$ such that for any two typed documents d_1 and d_2 we have that $\widehat{\mathit{Score}}(\mathcal{T}(d_1), \mathcal{T}(d_2))$ is equal to $\mathit{Score}(d_1, d_2)$.

Intuitively, $\mathcal{T}(d)$ walks the parse tree of the typed document d and transforms every internal node to a bag of words (where each word belongs to the base type). The transformation $\mathcal{T}$ is defined in Figure 8. Even though we use set notation to represent bags, we note that bags differ from sets in that the same element can occur more than once. Thus $\{a, b, a\}$ is a bag and is identical to $\{a, a, b\}$ —the ordering of elements is irrelevant. We use $\uplus$ to denote union operator on bags. Given two bags b_1 and b_2 we have that $b_1 \uplus b_2$ contains all the elements in b_1 together with all the elements in b_2. If an element e appears n_1 times in b_1 and n_2 times in b_2, then e appears $(n_1 + n_2)$ times in $b_1 \uplus b_2$.

For a document equal to Null, we have that $\mathcal{T}(\texttt{Null})$ is the empty bag denoted by $\{\}$. For a document of the form $\texttt{Base}(x)$, we have that $\mathcal{T}(\texttt{Base}(x))$ is given by the bag x which contains one element, namely x. For a document of the form $(\texttt{Bag}(\{v_1, v_2, \ldots, v_n\}))$ we have that $\mathcal{T}(\texttt{Bag}(\{v_1, v_2, \ldots, v_n\}))$ is given by the union of the bags obtained by transforming each of $v_1, v_2, \ldots v_n$ using $\mathcal{N}$. The function $\mathcal{N} : \texttt{NamedDoc} \rightarrow \widehat{\texttt{Doc}}$ maps named documents to bags of words. If a named document is of the form $\texttt{KeyValuePair}(k, x)$, we have that $\mathcal{N}(\texttt{KeyValuePair}(k, x))$ is obtained by concatenating the prefix k and a special character sequence, say "_$%$_", to every element in $\mathcal{T}(x)$. We assume that the special character sequence "_$%$_" does not appear anywhere in the original typed document. To see the rationale for this prefix operation, recall the definition of $\mathit{Match}(\texttt{KeyValuePair}(k_1, v_1), \texttt{KeyValuePair}(k_2, v_2), c)$ from Figure 7. Recall that *Match* evaluates to 0 if the keys k_1 and k_2 do not match, and evaluates to $\mathit{Score}(v_1, v_2, c\#k1)$ otherwise. Prefixing k to every element in $\mathcal{T}(x)$ has an identical effect during the computation of $\widehat{\mathit{Score}}$ on the transformed document.

For a document of the form $(\texttt{Ordered}(\{v_1, v_2, \ldots, v_n\}))$ we have that $\mathcal{T}(\texttt{Ordered}(\{v_1, v_2, \ldots, v_n\}))$ is given by the union of the bags obtained by transforming each of the unigrams $v_1, v_2, \ldots v_n$ using $\mathcal{N}$, as well as the union of bigrams obtained by joining every successive pair of transformed v_i's using the operator *Join*. The operator $\mathit{Join}(X, Y)$ obtains a cross product of terms in the two bags X and Y. To see the rationale for the join operation recall the definition $\mathit{Score}(\texttt{Ordered}(q), \texttt{Ordered}(d), c)$ from Figure 7. Note that the definition of $\mathit{Score}(\texttt{Ordered}(q), \texttt{Ordered}(d), c)$ sums up the scores of all matching unigrams and bigrams of q and d.

The following theorem relating *Score* and $\widehat{\mathit{Score}}$, can be established by induction over the type structure of typed documents.

