Automatically Identifying Known Software Problems

Natwar Modani[1], Rajeev Gupta[1], Guy Lohman[2], Tanveer Syeda-Mahmood[2], Laurent Mignet[1]
[1]*IBM India Research Lab, Block-1, IIT Delhi Campus, New Delhi, India.*
[2]*IBM Almaden Research Center, San Jose, CA, USA*
(namodani, grajeev, lamignet)@in.ibm.com; (stf, lohman)@almaden.ibm.com

Abstract

Re-occurrence of the same problem is very common in many large software products. By matching the symptoms of a new problem to those in a database of known problems, automated diagnosis and even self-healing for re-occurrences can be (partially) realized. This paper exploits function call stacks as highly structured symptoms of a certain class of problems, including crashes, hangs, and traps. We propose and evaluate algorithms for efficiently and accurately matching call stacks by a weighted metric of the similarity of their function names, after first removing redundant recursion and uninformative (poor discriminator) functions from those stacks. We also describe a new indexing scheme to speed queries to the repository of known problems, without compromising the quality of matches returned. Experiments conducted using call stacks from actual product problem reports demonstrate the improved accuracy (both precision and recall) resulting from our new stack-matching algorithms and removal of uninformative or redundant function names, as well as the performance and scalability improvements realized by indexing call stacks. We also discuss how call-stack matching can be used in both self-managing (or autonomic systems) and human "help desk" applications.

1. Introduction

One of the biggest challenges to self-healing systems is correctly diagnosing a problem based upon its externalized symptoms. However, typically half – and sometimes as much as 90 percent – of all software problems reported by users today are re-occurrences, or rediscoveries, of known problems, i.e. those whose cause has already been ascertained or is under investigation. Such rediscoveries present a significant opportunity for automatically repairing systems by searching a database of symptoms of known problems to find the best match with the symptoms of any new problem. But how to uniquely characterize any problem by its symptoms, and how to match those characterizations accurately, remains challenging, in general. Fortunately, there is a large class of software problems for which fairly structured symptoms can be used to characterize the problem, namely those that produce a function call stack among its symptoms. Systems typically generate function call stacks (we'll use the shorter term "call stacks" henceforth) when software "crashes", is terminated after a "hang", or an error is "trapped" and reported by the code itself. Call stacks reconstruct the sequence of function calls leading up to the failure via the operating system's stack of addresses that is pushed each time a function is called and popped when it returns. Call stacks typically contain at least the function name and offset in the routine at which each subroutine was invoked or the problem occurred or was detected. This paper uses the call stack as a symptom to characterize such problems.

Clearly, if two call stacks are identical, they almost surely represent the same problem, but what if they only partially match? The function in which the problem occurred is of course the most important to match, but that function may not be at the top of the stack if the code trapped the problem and invoked some standard routines to report and/or recover from the error, which provide little enlightenment on the nature of the problem. Furthermore, the path by which the execution got to the offending function may alter the values of key parameters that contribute to a problem, but the further in the stack we are from the offending function, the less likely that function is to have such an impact and the more likely it is to be common with other problem call stacks. And if the call stack contains recursive calls to the same functions, the number of recursions is rarely important for problem determination. Hence in our matching we want to weight matches nearer the top of the stack more heavily, after omitting redundant recursive invocations and the "uninformative functions" such as common error routines and entry-level routines.

