Technical Report SOCS-TR-2009.8

March 2009

Context-Sensitive Ranking of Dependencies for Software Navigation

Martin P. Robillard and Tristan Ratchford
School of Computer Science
McGill University
Montréal, QC, Canada
{martin,tratch}@cs.mcgill.ca

ABSTRACT

Software navigation during change tasks requires developers to perform numerous search cycles to discover new elements related to their task. In a typical search cycle a developer selects an element of interest, triggers a search for its dependencies, inspects the list of results, and selects a result that appears to be relevant. We are interested in improving the efficiency of developers engaged in program navigation by automatically searching for all dependencies to or from elements already discovered, and continually providing the list of dependencies, ranked in decreasing order of likelihood to be related to the task. In this paper, we present a framework for the systematic evaluation of ranking algorithms based on *multi-element contexts*, and report on a preliminary experiment to assess the value of three ranking strategies for dependencies: using the topology of the dependency graph, textual similarity, or combination of both. Our results show that all three strategies significantly improve the probability of recommending relevant elements in many situations, but that no ranking strategy appears to be universally optimal.

Keywords

Software Maintenance, Recommendation Systems, Software Investigation, Empirical Studies

1. INTRODUCTION

Developers engaged in a change task must typically *navigate* though source code to find and understand the program elements (e.g., fields or methods) that are relevant to their task. Program navigation is often assisted by search tools that can produce the list of elements that are structurally or textually related to the elements of interest. Search tools have been with us for a long time [18], and the functionality they provide is now fully and extensively integrated in modern development environments. For example, Eclipse can produce the list of elements that refer to a selected element through various shortcut keys, menu commands, and specialized views.

Unfortunately, using search tools involves a non-negligible overhead: users must select an element to serve as the search seed, trigger the search, inspect the list of results, and select the ones that appear to be relevant. Although it follows a straightforward process, searching code can nevertheless be inefficient: developers may waste time on, and then give up, the inspection of long lists of results [16], developers may have trouble identifying the right seed [11], etc. We believe that the efficiency of program navigation can be increased by automatically performing searches in the background and continually *recommending* elements that may be relevant to the current context.

Background searches were experimented with in state-of-the-art development tools such as Mylyn [8]. However, we propose to push the idea further by listing elements related to a development context *ranked in a context-sensitive order*. In other words, results of background searches should be ranked in an order that takes into account *the elements that have already been investigated.* This is in contrast to the standard structured/alphabetical order used by current integrated development environments (IDEs). By considering a *multi-element context*, this paper also expands on recent dependency ranking work by others [7, 14], who focused on the evaluation of dependencies to a single element (see Section 6). Considering multi-element contexts introduces many interesting challenges both for the design of ranking algorithms and for their evaluation, and addressing these challenges will help us advance towards the long-term goal of providing active information delivery [21] for software navigation.

Besides alphabetical order, three classes or alternative strategies can readily be envisioned to rank dependencies: based on properties of their structural association with the seed elements, based on textual similarity to the seed elements, or to use a combination of both strategies. However, given the myriad of factors that can influence the success of program navigation activities, it is difficult to predict whether advanced ranking strategies would result in efficiency gains, and whether one strategy would necessarily be more useful than another.

To compare different strategies as rigorously and reliably as possible, we designed a framework for the systematic evaluation of ranking strategies for dependencies to multi-element contexts where rankings are evaluated against benchmarks. We then conducted a series of experiments with this framework. Although human subjects were involved in the creation of the benchmarks, the experiments are not based on human behavior. The key idea of our experimental design is to sample a subset of elements from a benchmark,

obtain the references to these elements, rank the resulting elements, and measure how many top-ranked elements are also found in the benchmark. This design allowed us to produce robust and completely replicable results, while avoiding the risk inherent to the variability of human behavior under experimental conditions.

Our results show that the three strategies we assessed can significantly improve the probability of selecting relevant elements from dependent elements in many situations. However, we did not find statistically significant evidence of the superiority of a particular strategy, including the composite strategy, but instead discovered that different contexts tend to favor either structure-based ranking or textual similarity-based ranking. We conclude that the results of background searches should be ranked whenever possible, with various strategies available at the user's disposal.

The contributions of this paper include:

- An experimental framework to reliably evaluate dependency ranking strategies that explicitly takes into account multi-element contexts, and
- A detailed assessment of the value of advanced strategies for the context-sensitive ranking of dependencies in Java systems.

The remainder of this paper is organized as follows. In Section 2, we present the theoretical underpinnings for our experimental framework. In Section 3, we describe the three algorithms we evaluated as part of our experiment. In Section 4 we present the details of our experimental. We present our results in Section 5, discuss the related work in Section 6, and summarize the paper in Section 7.

2. THEORETICAL FRAMEWORK

Our experimental framework can be expressed precisely in terms of the concepts defined below.

Program

We assume the existence of a program P, which we abstract as a call graph $P = (M, C)$, where $m \in M$ is a method (or function) in the program, and C is the "method call" relation between two methods in the program. In the presence of dynamic binding, we consider that $(m_i, m_j) \in C$ if m_j is any method that can potentially be executed as the result of the call (e.g., all overriding or implementing methods).