THEOREM 1. *For any two typed documents q and d, we have that the score $\mathit{Score}(q, d)$ is equal to $\widehat{\mathit{Score}}(\mathcal{T}(q), \mathcal{T}(d))$.*

Example. Consider a section of the call stack from Figure 3. The feature parser for call stack extracts the ordered list `Ordered([ntkrnlml!KiSwapThread, ntkrnlml!KeWaitForSingleObject, ..., ntkrnlml!TerminateProcess])` as a typed document representing the feature. Note that the feature parser trims memory address values and suffixes such as `+x0x1b1` from

q = `Ordered([KiSwapThread, KeWaitForSingleObject, IopAcquireFileObjectLock, IopCloseFile])`

d_1 = `Ordered([A, KiSwapThread, KeWaitForSingleObject,B])`

d_2 = `Ordered([KiSwapThread, C, IopAcquireFileObjectLock,D])`

$\mathcal{T}(q)$ = `{KiSwapThread, KeWaitForSingleObject, IopAcquireFileObjectLock, IopCloseFile KiSwapThread_$%$_KeWaitForSingleObject, KeWaitForSingleObject_$%$_IopAcquireFileObjectLock, IopAcquireFileObjectLock_$%$_IopCloseFile }`

$\mathcal{T}(d_1)$ = `{A, KiSwapThread, KeWaitForSingleObject, B, A_$%$_KeWaitForSingleObject KiSwapThread_$%$_KeWaitForSingleObject, B_$%$_KeWaitForSingleObject }`

$\mathcal{T}(d_2)$ = `{KiSwapThread, C, IopAcquireFileObjectLock, D, KiSwapThread_$%$_C C_$%$_IopAcquireFileObjectLock, IopAcquireFileObjectLock_$%$_D }`

Figure 9: Example to illustrate the transformation

each stack frame and retains only the function name. We illustrate the *Score* function and the transformation $\mathcal{T}$ using a smaller ordered list below.

Suppose, we have typed documents representing a query q and two bug descriptions d_1 and d_2 as shown in Figure 9. $Score(q, d_1, \epsilon) = 3$, since q and d_1 have two matching unigrams, namely `[KiSwapThread]`, `[KeWaitForSingleObject]`, and one matching bigram, namely `[KiSwapThread, KeWaitForSingleObject]`. We also have that $Score(q, d_2, \epsilon) = 2$, since q and d_2 have two matching unigrams namely `[KiSwapThread]`, `[IopAcquireFileObjectLock]`. This matches our intent that q is more similar to d_1 than to d_2 since there are two contiguous function names `KiSwapThread` and `KeWaitForSingleObject` that match between q and d_1. On the other hand, if we had treated q, d_1 and d_2 as bags of words then we would have that $Score(q, d_1, \epsilon) = Score(q, d_2, \epsilon) = 2$, and the ordering and sequencing of words will not be taken into account.

Figure 9 also shows the values of T(q), $\mathcal{T}(d_1)$ and $\mathcal{T}(d_2)$. Assuming a constant *IDF* of 1, we have that $\widehat{Score}(\mathcal{T}(q), \mathcal{T}(d_1)) = 3$ since $\mathcal{T}(q)$ and $\mathcal{T}(d_1)$ have three terms in common, namely `KiSwapThread`, `KeWaitForSingleObject`, and `KiSwapThread_$%$_KeWaitForSingleObject`. We also have that $\widehat{Score}(\mathcal{T}(q), \mathcal{T}(d_2)) = 2$ since $\mathcal{T}(q)$ and $\mathcal{T}(d_2)$ have two terms in common, namely `KiSwapThread`, and `IopAcquireFileObjectLock`. Recalling the values of $Score(q, d_1, \epsilon)$ and $Score(q, d_2, \epsilon)$ we note that $Score(q, d_1, \epsilon) = \widehat{Score}(\mathcal{T}(q), \mathcal{T}(d_1)) = 3$, and $Score(q, d_2, \epsilon) = \widehat{Score}(\mathcal{T}(q), \mathcal{T}(d_2)) = 2$.