1-4244-0832-6/07/$20.00

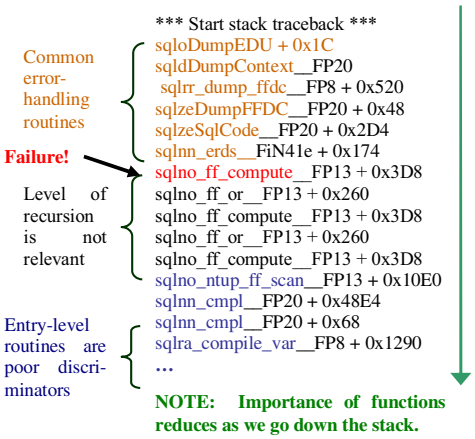

Figure 1: Typical call stack characteristics

An example of a C++ call stack from IBM[1] DB2[2] that illustrates these aspects is given in Figure 1, which for readability has omitted information such as the hex address, the C++ "mangled" names, the offsets, and the arguments of each function. Here the deepest function in the call flow is at the top, and the bottom of the stack contains the entry point function name. The failure occurs in the third recursive call of function *sqlno_ff_compute*, which calls the compiler error routine *sqlnn_erds* and subsequent routines to handle the error it detected. Functions below the compiler entry (*sqlnn_cmpl*) are very common, and hence are not good discriminators.

This paper evaluates the performance of algorithms for various components of a stack-matching system for automating the diagnosis of re-occurring problems by matching call stacks of new problems with those that may have been experienced before (and their remediation, if known), which are stored in a database. Specifically, we present algorithms for matching stacks, recursion removal, indexing, and identification of uninformative functions. The performance results show that our algorithms perform better than previously known ones.

The organization of the rest of the paper is as follows. Section 2 presents the prototype system architecture for known problem determination. We present our algorithms for stack matching algorithms in Section 3 and for normalizing the stacks by identifying the uninformative functions in Section 4. Section 5 discusses our indexing techniques to scalably search the database. We compare the performance of these algorithms using real-world data in Section 6, and discuss related work and conclude in Sections 7 and 8.

2. Prototype system

Call-stack matching represents an initial effort towards identifying already known problems from problem symptoms, either in a self-healing system or as an aid to "help desk" technicians. Figure 2 below shows the architecture of our implementation of the call-stack matching scheme. Our prototype is implemented using Java 1.4, accessing an IBM DB2 v8.1 database as its call-stack repository. Call stacks from previously reported problems are stored in the repository. A newly-arriving stack, which may be embedded in problem description text if coming from a human source, is parsed to extract the call stack and create a sequence of function names as the output[3]. Recursion and uninformative functions are removed, using the algorithms compared in the following sections, to get a normalized stack. This normalized stack is then used as the key of an index for searching for similar stacks from the stack repository, and for ranking how well each retrieved stack matches the query stack. If the query stack is not close enough to any of the stacks retrieved, it is deemed novel and will be inserted into the repository, so that it can be used in future searches.

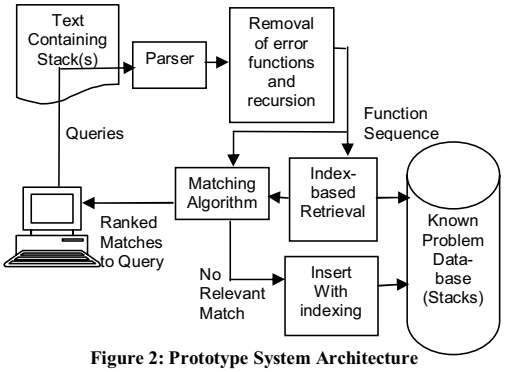

Figure 2: Prototype System Architecture

[1] IBM is a registered trademark of IBM Corporation in the United States, other countries, or both

[2] DB2 is a registered trademark of IBM Corporation in the United States, other countries, or both

[3] We ignore the offsets within functions to avoid differences introduced by different platforms, e.g., ENDedness and 32-bit vs. 64-bit.

Since the number of stacks in the repository can be huge, we create indices to search them efficiently. An index entry is computed from each normalized query stack. The index is used to retrieve from the repository a number of stacks that are likely to match the query call stack. These candidate stacks are sent as input to the matching algorithm. Algorithms for matching call stacks are described in later sections. The matching algorithm assigns a similarity score to each candidate stack with respect to the query stack. These matching scores are used to generate a ranked list of related stacks. We may limit the number returned from the matching algorithm by specifying either a threshold on the matching score or a fixed number of stacks to retrieve.