Seed and Context

A *seed* $S \subseteq M$ is the set of elements identified by the developer as relevant to a task. In practice, a seed can be identified in various ways, for example by monitoring the interactions of a users with an IDE. A *task context* $T \subseteq M, T \supseteq S$ is the hypothetical set of all program elements related to the task at hand. In the situation we model with our experiment, the goal of the developer is to discover T as efficiently as possible given S. In contrast to previous work [7, 14], a novel element of our approach is to support the evaluation of situations where $|S| > 1$.

Search

A *search* is a procedure that takes as input a program $P = (M, C)$ and a seed S, and produces a *result* $R \subseteq M | R \cap S = \emptyset$. As an example of a search, we could return all the callers of methods in S.

Ranking Algorithm

A *ranking algorithm* $\text{rank}(P, R, S)$ is a procedure that orders a set R according to a particular strategy and in the context of a particular seed S. A ranking $\bar{R} = \{R_1, ..., R_n\}$ is an *ordered* version of set R (which we denote with an overbar on the variable name). The set is ordered in decreasing values of *confidence* that the element will be relevant to the developer (i.e, that it will be included in the corresponding task context T), as determined by the ranking strategy. As an example of a simplistic ranking algorithm, we could sort elements in alphabetical order. Naturally, T is not a parameter of $\text{rank}(P, R, S)$ because at the time of a search we do not know what the task context is.

Filtering

Given a ranking $\bar{R}$, we define the operation $\text{filter}_n(\bar{R})$ to return the $\min(n, |\bar{R}|)$ elements of $\bar{R}$ with the highest rank. For example, $\text{filter}_2(\{m_1, m_2, m_3\})$ would return $\{m_1, m_2\}$. As a shorthand we represent $\text{filter}_n(\bar{R})$ as $\bar{R}_n$.

Ranking Precision

The precision of a ranking is a measure of the number of relevant elements in the ranking. We use the standard definition of precision from the field of information retrieval [1]. For a task context T and a ranking $\bar{R}$, the precision is defined as:

$$\text{precision}(\bar{R}, T) = \frac{|\bar{R} \cap T|}{|\bar{R}|}$$

We can use this formula in the same way to calculate the precision of filtered rankings $\bar{R}_n$.

3. RANKING ALGORITHMS

Given a seed S, we consider a single type of search that produces all of the callers and callees of all the methods in S (i.e. all the neighboring nodes in the call graph according to our definition of a program). We use this search as a model of multiple sequential cross-references queries, in part to produce larger result sets that can more fully test the ranking strategies. We then apply different ranking algorithms to the results of this search. The rest of this section provides the details of the ranking algorithms we studied.

3.1 Random

RANDOM ranks the elements of a set in no particular order. This algorithm will be used as a baseline for comparing the results of the other algorithms. We did not use Eclipse's default alphabetical ranking as a baseline because it potentially interferes with the evaluation of the approach based on textual similarity, and because we would not be able to explain why a general purpose scheme such as alphabetical ranking could be meaningful in the context of a task, except by accident.

3.2 Topology Analysis

Topology analysis of software dependencies (or "TOPOLOGY") is a technique to produce recommendations for software navigation based on the topology of a software dependency graph [10]. The idea behind this algorithm is to rank elements based on the closeness of their structural association with the elements of the seed, and where "closeness" is defined by a pair of heuristics. We study this ranking algo-

rithm as a representative of strategies based purely on structural analysis. The complete description of the approach, including the definition of the metrics used, can be found in a separate paper [10]; The following is a summary (adapted from [19]).

The topology-based algorithm takes into account two main characteristics of the call relations between elements: *specificity* and *reinforcement*. Specificity evaluates the "uniqueness" of a call between a seed element and a recommendation. For example, if a seed element x is called by five other elements, $x_1, ..., x_5$, and another seed element y is called by two other elements, y_1, y_2, then we say that y_1 and y_2 are more specific to the seed than $x_1, ..., x_5$. Based on the intuition that specific elements are more strongly related to the seed than less specific ones, our algorithm ranks more specific elements higher than less specific ones.

Reinforcement evaluates the strength of the intersection between the seed and a result set. For example, if a seed element x is related to five other elements $x_1, ..., x_5$, and four of these elements $(x_1, ..., x_4)$ are already in the seed, then we say that the remaining element (x_5 in our case), is heavily reinforced. On the other hand, if none of the elements in the result set are also in the seed, we do not consider the elements to be reinforced. Based on the intuition that reinforced elements are more strongly related to the seed than unreinforced elements, our algorithm ranks reinforced elements higher than less reinforced ones.

The algorithm works by separately analyzing the "calls" and "called by" relations. First, it obtains, for each element in the seed, the set of all elements related to it by the relation type currently analyzed. For example, for the relation type "called by", we obtain all callers of each method in the seed. We then use a formula to produce, for each related element, a degree of potential interest for the element that is based on our specificity/reinforcement criterion. We then merge the results of the analysis of each relation. In the end, our algorithm produces a single ordered set of elements directly related to the seed.

3.3 Textual Similarity

We implemented an algorithm that ranks elements based on how textually similar they are to the elements in the seed ("TextSim"), as a basic representative of ranking techniques based on an analysis of the keywords used in definitions and comments.

TextSim calculates a similarity coefficient between each method in a seed S and each method in a result set R. The higher the coefficient assigned to a pair of methods, the more textually similar they are. Methods in R are then ranked based on their *highest* similarity coefficient with *any* method in S.