Term Frequency and Inverse Document Frequency. The above description of *Score* and $\widehat{Score}$ are simplistic in that matches among all terms are weighted equally. In practice, search engines weight terms inversely proportional to the frequency with which they occur in the corpus. This has the effect of weighting matches on infrequently occurring terms higher than matches on frequently occurring terms. More precisely, $\widehat{Score}(\mathtt{Bag}(q), \mathtt{Bag}(d))$ is given by the nested sum $\sum_{t \in q} \sum_{s \in d}$ (**if** $(t = s)$ **then** $IDF(t)$ **else** 0). Here, we note that $IDF(t)$ is equal to $\mathbf{log} \frac{N}{DF(t)}$, where N is the number of documents in the corpus and $DF(t)$ is the number of times term t appears in the corpus. (see Section 6 of [16] for an introduction to *IDF*).

Implementation. Our implementation of $\mathcal{T}$ is done in C# using a visitor that walks over the parse tree of features (see Figure 1 for where the transformer fits into the architecture of the first phase). The documents for DebugAdvisor come from a variety of sources. There are two kinds of documents that we index for the first phase of DebugAdvisor: (1) bug records from the Windows bug databases, (2) debug logs from actual debugging sessions that happen from stress breaks and from debug sessions that are done by developers and testers. For collecting debug logs, we have implemented a wrapper around the kernel debugger, which logs all commands issued by the user and responses from the debugger into a text file. The wrapper allows the developer to record the bug identifier (if it exists) for the current debugging session.

We use 4 features: (1) bag of interesting words, such as "impersonating", "exclusive", "deadlock", "hang", "overflow", "multicore", etc (represented as a bag), (2) debug commands issued with their output results from the debugger (represented as bag of key-value pairs), (3) values of special attributes such as `CURRENT_IRQL, PROCESS_NAME, MODULE_NAME, BUCKET_ID`, etc. (represented as bag of key-value pairs), and (4) stack frame (represented as an ordered list).

4. SECOND PHASE

The results from the first phase can be thought as an "expansion" of the original query, and the second phase aims to leverage information from the expanded query to infer related entities. The benefit of this query expansion can be quite significant, because often the raw query (say just a stack trace from a machine being debugged) matches a set of related bugs and these bugs have a number of explicit links to related entities. However, just listing the links explicitly included in the set of bugs found in the first phase is not enough. Important relationships can be found by performing transitive closure on the relationship graph starting with the set of bugs found in the first phase. The goal of the second phase is to perform such a transitive closure and retrieve a ranked list or related entities.

A relationship graph is a weighted multi-partite graph $G = \langle V, E \rangle$, where the set V of vertices is partitioned into n mutually disjoint partitions $\{V_1, V_2, \dots, V_n\}$, such that $E \subseteq V \times V \times \mathbf{Nat}$ contains only edges between two vertices in different partitions. That is, if $\langle s, t, w \rangle \in E$, and $s \in V_i$ and $t \in V_j$, then we have that $i \neq j$. The third component of the edge w is a weight, a natural number, which is a measure of the relationship from s to t —-the larger the weight, the tighter the relationship.

The vertices of the relationship graph used by DebugAdvisor have one partition V_{bugs} for bug descriptions (this includes bug reports as well as logs from interactive debugging sessions as mentioned in Section 2), one partition for V_{files}, one partition $V_{functions}$ for functions, and one partition V_{people} for people. We use an existing tool called BCT [20] to build this relationship graph from various data repositories. For every resolved bug in the bug database, BCT looks for check-ins in source code version control that are associated with fixing the bug, and computes functions and source files that were changed to fix the bug, and the people who made those changes. BCT creates relationship links between these entities using such an analysis.

The output of the first phase, R_1, is a set of bug descriptions that are highly related to the query. That is $R_1 \subseteq$

V_{bugs}. The goal of the second phase is to use R_1 as a starting point to perform link analysis on the relationship graph and compute a 4-tuple $R_2 = \langle v_{bugs}, v_{files}, v_{functions}, v_{people}\rangle$, where $v_{bugs} \subseteq V_{bugs}$, $v_{files} \subseteq V_{files}$, $v_{functions} \subseteq V_{functions}$, $v_{people} \subseteq V_{people}$.