3. Stack matching algorithms

In this section, we present similarity measures and algorithms for call stack matching. These algorithms can be seen as variants of classical string matching algorithms, with modifications to exploit the particular characteristics of the normalized call stacks described above. Each of these matching algorithms assigns to a given pair of stacks a similarity score ranging between 0 for "no match" and 1 for "identical". These scores are used to select the subset of the call stacks that are deemed to be similar to the query call stack.

We first introduce some notation. We will denote the query stack by S and a stack from the repository as T. The stack consisting of a sequence of the first i function names from a stack Q is represented as Q_i. Also, the i^{th} function name in the stack Q will be denoted by q_i.

3.1. Edit Distance

Edit distance is a popular measure for string similarity [3]. Here we treat each function name in a call stack as a single symbol, and define the edit distance between the two call stacks as the number of edit operations (add, delete, replace) that are needed to convert the first call stack into the second call stack. The edit distance between two call stacks S and T can thus be defined in a recursive fashion as follows, using the notation defined above:

$$D_w(S_i,T_j) = \min\{ \begin{array}{l} D_w(S_{i-1},T_{j-1}) + w(s_i,t_j) \\ D_w(S_{i-1},T_j) + w(s_i,\phi) \\ D_w(S_i,T_{j-1}) + w(\phi,t_j) \end{array} \quad (1)$$

where ϕ represents the 'blank' (i.e., the edit operation is delete), D_w is the edit distance between the two call stacks, and w is the distance between two function names. We take $w(s_i, t_j)=0$ if the two function names from S and T, respectively (denoted by s_i, t_j) are the same, and 1 otherwise. This definition is amenable to a dynamic programming formulation. We then can define the similarity score between S and T as:

$$S_E(S,T) = (\max_i(|S|,|T|) - D_w(S,T)) / \max(|S|,|T|) \quad (2)$$

where |S| is the length of the call stack S. Clearly, $0 <= S_E(S,T) <= 1$. The space and time complexity of calculating $S_E(S,T)$ is $O(m^2)$, where m is the length of the stack.

3.2. Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) based similarity algorithm, proposed in [2], scores the similarity between two call stacks based on a longest common subsequence alignment cost, which is a function of the selectivity of functions, their positions from the top of the stack, and a gap penalty that increases exponentially with the extent of that gap. The alignment cost can also be computed using a dynamic programming formulation. The similarity score is obtained from the alignment cost as in Equation (2). The space and time complexity of computing the LCS-based similarity is $O(m^2)$, where m is the length of the stack.

3.3. Prefix Match

This algorithm is based on the basic premise that two call stacks arising from the same problem will have the same function names closer to the top of both stacks. In this algorithm, we find the longest common prefix by counting the number of consecutive functions starting from the top of the stacks that match across the two stacks. The size of this longest common prefix, normalized to the length of the longer of the two stacks, is taken as the similarity score of the two call stacks. More formally, the similarity score of the two stacks, based on prefix match, $S_p(S, T)$, is defined as:

$$S_p(S,T) = \arg\max_i\{S_i : S_i = T_i\} / \max(|S|,|T|) \quad (3)$$

The space and time complexity of this algorithm is linear in the length of the stack, i.e., $O(m)$. This algorithm can also take advantage of a lexicographically sorted list of call stacks to search the matches in $O(m \log n)$ time, where n is the number of call stacks in the list.

4. Stack normalization

In this section, we present techniques for identifying uninformative functions and removing them.

4.1 Recursion removal

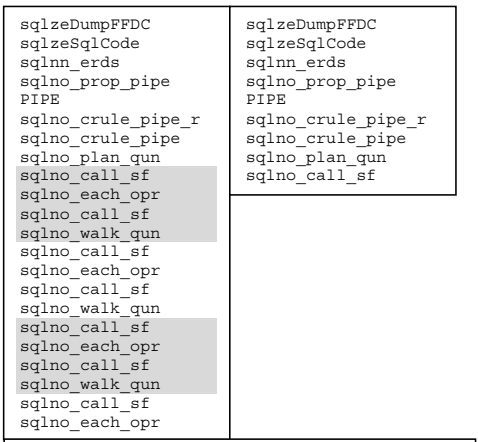

Figure 3: Call stack with and without recursion.