TextSim calculates the similarity coefficient for a pair of methods using the Vector Space Model [6]. With this approach, each method is modeled as a vector of index terms. A term vector is a mapping of the number of occurrences of terms within a particular *document*.

We analyzed the source code of all methods to produce vectors based on terms in the method's name, the names of the variables declared within its scope, the names of its parameters, as well as its associated comments (both regular and JavaDoc). Within a JavaDoc comment block, the general comment text and the text of the comments for the following tags were used: *@param*, *@return*, *@exception* and *@throws*. We then obtain similarity coefficients by taking the cosine distance of a pair of methods' respective term vectors. When normalized, the resulting values will range from 0 to 1, where 1 is a perfect match between term vectors and 0 implies no common terms.

Before generating the final term vectors, we used the Apache Lucene[1] text indexing and searching library to remove all stop words (such as *the, and, if*) and punctuation from each text block and convert all text to lowercase. Furthermore, we broke up compound terms, such those separated by hyphenation, or camel case, into separate terms.

To address a number of well-known shortcomings of the basic vector model, we weighed the terms in each vector using the tf-idf scheme. A tf-idf vector of term weights has two components: the term frequency (tf), and the inverse document frequency (idf). Intuitively, the higher the tf component, the more strongly the term represents a method.

The second component of tf-idf is the inverse document frequency (idf). The idf captures the intuition that the more methods that include a term, the less that term can be used to discriminate between different methods. Combining the two components using a standard formula yields the final tf-idf weights [1].

3.4 Topology–Textual Similarity Hybrid

The Hybrid algorithm combines Topology with TextSim by computing a ranking by averaging the rank of each element produced by the two other algorithms. In other words, if an element was ranked first using Topology and third using TextSim, the element would receive the average of these two and would be ranked second with the Hybrid ranking. Collisions are resolved by pseudo-randomly ordering elements with the same calculated rank. Our initial intuition is that Hybrid would be superior to Topology and Hybrid because it promotes elements that are related *both* structurally and textually to a seed.

4. EXPERIMENTAL SETUP

We designed our experimental framework to simulate the generation of a recommendation in a situation where a developer has uncovered a small number of relevant methods (the *seed*), and is interested in investigating methods that are related to the seed, but has no a priori knowledge about which of the related methods are likely to be the most interesting. Naturally, if the developer knows the code well, they can safely ignore the recommendations. For example, a developer working on the *line folding* feature in a text editor discovers a few methods related to folding in a `Buffer` class. Since these methods are referenced by dozens of other methods, the developer must perform many searches to find the next method to inspect. Instead, we can try to recommend this method by obtaining all the dependencies to the already-discovered `fold` methods and recommending the top-ranked ones for inspection. Using the terminology of Section 2, we formalize our experimental framework as follows.

[1]http://lucene.apache.org/

Obtaining Benchmark Data

1. **Target program.** Choose m target programs $P_i = (M_i, C_i)$.

2. **Task Context.** Given a program $P_i = (M_i, C_i)$, determine n task contexts $T_{i,j} \subseteq M_i$. A task context should include elements that are related to the implementation of a clearly-defined high-level concern or feature that could realistically be associated with a software modification task.

This part of the experimental setup results in a total of $m \times n$ benchmark task contexts.

Running Trials on Task Contexts

For each task context T_i:

1. **Samples.** Randomly generate n seeds S_i of p elements such that $\forall\ S_i, S_i \subseteq T$. The seeds S_i thus model a set of elements a developer has identified as interesting. For our initial set of experiments, as described in this paper, we chose seeds of two elements because a single element contains too little information to properly indicate the intention of the developer. As for higher numbers, we expect that the larger the seed size, the better the results, so we chose to use two elements to test the ranking algorithm in the most unfavorable conditions. In practice, the number of possible distinct seeds is bounded by the number of distinct sets of elements that can be formed from the elements of T ($\binom{|T|}{p}$). To strike a balance between a good support for statistical testing and practical feasibility, we chose a baseline of $n = 20$. Therefore, for each task context we generate $\min(20, \binom{|T|}{2})$ sample seeds.
2. **Search.** For each seed S_i, produce R, the result of a search defined at the beginning of Section 3.
3. **Rank.** Apply all the ranking algorithms to R, to produce the sets $\bar{R}_{\text{Topology}}$, $\bar{R}_{\text{TextSim}}$, and $\bar{R}_{\text{Hybrid}}$.
4. **Measure.** Measure the precision of R. This precision corresponds to the probability of randomly choosing a task context element from the set of results. Measure the precision of the filtered versions of sets ranked according to the various algorithms, using a filter value of 5. For example, $\bar{R}_{\text{Topology},5}$. This precision corresponds to the probability of randomly choosing a task context element from the set of five most highly ranked results according to the ranking algorithm. We chose a value of 5 based on experience [10], as a reasonable measure of results that a developer would consider before considering giving up looking through the list. We do not measure recall because the algorithms are not designed to recreate the task context, but instead to recommend one or two elements to the developer's attention.

We consider the trial a success for an algorithm a if:

$$\text{precision}(\bar{R}_{a,5}, T_i) > \text{precision}(R, T_i)$$

In other words, we consider that our ranking had value *for this task context and sample seed* if choosing from the top elements improves a developer's chance of inspecting an element in the task context.