The second phase attempts to identify vertices R_2 in the relationship graph that are most correlated with the vertices R_1 that are produced by the first phase. Formally, a relationship graph can be viewed as a *Markov chain* (see [16]) with $|V|$ vertices, and a $|V| \times |V|$ transition probability matrix each of whose entries lies in the interval $[0, 1]$. The transition probabilities are obtained by normalizing edge weights of the relationship graph such that the sum of the normalized edge weights going out of each vertex is 1. We desire to compute the steady state probability distribution over the vertices of the Markov chain for a random walk starting at vertices R_1. We want to return the vertices with large steady state probabilities in each of the vertex partitions as the result R_2 of the second phase.

In our implementation, we use factor graphs [15] to compute these steady state probabilities. We associate one Bernoulli random variable with each vertex in V. The random variable for each element in R_1 is set to an initial distribution where the variable takes a value 1 with probability 0.9 and 0 with probability 0.1. All the other random variables (associated with vertices in $V \setminus R_1$) are set to have an initial distribution where the variable takes a value 1 with probability 0.5 and 0 with a probability 0.5. For every edge $e = \langle u, v, w\rangle$ which associates verities u and v with weight w, we add a factor F_e (a probabilistic constraint) which constrains the joint probability distribution of the random variables X_u and X_v associated with u and v. The constraint F_e states that X_u and X_v take the same value with probability p and different values with probability $1 - p$. The value of the parameter p depends on the weight w (normalized with the weights of other edges connected to u and v). We use factor graph inference [15] to compute the posteriori probabilities of each random variable, and choose the random variables with highest posteriori probabilities for result set R_2. Our implementation of DebugAdvisor uses the factor graph package Infer.NET [1].

5. EVALUATION

We have deployed DebugAdvisor to the Windows Serviceability group inside Microsoft. Engineers in this team are spread across two locations —Redmond (USA) and Hyderabad (India). We want to address two questions: (1) How effective is DebugAdvisor in serving the needs of our users? (2) How effective are the various components of DebugAdvisor when compared with existing approaches like full-text search? We answer the first question with usage numbers, user feedback logged in the tool, and anecdotal feedback through emails from users, and interviews with users. We answer the second question by comparing precision-recall numbers for the two phases of DebugAdvisor with existing techniques.

5.1 Usage and feedback

Over the past 6 months, we have done 2 major releases and 3 minor (bug fix) releases of DebugAdvisor. The first major release was done for a small early-adopter community of about 10 engineers. We received useful feedback from this early-adopter community. Users gave us several requirements to make the tool useful to them: (1) additional bug databases that they wanted us to index and search, (2) need to automatically update the indexes as new bugs get filed (ensuring that the search done by DebugAdvisor includes recent bug reports), (3) request to index and search through actual logs of debug sessions. We addressed most of these issues and made an enhanced release to the entire Windows Serviceability group. The scale and engineering effort required for the current release of DebugAdvisor service is non-trivial. The service currently indexes **2.36 million** records (including bug reports, attachments to bug reports, debug logs, etc) collected over several years. The DebugAdvisor service also builds and analyzes relationships between users, files, functions and bug reports, using data from several hundred thousand version control check-ins.

Description	Value
Number of users	129
Number of queries	628
Unsolicited responses: recommendation *useful*	161
Unsolicited responses: recommendation *not useful*	47
Percentage unsolicited responses: *useful*	**78%**
Number of active bugs where we solicited responses	20
Solicited responses: recommendation *useful*	15
Solicited responses: recommendation *not useful*	5
Percentage solicited responses: *useful*	**75%**

Figure 10: User responses

We log all users and all queries to the DebugAdvisor service. Usage statistics are given in Figure 10. We have had **129 users**, and **628 queries** in the first 4 weeks since the current release. For each item we recommend (bug reports, people, files, etc) our UI offers the facility to respond stating if the recommendation was useful or not useful. We received 208 responses (given voluntarily by users), and **78%** of the responses state that recommendations were useful.