Recursive call sequences occur often in call stacks, due to the exploitation of recursion in programming. **Error! Reference source not found.**(a) shows an example call stack from IBM DB2 where, starting from the top, a sub-sequence (*sqlno_call_sf*, *sqlno_each_opr*, *sqlno_call_sf, sqlno_walk_qun*) is repeated three times, followed by the subsequence (*sqlno_call_sf*, *sqlno_each_opr*, *sqlno_call_sf*). The number of repetitions is typically dependent on the input parameter values. Since we do not exploit input parameters to match problem symptoms, we remove recursion in the incoming function call stacks before using them for matching. **Error! Reference source not found.**(b) shows the stack after recursion is removed.

For removing recursion, we keep two pointers: one starts from the top of the stack and the other from the bottom of the stack. If the function names pointed at by these two pointers match, then all the function names appearing between the top pointer and the bottom pointer, including the one at the bottom, are removed from the processed stack. In this case, the resulting stack is again searched for the presence of recursion.

```
removeRecusion(stack)
    for i=1:stack_length
        for j=stack_length:i+1
           if (si = sj)
             newStack= concat
      (stack(1:i),stack(j+1, stackLength))
           endif
        endfor
    endfor
end
```

Figure 4: Recursion removal algorithm

Figure 4 shows an outline of the recursion removal algorithm. The *concat* function creates a function sequence from the two input sequences by putting the second sequence below the first. The processed stack is used as input for iteratively applying the algorithm until we have all distinct function names in the processed stack. Our scheme removes recursion which the scheme in [[2]] fails to remove. For example, for stacks of form *abbbabbab* (where *a*, *b* are two function names) the recursion removal algorithm given in [[2]] will result in *ababab* whereas our algorithm would reduce it to *ab;* which is arguably a better representation. Furthermore, the complexity of this algorithm is $O(gm^2)$, for stack length m and number of functions appearing multiple times in the call stack g; whereas that for the algorithm presented in [[2]] is $O(m^3)$. In practice, g is likely to be much smaller compared to m.

4.2 Uninformative functions

Since all of our stack matching algorithms (correctly) weight the top of the stack more heavily, it is critical that uninformative ones such as error-handling routines be removed. The reason is two-fold. First, they are poor discriminators, resulting in false positives, because coding standards typically require that all functions call the same error-handling routines. Second, changes to the error-handling routines from one version to the next could cause false negatives (missed matches), even though the functional code invoking those routines has not changed. So removing such error-handling function names from the top of the stack is an important preprocessing step for the stack matching algorithms. One can get a list of such functions from the domain expert, of course, but for commercially-sized software, even the experts are challenged to provide a complete list accurately, since each component may its own error-handling routine. Hence there is a need for automatically detecting such uninformative function names. We developed two automated schemes for this purpose.

The first, **Function Frequency (FF)** algorithm, is inspired by the document frequency-based feature selection technique in classical information retrieval. We first find the frequency of occurrence of each function name across all the call stacks. All functions having occurrence frequency greater than some pre-established threshold are considered to be too common, and hence uninformative functions. Alternatively, one could also take the k most frequent functions and consider them as the uninformative functions. In our experiments, we take the second approach (with different values of k), as there is no principled way to choose the threshold. This frequency-based approach, while useful for information retrieval, can potentially label as uninformative function names that may not be error-reporting functions, but may be due to a particular function having many reported problems. More importantly, if a component does not have a large number of problems reported, this approach will not find the uninformative functions for that component.