Applying Statistical Tests

The previous test can determine if a particular algorithm has value for a given task context and seed. To be able to interpret the results at a higher level, we perform statistical testing on the n seeds derived for a task context to see if any observed difference in top-5 precision is statistically significant *for the given task context.*

4.1 Statistical Analysis

Essentially, differences between rankings can be expressed in terms of differences in the probabilities of randomly choosing a task context element. The intrinsic value of a ranking algorithm can be determined by comparing the ranking of the algorithm with randomly-ordered results. For example, if 3 out of 10 elements in a result are in the task context, the chance of randomly choosing a task context element regardless of how they are ranked is 0.3. If an algorithm ranks the results and 2 of the 3 task context elements are in the top 5, then randomly choosing from the top 5 elements results in a 0.4 chance of selecting a context element. In this case, we would consider the ranking to be valid because it increases the chance of finding a relevant element in the list of results. For all trials, we calculate the difference in probability

$$\text{precision}(\bar{R}_{a,5}, T_i) - \text{precision}(R, T_i)$$

where a is Topology, TextSim, or Hybrid.

Similarly, we can compare algorithms between them. For the purpose of our experiment, we compare Topology and TextSim against each other, and both Topology and TextSim against Hybrid.

To generalize our results from individual trials to entire task contexts, we must show that any observed difference is statistically significant across all samples considered. For this purpose, we strove to use a statistical analysis that could demonstrate the superiority of an algorithm for a given task context while being as easy to interpret as possible. Because we cannot make any assumption about normality in our population data, we use the Wilcoxon signed-rank test. This procedure tests the null hypothesis that scores are distributed symmetrically around a specified central value (0 in our case). In other words, if no algorithm is better than the other, roughly half the probability differences will be negative, and the median will be close to 0. We can reject the hypothesis if the estimated median is significantly non-zero (two-tailed test), or greater than 0 (one-tailed test).

We used one-tailed tests to compare all three algorithms with Random, as well as Hybrid with both Topology and TextSim, because our initial hypothesis is that all algorithms should beat Random, and because we suspected that Hybrid should beat both its components taken individually. We used a two-tailed test to compare Topology and TextSim because we had no a priori theory about which should be superior.

4.2 Benchmark Data

Our experiment relies on the presence of benchmarks in the form of task contexts, namely, groups of methods that could reasonably be associated with a task.

We derived our benchmark data from the results of a previous empirical study of manual concern location [13]. In this previous study, different human subjects were asked to identify the source code for 16 different features (also called

Table 1: Target Systems

System	Version	LOC
GanttProject	2.0.2	43 246
Jajuk	1.2	30 679
JBidWatcher	1.0pre6	23 051
Freemind	0.8.0	70 435

"concerns") in four non-trivial Java systems (Table 1). The feature descriptions provided to the subjects consisted of a few paragraphs of text explaining how the feature worked as mentioned in the user manual, help pages, or user interface of the system. For example, for the GanttProject system, one feature had to do with allowing users to add relationships between tasks in a Gantt chart.

For each feature, three different subjects were asked to identify the fields and methods that were judged to be the most relevant to the implementation of the feature. The subjects were undergraduate and graduate students with Java programming experience in Eclipse. No method, process, or tool (besides the features of the Eclipse environment) were given to them to complete this task. This process resulted in $16 \times 3 = 48$ different mappings between a feature and source code. We directly reused these task contexts with only one small modification: we removed the fields from all contexts, since the search operation we study in our experiment is based solely on call dependencies between methods. Of the 48 modified benchmark contexts, 8 have from 3–5 elements (inclusive); 18 have 6–10 elements; 13 have 11–15 elements, and 9 have more than 15 elements. Because context sizes of less than 6 elements cannot produce at least 15 distinct seeds (i.e., combinations of 2 elements), we had to discard these contexts as unusable for the purpose of our statistical analysis.

Overall, the 40 remaining task contexts represent sets of scattered code elements pertaining to different concerns, in different systems, and identified by different subjects. We consider this data to represent a good range of variability in the program navigation scenarios that could be encountered by developers. In particular, the fact that we have multiple contexts produced by different subjects for the same task allows us to interpret the sensitivity of our results to different personal styles of program navigation. Moreover, large variation between task contexts for a common concern, which we observed in [13], allows us to consider all task contexts as pseudo-independent (i.e., not *necessarily* correlated). In prior work [13, Table 2], we measured the level of similarity between each context for a given task, and our overarching result was that contexts produced by different developers vary widely. Even the most similar contexts did not come even close to being equal. As evidence, for the task with the most *similar* contexts (c10), only 6 out of a total of 23 elements had been included by all three subjects.[2] Overall, the contexts for GanttProject and Jajuk were the least similar, and the contexts for JBidWatcher and Freemind were the most similar. We take this lack of complete independence into consideration in our interpretation of the results. Our benchmark data is available online.[3] It includes the source code of the four systems analyzed, the descriptions of all features considered, and the original feature mappings (i.e., task contexts) produced by all the subjects.

[2]This number corresponds to the original analysis, which included fields in the task contexts.

[3]http://www.cs.mcgill.ca/~martin/concerns

4.3 Threats to Validity

In designing our experimental framework, we favored a strong quantitative design that would facilitate evaluation of many program navigation situations, that would not require human judgment to compare the algorithms, and that could be completely and independently replicated. These benefits are naturally counter-balanced by a number of limitations, most having to do with the artificial and controlled context in which we conduct our comparison of the three algorithms. Three consequences of our experimental design are particularly noteworthy.