We performed another study on **20** randomly picked active bugs from the bug database. We used DebugAdvisor to search for related information, and then contacted the bug owner (i.e, the person who is working on the bug) with the information returned by DebugAdvisor, and asked them for feedback on whether the search results returned were useful. As shown in Figure 10, **75%** of these responses stated that search results were useful. In addition, we note that in **3** cases (out of 20) an exact duplicate of the active bug was found leading to immediate resolution of the bug.

5.2 Anecdotal evidence

Anecdotal evidence from users has been positive.

Stress break. An engineer debugging a failure during stress testing sent us the following email: *"We had one break today from DTP41 testing. So the first thing I did was to go to http://debugadvisor, copied the stack from the break into the search query, and pressed Search. And right there the first bug given back by DebugAdvisor, which is the exact match for this break. I didn't even need to check the other results it returned."*

Customer support. An engineer from customer support sent us the following email: *"..and my query worked successfully. Additionally, the second bug in the list returned the right match, as my debugging found it is the same issue and*

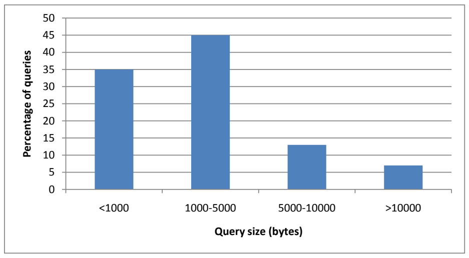

Figure 11: Sizes of queries from users.

now I can request a port to Windows 2008. Thanks for the help!"

Other positive feedback. Several users sent us their positive experience without mentioning specific details about the issues they tried the tool on. One user's feedback said: *"This tool seems smart in searching related bugs based on stack pattern match. I pasted a callstack and clicked search button and I got some useful information for the related bug, [other tools do] not have such feature like search by bulk text."*

Another user said: *"..right on 3x in a row! idea: display some sort of relevance/confidence indicator for each guess. It is very useful to use full description to search for related bug. It is very accurate in finding related bugs."*

Another user said: *"Nailed it on the very first result! Impressive, indeed :) Will there be a possibility of search word highlighting in the future?"*

Negative feedback. However, not every user found what they were looking for. One user said *"I threw a few queries at it and it returned nothing interesting. Do we still need to wait until a critical mass of debug logs is built up or something?"* .

After some investigation, we found that this user's team works on very recently filed bugs, and ones for which we are less likely to have historical data similar to the issues under investigation. We believe that this team will benefit more from indexing and searching logs from recent debugging sessions. We are pushing wider adoption of our debugger wrapper that automates collection of debug logs in this team.

Another user said *"252422 and 251122 are very similar bugs for search query [and they are returned by DebugAdvisor] but are separated by a large number of unrelated results, leading me not to look at the proper related bugs."* In this case, we got the relevant information, but did not rank them appropriately. We are investigating tuning our ranking algorithm using the data we collected from such feedback.

5.3 Quantitative Data

Query size and performance. Figure 11 shows distribution of query sizes from the 628 queries collected over 4 weeks of deployment. We note that more than 65% of the queries were over 1K bytes, and about 25% of the queries were over 5K bytes. Response time for queries depends on query size. For short queries (a few words) response times range over 15 to 20 seconds. Average query times for fat queries —ranging in size from 4K to 30K— is about 2 minutes. The longest query time we have ever encountered is