Our second scheme, **Supervised Learning (SL)**, detects uninformative function names quickly and more reliably. Inputs to this scheme are a set of problem stacks along with their corresponding resolutions. If two stacks have the same corresponding resolution, they are considered to be duplicates of each other. Thus we have pairs of call stacks along with 'ground truth' indicating whether they are duplicates of each other or not. In this scheme, three counters are maintained for each function name, which are initialized to 0:

False Positive Count (FPC): FPC_f is incremented for all the matching function names f that appear above the first non-matching function name in a call-stack pair generated due to different problems. If the entire stack is identical, we increase the FPC_f for all the functions f in the stack.

False Negative Count (FNC): FNC_f is incremented for all the non-matching function names f that appear above the first matching function in a call-stack pair generated due to the same problem. If there are no matching functions, we increase the FNC_f for all the functions f in the two stacks.

Frequency Count (FC): FC_f is incremented for each appearance of a function f in any function call stack.

An aggregate measure (AM_f) is defined to be a non-decreasing function of FPC_f, FNC_f, and FC_f for each function name. Functions having a higher aggregate measure are likely to be uninformative functions. We chose:

$$AM_f = \frac{FPC_f}{FPC_{\max}} + \frac{FNC_f}{FNC_{\max}} + \frac{FC_f}{FC_{\max}} \quad (4)$$

Since this algorithm takes into account the usefulness of the function name in finding duplicate problems, we expect it to perform better than the Function Frequency algorithm.

5. Indexing

Indexing is used in databases to improve search performance. It reduces number of rows to which possibly complex filter is applied. For call-stack matching, various matching algorithms were described in Section 3. If n stacks are stored in the repository, the complexity of these matching algorithms will be either $O(nm)$ for the prefix matching algorithm or $O(nm^2)$ for the others. By using efficient indexing, we can reduce the effective value of n to improve the response time. However, we must ensure that any stack in the database that might match the search stack does not get rejected at the indexing stage. Thus we need to decide upon a key on which to index such that the indexing step will return a superset of call-stacks that are likely to match with the given query stack.

The solution to the above indexing problem is non-trivial, since there is no well-established definition of stack similarity measure, as we ourselves have presented three different measures for this purpose. In traditional databases, indexes are created for a particular "*where*" condition. For example, an equality index cannot be used efficiently for inequality or substring queries. We propose two method of indexing. Which indexing scheme to use may depend upon the matching algorithm, as well as some architectural issues, as explained later.

5.1 Top-k indexing

This algorithm is based on the intuition that two call stacks are likely to be due to the same problem if the top of the stacks is same. We store a hash key of the top-k function names for each stack in the repository. When a query stack arrives, we compute the hash of the top-k function names from it and query the repository for stacks that have the same hash key. The value of k can be chosen to control the number of candidates returned by the indexing scheme. While this scheme will not result in false-negatives for the prefix match algorithm, it may result in false-negatives for the LCS and edit distance based matching algorithms, as similarity score based on these algorithms may be reasonably high even if the top of the stack does not match.

```
topKIndexer(queryStack)
    kMax = Max function names to consider
    nMin= Min of stacks to return
    for i = 1 to kMax
        let n = number of stored stacks
        with the same top-i functions
        if(n < nMin)
            if (i== 1)
                k=1
            else
                k=i-1
return all stacks with same top-k
functions
```

Figure 5: Sketch of top-k based indexing algorithm.

In the help desk scenario described in Section **Error! Reference source not found.**, the service personal (L1) is presented a fixed number of results by the matching algorithm. In this situation, this indexing scheme can be further tuned to use the appropriate value of *k* automatically to minimize the number of stacks that need to be considered by the matching algorithm, thereby reducing the query response time.