Artificial Contexts

Although our benchmark task contexts were created by human subjects, they were not collected through actual program change tasks. However, analysis of our tasks contexts clearly shows that different people have different views of what constitutes a relevant task context. As a result, our benchmarks are not necessarily better or worse than data that would have been collected by interviewing a developer at the end of a task. As an advantage, having different contexts for a given concern allows us to better draw conclusions about the performance of the different algorithms for a specified task.

Approximate Performance Model

Our main measure of performance is to compare with a random selection of elements through a list. However, people do not necessarily choose elements at random: their intuition, experience, and numerous other factors come into play. Randomness is the only objective way we could approximate behavior, considering that personal characteristics of subjects may lead to either better or *worse* performance than RANDOM (for instance in the case of mistaken assumptions about the design of the system).

Approximate Seed Selection Model

We generate our seeds by randomly selecting two elements from the task context. In practice, developers would probably not identify elements of interest randomly. For the seeds, however, the fact that both elements are selected from the tasks context means that they are at least conceptually related.

5. RESULTS

We first present the results of the basic assessment of our three algorithms (against RANDOM), and then the results of the comparative assessment of algorithms against each other. For each assessment, we present our raw results and derive high-level observations from the data.

5.1 Basic Assessment

Table 3 reports the results of the comparison of the three ranking algorithms with random selection. Each row corresponds to the trials for one task context. In the first column, we list the name of the (task) context as listed on our website (see Section 4.2). The rows are grouped by concern (thin lines) and by system (thick lines). Contexts 01–04 are

Table 2: Successes vs. Random

System	Topology	TextSim	Hybrid
GanttProject	6/12	3/12	4/12
Jajuk	5/9	7/9	6/9
JBidwatcher	12/12	10/12	11/12
Freemind	7/7	7/7	7/7
Total	30/40	25/40	28/40

defined on GanttProject, contexts 05–08 on Jajuk, contexts 09–12 on JBidWatcher, and contexts 13–15 on Freemind (all three task contexts for concern 16 were eliminated due to insufficient elements in the context, see Section 4.2).

For each row, the table is organized into three major sections corresponding to the evaluation of the performance of a given ranking algorithm with respect to random ranking. Each major section comprises two column listings of the score median estimated by the test (Med. Est.), and the level of statistical significance of the test (p-value), respectively. A statistically significant result for a positive median indicates that there is a statistically significant observed increase in the probability of choosing a context element by selecting one of the top-5 elements provided by the ranking: In other words, the algorithm was reliably successful for that context. Results significant at the 0.05-level are indicated with an asterisk (*), and results significant at the 0.01-level are indicated with a double asterisk (**).

Scanning down the p-value column for a ranking algorithm, asterisked values identify contexts for which the given algorithm would have been useful. For example, going down the column for TOPOLOGY, we see that the algorithm succeeds in 30 out of 40 of the cases, and that it succeeds for all the concerns of the JBidwatcher system (c09–c12). Alternately, by scanning a given row we can see which of the algorithms beat random for a given context. Table 2 summarizes the success level (considering any results significant at the 0.05 level or more to be a success).

Observations

The data of Tables 3 and 2 provides partial answers to our research question and allows us to make a number of high-level observations about the success of context-sensitive ranking of search results.

General success level

Given the approximate nature of context-sensitive ranking algorithms, we naturally expect that rankings will not always be useful. However, our data shows that all the ranking algorithms we evaluated were successful at least for a majority of contexts (63–75%).

Comparative success level

All three algorithms also fared more or less equally compared to a random baseline. In particular, contrary to what we had expected, HYBRID was not necessarily superior to its individual component algorithms.

System-relative success level

Perhaps the most surprising aspect of our results is the apparent sensitivity of the ranking algorithms to the different systems they are applied on. Although there is not enough data at this level of granularity to validate the relation statistically, the results of Table 2 are eloquent: with very few exceptions, all three ranking algorithms were successful for all task contexts defined on JBidWatcher and Freemind (the smallest and largest of our target systems, respectively). In contrast, the algorithms fared reasonably well on Jajuk and not well at all on GanttProject. A detailed analysis of the data for GanttProject explains the poor performance of the ranking algorithms for three difficult contexts (c01-p14, c02-p06, and c04-p09). For these contexts, many of the seeds contained at least one element that was not structurally related to any other element in the context, making it very unlikely to reliably produce meaningful rankings with our metric, for any algorithm considered. This was not the case for c01-p07 and c03-04, however. For these context, we found no obvious characteristic of the system that could readily explain the lack of results for the three algorithm, which may simply have been a coincidence.

Context-relative success level

Taking into account the similarity between the benchmark task contexts documented in a previous report [13, Table 2], we noticed that the four most similar contexts (c10, c11, c12, c13) also show a strong correlation in the results of the algorithm. Although the correlation can be explained analytically, it is not clear why the results are positive because there is no a priori relation between how similarly humans produce task contexts and how successful synthetic ranking algorithms perform on these contexts. The impact of this context similarity on our results is simply one of proportion: In Table2, for JBidWatcher, instead of considering the algorithms successful in 12/12 cases (for instance), it is probably more reasonable to consider that it was always successful, but in 4–5 cases. Except for the most similar contexts just discussed, it is impossible to predict the results of the algorithms based on the similarity of the underlying contexts. For instance, results for contexts c09-* are positive across the board, but the different versions of this context are far from similar (only 3 common elements).