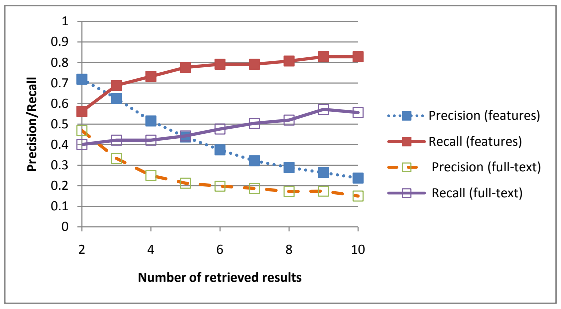

Figure 12: Precision and recall for first phase of DEBUGADVISOR**(challenging queries).**

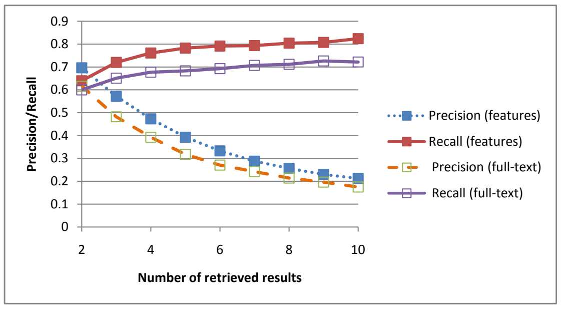

Figure 13: Precision and recall for first phase of DEBUGADVISOR**(all queries).**

5 minutes. A common sentiment among DEBUGADVISOR users is that they are willing to wait for a few minutes as long as the tool produces a useful result, because the previous alternative they had (querying various repositories with ad-hoc small queries and combining the results manually) was very painful and time consuming.

Precision and recall. Suppose we retrieve k documents in response to a query. Suppose r of these documents fall within the expected results, and there are a total of t expected results (note that $r \leq k$ and $r \leq t$). Then, *precision* is defined as the ratio $\frac{r}{k}$, and *recall* is defined as the ratio $\frac{r}{t}$ [16]. A high precision value indicates that most results retrieved were useful, and a high recall value indicates that most useful results were retrieved.

Measuring precision and recall for a tool such as DEBUGADVISOR is tricky because it is very hard to specify a complete set of expected results for any query. We took the following approach. We picked **50 queries** from existing bug reports at random, studied them extensively over a period of several months, and manually constructed a set of expected search results for each of these queries.[4]

During manual analysis of bug pairs classified as "related" (and therefore expected to be found by the system), we noticed that several of them, especially those classified as "duplicate", have significant portions of matching text in the query —a quick glance is all that is needed to realize that it is

[4]When using an existing bug report as a query, we concatenate all the text in the bug report and use it as a fat query.

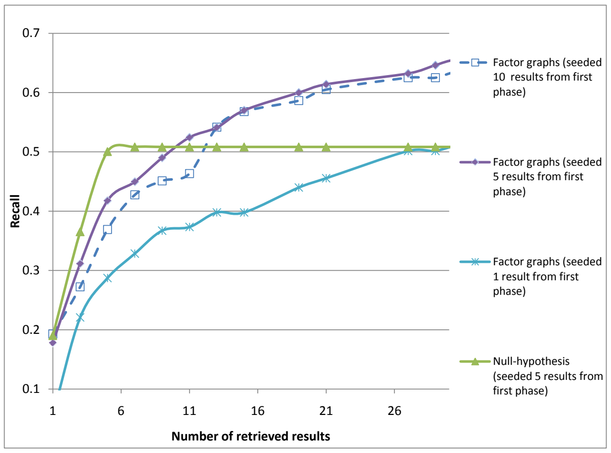

Figure 14: Recall for related people from second phase of DebugAdvisor.

the same issue. However, significantly, we noted that about **30%** of the related bug pairs we examined required nontrivial analysis of content to determine their relatedness, often because a substantial portion of the text is different (such as when a symptom of an operating system bug manifests itself through different applications.) We classify these kinds of queries as "challenging". Of the 50 queries we analyzed (which were chosen at random), we found that 16 queries were challenging. Figure 12 compares precision and recall numbers for these challenging queries between DebugAdvisor's first phase and full-text search. We note that DebugAdvisor improves recall by **33%** on these queries. With all queries (including challenging and duplicate queries), DebugAdvisor's first phase improves recall by **14%** as shown in Figure 13.