Let *nMin* be the number of results required. Then the indexing scheme needs to return at least *nMin* stack ids. Figure 5 shows a sketch of this indexing algorithm. In this algorithm, *kMax* is the maximum number of function names that will be used for indexing. We start with *k*=1 and return stored stacks having the same *top-k* functions as the query stack such that *k* is maximum but the number of stacks returned is more than *nMin.*

5.2 Inverted Index

This algorithm is based on the observation that if a stack in the repository does not contain 'enough' function names that also appear in the query stack, it cannot get a high score from the matching algorithms. If we maintain a mapping from the function names to the stacks in which they appear, we can look it up to find the stacks that have a minimum number of functions present in the query stack. The mapping can be implemented using a hash table with the function names as the key and list of stack identifiers that contain this function as the value. As each new stack is stored in the repository, its identifier is appended to the lists of stacks for all the functions appearing in the stack. If a new function name is encountered, then a new entry is created for that function name and the incoming stack identifier is added as the stack list. Given a query stack composed of a sequence of function names, the hash table is searched for each function name appearing in the query sequence, and a multi-set is created from the corresponding stack identifiers. The indexing algorithm returns all the stacks that appear in the multi-set more often than a threshold *T*.

In the next section we evaluate the performance of various combinations of matching algorithms, indexing schemes, and uninformative informative function removal schemes, using the prototype architecture described in Section 2 0.

6. Performance measurement

In this section we evaluate the effectiveness of various steps of stack matching described above, using call stacks from commercial products. Data collection and performance parameters, along with the experimental methodology, are explained in Sections 6.1 and 6.2, respectively. In section 6.3 we compare the performance of the stack matching algorithms presented in Section 3. The effectiveness of our uninformative function removal algorithms is presented in Section 6.4.

6.1. Data collection

The experiments were conducted using call stacks from actual problem reports for IBM DB2 for Linux, Windows, and UNIX. When DB2 crashes, traps, or encounters a hang situation, it generates a 'trap' file, which includes one or more call stacks (one per process). Today, the customers e-mail this trap file to DB2 Service, along with other symptoms of the problem. We collected 525 such traps and the corresponding solutions from an historical database of problems containing call-stacks. For the purpose of establishing the ground truth for our experiments, two problems reported are considered to be duplicates of each other if they refer to the same solution. Using this criterion, we found that these problem records had 461 distinct call stacks (prior to normalization). Out of these 461 stacks, 142 stacks had recursion. This illustrates the need for recursion removal; otherwise, the similarity score would not be a true indicator of stack similarity.

6.2. Experimental methodology

Our performance measurement experiments started with an empty repository. Each call stack was then treated as a query stack, in the order of their historic arrival to the problem repository. We searched for a similar stack in the repository by setting a threshold on the similarity score. All problems corresponding to the stacks (from the repository) that have a similarity score higher than the threshold were deemed to be duplicates according to the algorithm in question. We then

counted the number of results returned and the correct results among them that were used to calculate performance parameters, as explained later. Query stacks not found to be similar to any in the database were immediately inserted into the repository, thereby making it available for matching for future queries. We repeated the experiments for different settings of the threshold and for different algorithms. As with any document search algorithm, the stack matching algorithms may return problem records which are not duplicates (false positives), and it may also miss some problem records which are duplicates based on the ground truth (false negatives). Hence, as in the information retrieval domain, a natural way to measure the performance of the matching algorithms is in terms of average precision and recall, which we now define.

Let n be the total number of results returned by the matching algorithm for all the queries, d be the number of (actual) duplicates amongst these returned results, and D be the total number of actual duplicates in the data set. We define the average precision p to be d/n and the average recall r to be d/D. Precision measures the fraction of results returned that are actually duplicates of the query stack. Recall measures the fraction of duplicates discovered by the matching algorithm.

6.3. Comparison of matching algorithms

First, we compare the performance of the three matching algorithms presented in Section 3. In this experiment, we removed any redundant recursion, but not any uninformative function names. Figure 6 shows the precision versus recall plot for the three matching algorithms. As expected, precision can be traded for higher recall. The Prefix Match algorithm demonstrated consistently higher precision for the same recall values than both the Edit Distance and LCS-based algorithms, although it never achieves a recall higher than 40%. While in theory the LCS and Edit Distance algorithms can achieve recalls up to 70%, for any recall higher than 50%, the corresponding precision drops below 5%, rendering them unusable. Also, the performance of the Edit Distance algorithm and the LCS algorithm does not seem to be very different. Hence, from here on, we present results only for the LCS and Prefix Match algorithms, for clarity.