5.2 Comparative Assessment

Evaluating the differences between algorithms yielded much fewer statistically significant results. This can easily be explained by two factors. First, while the precision of an unfiltered result set R can (theoretically) take any value, the precision of a filtered result set R_5 can take only six distinct values (0/5, 1/5, 2/5, etc.). As a result, the difference between the filtered precision calculated for two algorithms can take only 11 distinct values (0/5, ±1/5, ±2/5, etc.). Because of a lower granularity in the range of possible difference scores, the likelihood that any distribution will be symmetric is increased. Second, as discussed in Section 5.1, all three ranking algorithms fared more or less equally against RANDOM, so we could not expect large differences between algorithms. In fact, inspection of the individual trial difference scores shows many sets of 0-valued differences. Attempts to increase the range of differences by increasing the filter value to 50% of the total result set did not yield more results. We thus report our initial findings with a filter window of 5.

Table 5 follows the same format as Table 3, but reports on the results of the tests comparing the algorithms between themselves. Again, tests involving HYBRID are one-tailed, whereas the comparison of TOPOLOGY with TEXTSIM

Table 3: Results of the Wilcoxon Signed-Rank Test, Comparison with Random

	Topology/Random		TextSim/Random		Hybrid/Random	
Context	**Med. Est.**	**p-value**	**Med.Est**	**p-value**	**Med.Est**	**p-value**
c01-p03	0.015	0.048*	0.015	0.0428*	0.015	0.0691
c01-p07	0.015	0.2523	0	0.5157	0.015	0.1359
c01-p14	-0.03	0.9991	-0.03	0.9848	-0.03	0.9848
c02-p03	0.06	0.0312*	0.03	0.1329	0.07	0.0204*
c02-p04	0.075	0.0012**	-0.04	0.994	0.015	0.2256
c02-p06	0	0.6946	0	0.336	0	0.6946
c03-p04	0	0.3114	0	0.3114	0	0.3114
c03-p07	0.045	0.1652	0.1625	0.0017**	0.105	0.0053**
c03-p08	0.14	0.0005**	0.14	0.0015**	0.14	0.001**
c04-p09	0.0275	0.0763	-0.05	0.9986	-0.015	0.7303
c04-p11	0.195	0.0004**	-0.03	0.9933	0.085	0.0069**
c04-p40	0.0075	0.0155*	-0.0075	0.9912	0	0.1404
c05-p04	-0.015	0.5	0.06	0.0071**	-0.015	0.5
c05-p11	0.1275	0.0012**	0.0475	0.0366*	0.08	0.0024**
c06-p04	0.155	0.0045**	-0.035	0.8185	0.085	0.0344*
c06-p08	0	0.2376	0.0125	0.0286*	0.0575	0.0121*
c07-p05	0.04	0.012*	0.07	0.006**	0.085	0.0042**
c07-p08	-0.0325	0.8804	0.0825	0.0396*	-0.065	0.981
c07-p10	0	0.4498	-0.04	0.993	-0.02	0.6933
c08-p12	0.37	<0.0001**	0.17	0.0003**	0.335	<0.001**
c08-p13	0.095	0.0002**	0.125	0.0002**	0.085	0.005**
c09-p10	0.135	0.0037**	0.135	0.003**	0.115	0.0097**
c09-p12	0.165	0.0002**	0.14	0.0006**	0.195	0.0004**
c09-p13	0.115	0.0013**	0.17	0.0004**	0.17	0.0004**
c10-p14	0.0075	0.0289*	0	0.2026	0	0.0774
c10-p21	0.05	0.0103*	0.125	0.0016**	0.125	0.0016**
c10-p23	0.085	0.0016**	0.11	0.0013**	0.12	0.0013**
c11-p23	0.2825	0.0002**	0.25	0.0002**	0.26	0.0002**
c11-p24	0.13	0.0004**	0.235	<0.0001**	0.185	<0.0001**
c11-p25	0.285	0.0002**	0.255	0.0002**	0.28	0.0002**
c12-p07	0.065	0.0173*	0.05	0.0824	0.0675	0.0173*
c12-p23	0.175	0.0004**	0.145	0.0005**	0.1875	0.0003**
c12-p26	0.195	<0.0001**	0.2025	0.0003**	0.23	<0.0001**
c13-p21	0.205	0.0002**	0.15	0.0002**	0.2125	0.0002**
c13-p27	0.1125	0.0151*	0.17	0.0004**	0.185	<0.0001**
c13-p28	0.31	<0.0001**	0.1	0.0044**	0.1525	<0.0001**
c14-p21	0.11	0.0019**	0.2075	0.0004**	0.175	0.0004**
c14-p28	0.1	0.017*	0.225	0.0003**	0.155	0.0036**
c15-p29	0.14	0.0007**	0.1725	0.0002**	0.165	0.0002**
c15-p32	0.075	0.0012**	0.015	0.0496*	0.0475	0.0113*

is two-tailed. A positive estimated median indicates that TOPOLOGY was superior for that test (and vice-versa). In the interest of space, we only include data points where we have a statistically significant rejection of the null hypothesis. Because it compares the two most different algorithms, the Topology/TextSim column contains the most significant values.