We also performed precision-recall comparisons for queries from actual users, using the responses that users found to be useful as the expected set of golden results. Compared to the first phase of DebugAdvisor, we found that full-text search is able to recall only **65%** of the results found to be useful by our users, clearly illustrating the value of our feature-based approach.

Figure 14 presents recall numbers for related people retrieved by the second phase of DebugAdvisor. We find that the recall varies depending on how many results from the first phase are used to seed the second phase. We find that best recall numbers for the second phase are obtained by using 5 top results from the first phase to seed the second phase. If we use less than 5 results, we seem to be missing some important seeds, and if we use a larger number of results (say 10), we seem to be introducing noise into the starting seeds. Figure 14 compares the recall for factor graphs with a null-hypothesis approach (the latter just returns the people associated with the seeds from first phase without doing any search). We find that recall improves with factor graphs for larger values of retrieved results. We are working on improving our ranking algorithm to manifest this improvement for smaller number of retrieved results.

Threats to validity. All empirical studies were done with the Windows bug data. While Windows is an extremely large and diverse software base with over 100 million lines of code, ranging from GUI to low-level kernel components, further work is needed to evaluate if results generalize to other projects. It is possible that some users were unsatisfied with the tool, but did not send a negative response. This could affect the reported percentage of positive responses. Our precision and recall numbers were computed with a modest set of randomly selected queries. Though these were selected at random, further work is needed to evaluate if empirical results generalize to a larger data set. The main difficulty in scaling evaluation of precision and recall to larger data sets is the manual effort needed to determine the set of all expected results.

6. RELATED WORK

Mining software repositories and using the computed information to help improve the productivity of programming tasks has been an active area of research for the past several years. Kim, Pan and Whitehead have used historical data from fixed bugs in version control to detect project-specific bugs that still exist in the code [13]. The eROSE tool mines software version histories and uses mined information to suggest other places that need to be changed when the programmer attempts to change a method or a class [24]. The Vulture tool correlates information mined from source code modification history, vulnerability reports, and software component structure to provide risk assessments of areas of code that are more likely to be sources of future vulnerabilities [17]. Our work mines software repositories to help programmers and testers during debugging.

The presence of duplicate bugs in software repositories has received much attention. Runesan et a.l [18] propose using natural language processing techniques, and Jalbert and Weimer [12] propose using textual similarity and clustering techniques to detect duplicate bug reports. Wang et al. propose combining similarity information from both natural language text and execution traces to detect duplicate bugs [23]. Bettenburg et al. propose detecting and merging duplicate bug reports so as to give developers maximum information from various sources to diagnose and fix bugs [3]. Bettenburg et al. present a study on what constitutes a good bug report by conducting surveys with open-source developers and bug reporters [2]. Interestingly, their paper states that the presence of a stack trace is one of the most important desideratum for a developer to resolve the bug quickly. The mixture of structured and unstructured information in bug reports has been noticed before. Bettenburg et al. have proposed a tool infoZilla to extract structural information from bug reports [4]. The first phase of DebugAdvisor is the next step in this line of research —we allow domain experts to identify structure in bug reports, and propose a transformation that allows such domain specific notions of similarity to be implemented using existing search engines that are structure agnostic. The distinguishing features of our work are two fold: (1) We allow our users to query our system using a "fat query" which is a textual concatenation of all the information they have about the issue in hand, including text from emails, text from logs of debug sessions, viewing core dumps in the debugger, etc. (2) Our tool architecture and the notion of typed documents enables a clean and systematic way to incorporate domain-specific knowledge into an information retrieval system for handling fat queries. This enables leveraging the domain expert's knowledge about feature similarity while still reusing existing robust and scalable infrastructures for full-text search.