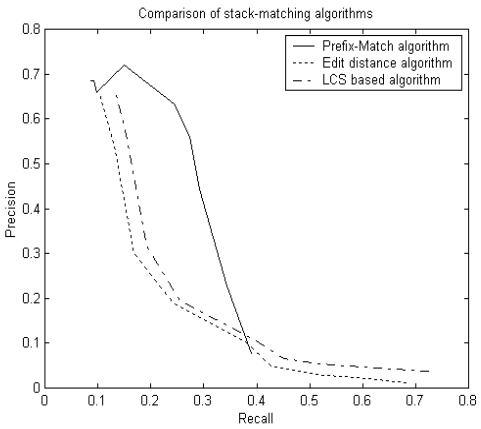

Figure 6: Comparison of stack matching algorithms.

6.4 Effectiveness of indexing

We have presented two indexing schemes in this paper. The *top-k* scheme is expected to speed up the query processing significantly. It will not change the precision and recall values for the prefix match algorithm (for k=1) but may increase the number of false negatives (D-d) for the LCS based algorithm, as explained earlier, thereby reducing its recall. The inverted index approach can be used with either matching algorithm without affecting their precision and recall (with threshold T=1). However, in this case, the speed-up is not expected to be as much as the top-k approach.

Since the query response time is more important in the help desk scenario, we simulated that scenario in our experiments by asking the matching algorithms to provide at most 5 recommendations for each query (instead of setting a threshold on the similarity score). Of course, if there were less than 5 problem records in the repository having a non-zero similarity score, the matching algorithms will return fewer results. The response-time as well as precision and recall for each experiment are shown in **Error! Reference source not found.**.

Matching Algorithm	Resp. Time (msec.)	Precision	Recall
Prefix Match (no indexing)	1813.00	62.81 %	36.53 %
Prefix Match + Top-K indexing	2.04	62.81 %	36.53 %
Prefix Match + Inverted Index	154.00	62.81 %	36.53 %
LCS (no indexing)	1888.00	52.69 %	42.31 %
LCS + Top-K indexing	2.13	60.34 %	39.90 %
LCS + Inverted Index	157.00	52.69 %	42.31 %

Table 1: Effectiveness of indexing.

The schemes with top-k indexing are almost two orders of magnitude faster than that the ones without indexing, whereas the inverted index improves response time by a factor of about 12. Further, response times of prefix match and LCS with the same indexing schemes are approximately the same. Both observations can be explained from the fact that the response time is dominated by disk access time. Since the candidates returned by top-k indexing will be a small subset of that returned by the inverted index (for corresponding values of k and T), top-k indexing is faster.

While we expected the recall reduction for the LCS matching algorithm when used in conjunction with top-k based indexing, the improvement in the precision is not so obvious. It is due to the fact that by discarding the stacks with mismatch right at the top, we discard most of the stacks corresponding to the non-duplicate problems.

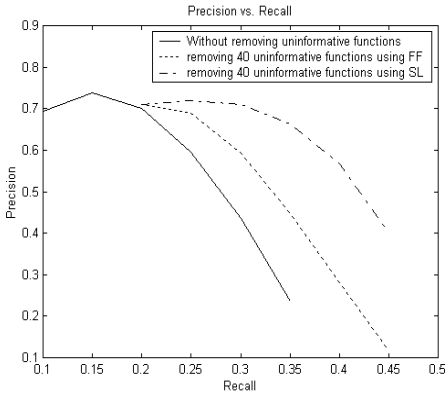

Figure 7: Comparison of the uninformative function name removal algorithms, for Prefix Match.

6.4. Removing uninformative functions

This section evaluates how well removing uninformative functions improves the precision and recall of our stack-matching algorithms. Figure 7 shows that the performance of the Prefix Match algorithm improves significantly by removing uninformative function names using the algorithms presented in Section 4. Similar improvements were observed with other stack-matching algorithms, as well. Note also that the Supervised Learning algorithm performs better than the Frequency-based approach. Another set of experiments were performed while removing the top 20, 40, and 60 most frequent function names. We observed no discernable performance improvement increasing the number of frequent functions removed from 40 to 60 functions removed. In general, removing uninformative functions increases recall values. However, for very short stacks, removing a large number of functions can actually reduce recall by removing vital information.