Table 4 summarizes the success level (considering any results significant at the 0.05 level or less to be a success). The results for the TOPOLOGY/TEXTSIM comparison are split across two columns to distinguish cases where TOPOLOGY beats TEXTSIM (positive median) and the inverse (negative median).

Table 4: Inter-Algorithm Successes

System	**Hybrid/ Topo**	**Hybrid/ Text**	**Topo/ Text**	**Text/ Topo**
GanttProject	0/12	2/12	4/12	0/12
Jajuk	0/9	2/9	3/9	1/9
JBidwatcher	3/12	0/12	0/12	2/12
Freemind	1/7	2/7	1/7	2/7
Total	4/40	6/40	8/40	5/40

Table 5: Results of the Wilcoxon Signed-Rank Test, Inter-Method Comparisons

	Hybrid/Topology		Hybrid/TextSim		Topology/TextSim	
Concern	**Med. Est.**	**p-value**	**Med.Est**	**p-value**	**Med.Est**	**p-value**
c02-p04			0.05	0.0098**	0.1	0.0009**
c04-p09					0.1	0.0107*
c04-p11			0.1	0.0021**	0.2	0.0005**
c04-p40					0.05	0.0197 *
c05-p11					0.1	0.0401*
c06-p04			0.1	0.0205*	0.2	0.0092**
c07-p08					-0.1	0.0187 *
c08-p12			0.1	0.001**	0.2	0.0082**
c09-p13	0.05	0.0098**			-0.05	0.0197*
c10-p21	0.05	0.0098**				
c11-p24	0.05	0.0098**			-0.1	0.0027**
c13-p21			0.05	0.0098**		
c13-p27	0.1	0.0117*				
c13-p28			0.1	0.0024**	0.2	0.002**
c14-p21					-0.1	0.0248*
c14-p28					-0.1	0.0243*

Observations

These tables help us answer our research questions and provide additional insights on the relative success of each algorithm.

Evaluation of HYBRID

Based on our evidence the value of a hybrid approach is not demonstrated. Comparisons with RANDOM (Table 2) show that HYBRID did not help increase the number of cases where a ranking is useful, and inter-algorithm comparisons (Table 4) show only a few cases where HYBRID is a better option. From Table 5 we also see that the cases where HYBRID beats TOPOLOGY and/or TEXTSIM are in fact mutually exclusive, so we have no evidence of cases where HYBRID would be the absolute best strategy.

TOPOLOGY *vs.* TEXTSIM

Although in a given context one algorithm may be better suited than another, the finding that overall TOPOLOGY is successful more often is corroborated by both the comparison with RANDOM and the inter-algorithm comparisons.

5.3 Conclusions

We are encouraged by our findings that context-sensitive ranking of dependencies seems to improve the chance of selecting elements relevant to a task in a majority of cases. Although much more work is needed to make context-sensitive ranking a practical reality, our initial results pave the way for additional research on the topic, in particular by showing that one promising avenue for improving the success of the approach as a whole seems to be through the early determination of the type of ranking algorithm that may be the most appropriate to a given task context.

6. RELATED WORK

As part of our previous work on the development of the topology analysis of software dependencies [10], we evaluated the TOPOLOGY algorithm using an experimental framework different from the one described in this paper. We also performed the experiment on a different flavor of the algorithm (which considered field accesses), considered different filtering sizes (1–5), and used a different statistical test (generalized estimating equations). Our experience with this work led to the improved (and simpler) framework described in this paper. Also, we conducted our earlier experiment on different benchmark contexts defined on four target systems other than the ones used in this paper. Despite all the differences, the overall results showed that choosing from the top results based on topology analysis of software dependencies improved the odds of selecting a context element in most, but not all, of the scenarios considered. Our most recent results confirm this initial finding. In contrast to the above mentioned previous work, however, this paper focused on the methodological and analytical problems of comparing different ranking algorithms, of which TOPOLOGY is only one instance.

We have also tried to compare the value of topology analysis of dependencies through comparative studies involving human subjects using a variety of tools [4]. These studies have been much less conclusive, primarily due to the large influence of the difficulty of the task on the subjects. The challenge encountered in this prior work partially motivated our efforts to develop a quantitative evaluation framework for ranking algorithms.

Related work by others include that of Hill et al. [7], who proposed a sophisticated textual similarity algorithm for ranking structural dependencies to a single seed element. Their algorithm takes as input a method and produces its "neighborhood" of structural dependencies (what we call the search result), ranked according to their textual-similarity

metric. For this reason, although the paper describes the approach as integrating lexical and structural information, we comparatively consider the approach to be only textual similarity-based because the only structural analysis performed is to extract calling dependencies (which must be done for all algorithms in our experimental framework). However, in addition to the standard term-based textual similarity analysis exemplified by our TEXTSIM implementation, their technique also weighs terms based on where it appears (e.g., method name vs. local variable), and uses logistic regression to determine the optimal weights for terms. Hill et al. evaluate their technique on the same benchmark set as the one used for this paper, and include a comparison of with a version of TOPOLOGY (labeled "Suade" in the paper).