Just as there are duplicates in bug reports, there are clones in code. Clone detection is an active area of current research [11]. Structural similarity between pieces of code can be used to improve productivity of programming tasks. Cottrell et al. automatically extract structural relationships between code for the purpose of generalization tasks [7], and in particular code reuse [8]. While these works employ heuristics to discover structural similarity between pieces of source code, the architecture of DebugAdvisor uses typed documents to enable domain experts to specify structure in bug reports in a domain-specific manner.

We are not the first to build relationship graphs between various artifacts. The Hipikat recommendation system [9, 10] builds and uses a relationship graph that relates various software artifacts. Similar relationship graphs have since been built by others, for example, see Gina Venolia's Bridge [22] and Alex Tarvo's BCT [20]. Random walks on such relationship graphs have also been suggested before. The Fran tool [19] performs random walks on call graphs to recommend related API functions. The distinguishing feature of our work is to layer the search in two phases: The first phase uses similarity between bug reports to find other bug reports related to the current report. The second phase uses random walks on the relationship graph starting at the bug reports identified by the first phase to provide recommendations of related people, source files and functions.

Acknowledgment. We thank Pratima Kolan, Udayan Khurana, Krishna Mehra, Ritesh Nayak, Kanika Nema, Aditya Nori, Aditya Sankar, Vibhuti Sengar, Raghavendra Udupa and Gina Venolia for their help in implementing and evaluating various parts of DebugAdvisor. We thank Vijay D'Silva, Prem Devanbu, Pieter Hooimeijer, Venkatesh-Prasad Ranganath and Tom Zimmerman for comments on early drafts of this paper.

7. REFERENCES

[1] Infer.NET. http://research.microsoft.com/ /en-us/um/cambridge/projects/infernet/.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *FSE 08: Foundations of Software Engineering*, pages 308–318, 2008.

[3] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful ... really? In *ICSM 2008: International Conference on Software Maintenance*, pages 337–345, 2008.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *MSR '08: Mining Software Repositories*, pages 27–30, 2008.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[6] S. Budge, N. Nagappan, S. K. Rajamani, and G. Ramalingam. Global software servicing: Observational experiences at Microsoft. In *IGCSE 2008: Global Software Engineering*, 2008.

[7] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *FSE 07: Foundations of Software Engineering*, pages 165–174, 2007.

[8] R. Cottrell, R. J. Walker, and J. Denzinger. Jigsaw: a tool for the small-scale reuse of source code. In *ICSE Companion 2008*, pages 933–934, 2008.

[9] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE 03: International Conference on Software Engineering*, pages 406–418, 2003.

[10] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Engineering*, 31(6):446–465, 2005.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE 08: International Conference on Software Engineering*, pages 321–330, 2008.

[12] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *DSN 08: Dependable Systems and Networks*, pages 52–61, 2008.

[13] S. Kim, K. Pan, and E. J. Whitehead. Memories of bug fixes. In *FSE 06: Foundations of Software Engineering*, pages 35–45, 2006.

[14] J. M. Kleinberg. Hubs, authorities, and communities. *ACM Computing Surveys*, 31, 4es, 5, 1999.

[15] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Information Theory*, 47(2):498–519, 2001.

[16] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[17] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *CCS 07: Computer and Communications Security*, 2007.

[18] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE 07: International Conference on Software Engineering*, pages 499–510, 2007.

[19] Z. M. Saul, V. Filkov, P. T. Devanbu, and C. Bird. Recommending random walks. In *FSE 07: Foundations of Software Engineering*, pages 15–24, 2007.

[20] A. Tarvo. Using statistical models to predict software regressions. In *ISSRE 08: Software Reliability Engineering*, pages 259–264, 2008.

[21] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR 08: Information Retrieval*, pages 563–570, 2008.

[22] G. Venolia. Textual allusions to artifacts in software-related repositories. In *MSR 06: Mining software repositories*, pages 151–154, 2006.

[23] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE 08: International Conference on Software Engineering*, pages 461–470, 2008.

[24] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.