7. Related work

Brodie et al. [2] were the first to comprehensively address the problem of finding known problems by searching a database of symptoms, and the first to use call-stack matching for doing so. That paper introduced the need for recursion removal and identifying uninformative function names, and proposed the LCS algorithm that is evaluated in this paper. However, most of the discussion in this paper was qualitative with few experimental results. Further, this paper introduces alternative algorithms for each of the sub-problems, which perform better, as shown by our experiments. Another paper by Brodie and different co-authors [4] also uses call-stack matching for known problem determination, but does not attempt to identify the uninformative functions or remove recursive function calls, and uses a much simpler matching algorithm that finds the single longest common subsequence of function names.

Call stacks have also been analyzed for purposes other than problem determination. In [5], call stacks are used in an anomaly detection algorithm to detect unauthorized system intrusions. The collection of Java call stacks in a distributed environment is described in [6], and various techniques for comparing the call stacks are discussed.

The general idea of solving problems by matching symptoms against a historical database is also a well-known technique, known as case-based reasoning (CBR). It has been applied to customer support and help-desk situations [7], [8], but these approaches try to find similarities in the textual descriptions of problem reports supplied by users, not at the program execution level. They typically do not consider call stacks or the approach of looking for sequences in them.

Indexing of sequence databases for fast search continues to be an open problem in the string-matching literature, with most approaches using one or more heuristics to prune the search [9]. Among the approaches tried include preprocessing candidate matches and storing in suitable data structures (e.g., B-trees in BLAST [9]), the use of hash functions based on a collection of prefix strings, etc. The design of indexing schemes that ensure no false negatives while still reducing the number of false positives continues to be a challenging problem.

8. Conclusions and future work

In this paper, we presented algorithms for various components of a stack-matching system to identify automatically when software problems already have a known solution, and compared their performance. Our primary goal in the future is to extend this work to storing and matching symptoms less structured than call stacks.

9. References

[1]. Michael Buckley and Ram Chillarege, "Discovering relationships between service and customer satisfaction", IEEE International Conference on Software Maintenance (ICSM), 1995.

[2]. Mark Brodie et al, "Quickly Finding Known Software Problems via Automated Symptom Matching", IEEE International Conference on Autonomic Computing (ICAC), 2005.

[3]. Levenshtein V.I., "Binary codes capable of correcting deletions, insertions and reversals", Soviet Physics Doklady 10(8), pp. 707-710.

[4]. Brodie, M., Ma, S., Rachevsky, L, and Champlin, J., "Automated Problem Determination using Call-Stack Matching", Journal of Network and Systems Management, special issue on self-managing systems, June 2005.

[5]. Feng, H. H., Kolesnikov, O., Fogla, P., Lee, W. and Gong, W., "Anomaly Detection Using Call Stack Information", Proceedings of the 2003 IEEE Symposium on Security and Privacy, 2003, pg 62.

[6]. Lambert J. and Podgurski, A., "xdProf: A Tool for the capture and analysis of stack traces in a distributed Java system", International Society of Optical Engineering (SPIE) Proceedings, Volume 4521, 2001, pg 96-105.

[7]. Acorn, T., and Walden, S., SMART: Support Management Reasoning Technology for Compaq Customer Service, Innovative Applications of Artificial Intelligence, Volume 4, 1992.

[8]. Li, T., Zhu, S., and Ogihara, M., Mining Patterns from Case Base Analysis, Workshop on Integrating Data Mining and Knowledge Management, 2001.

[9]. Navarro G. et al, "Indexing Methods for Approximate String Matching", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2001.

[10]. Altschul S.F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," J. Molecular Biology 15 (1990), 403-410.