Another related approach is that of Saul et al., who recently proposed two techniques to analyze software dependencies to identify code related to a function of interest, which they also evaluated through synthetic experiments [14]. One of their approaches (FRAN) attempts to rank its neighboring dependencies using a strategy based on a random walk of the neighboring dependency graph; the other (FRIAR) uses sets of functions commonly called together to make inferences about which function is the most related to a seed element. Their evaluation of both approaches (which included a comparison with a version of our TOPOLOGY algorithm), involved feeding 330 functions of the Apache project to determine how well each approach could rediscover benchmark elements, using statistical analysis to determine significance, as we have done.

In their evaluation, both Hill et al. and Saul et al. compare the performance of their ranking algorithm to that of the Suade algorithm [10] which we have used as our TOPOLOGY algorithm, and report superior performance for their technique. However, both papers only report on evaluations using *single element contexts*. This choice analytically explains the inferior performance of TOPOLOGY because, for single element contexts, the better part of the TOPOLOGY algorithm is not exercised (see Section 3.2) and the remaining ranking heuristic is too simple to produce reliable rankings in a majority of cases. More importantly, the limitation of the experiments to single element contexts makes it practically impossible to determine how the different approaches would fare on multi-element contexts. In brief, although ranking dependencies to single elements potentially offers valuable benefits to developers who explicitly trigger a cross-reference dependency search, the long-term goal of our research is to facilitate active delivery of recommendations through background searches based on a non-trivial context of interactions. To facilitate advancing towards this goal, we provided an experimental framework that makes provisions for multiple-element contexts, as well as a comparative evaluation of relatively simple baseline techniques.

A wide variety of other systems exist that provide recommendations on elements to visit by using a wide variety of data sources and analyses. We discuss representative examples. *Feature location* approaches attempt to identify program elements related to a high-level concepts, with varying degrees of automation. Originally developed based on dynamic analysis [20], the idea has also been investigated using other techniques such as natural language analysis [15], or using hybrid approaches [9]. In the context of this paper, feature location approaches are more elaborate than the cross-reference searches we investigated, as their goal is generally to identify *all* elements related to a concept, as opposed to helping developers identify elements of interest among the results of a given query. Similar to feature location techniques, *impact analysis techniques* [2] usually involve the analysis of software dependencies (e.g., through coupling measures [3]) to produce an assessment of the ripple effects of changes in a software system. The goal of feature location and impact analysis techniques is to provide an extensive set of source code locations meeting a criterion, as opposed to the scenario of supporting incremental program navigation we intend to support.

Program navigation analysis techniques involve monitoring the source code elements visited by a developer in an integrated development environment, and using this information to support the work of developers [5, 8, 12, 17]. These approaches are conceptually similar but vary in the type of interaction data collected and analyzed, and in the nature of the information provided to developers. Among the approaches cited, Mylyn [8] is noteworthy in that support for context-based background searches was developed for it, albeit without ranking the results in a context-sensitive manner. This feature was not released, however.

Yet another class of approaches recommend source code elements related to a task using an analysis of the revision history of a software system [22, 23]. Although one could envision designing quantitative experiments to validate most code recommendation approaches, the experimental framework we contribute in this paper only applies to approaches where the seed (queried element) is of the same type as the elements found in the results, and where the seed is conceptually associated with the results.

7. SUMMARY

Software navigation is generally assisted by search tools that produce results that are not ordered in a task-meaningful way, requiring developers to scan long lists of results to find potentially relevant elements. Automatically recommending developers structurally-related elements that are likely to be pertinent to their task has the potential to increase the efficiency of developers navigating source code. To help assess the value of ranking schemes for dependency recommendations based on multi-element contexts, we designed a completely replicable experimental framework. We used our framework to evaluate three different ranking algorithms based on: dependency structure, textual similarity, and a combination of both. Our results show that all three algorithms significantly increase the chance of selecting a relevant element from the search results in a majority of the benchmark cases studied. All three algorithms also fared almost equally well *overall*, but for individual cases we observed many situations where one type of fundamental approach works very well while the other does not. The main conclusion we draw from this observation is that it may not be desirable to apply a universal composite algorithm that may be weaker than some of it components. Ultimately, characteristics of the task could guide the choice of algorithm used to rank background search results. Future research should help determine what these characteristics are, and how we can take them into account for choosing the ranking strategy.

Acknowledgments

The authors are grateful to Nachi Nagappan for his valuable insights on our statistical analysis, and to Barthélémy Dagenais and Gail Murphy for comments on the paper. This work was funded by NSERC and IBM.

8. REFERENCES

[1] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[3] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. pages 475–482, 1999.

[4] Brian De Alwis, Gail C. Murphy, and Martin P. Robillard. A comparative study of three program exploration tools. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 103–112, 2007.

[5] Robert De Line, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, 2005.

[6] A. Wong G. Salton and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[7] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 14–23, 2007.

[8] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–11, 2006.

[9] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[10] Martin P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4):1–36, 2008.

[11] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[12] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234, 2003.

[13] Martin P. Robillard, David Shepherd, Emily Hill, K. Vijay-Shanker, and Lori Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, 2007.

[14] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–24, 2007.

[15] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, pages 212–224, 2007.

[16] Jonathan Sillito, Gail Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 23–34, 2006.

[17] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, 2005.

[18] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–33, April 1981.

[19] Frédéric Weigand-Warr and Martin P. Robillard. Suade: Topology-based searches for software investigation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 780–783, 2007.

[20] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995.

[21] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 60–68, 2000.

[22] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[23] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th ACM/IEEE International Conference on Software Engineering*, pages 563–572, 2004.