Understanding the Impact of Support for Iteration on Code Search

Lee Martie, André van der Hoek, Thomas Kwak
University of California, Irvine
Department of Informatics
Irvine, California 92697-3440, U.S.A.
{lmartie, andre, tkwak1}@uci.edu

ABSTRACT

Sometimes, when programmers use a search engine they know more or less what they need. Other times, programmers use the search engine to look around and generate possible ideas for the programming problem they are working on. The key insight we explore in this paper is that the results found in the latter case tend to serve as inspiration or triggers for the next queries issued. We introduce two search engines, CodeExchange and CodeLikeThis, both of which are specifically designed to enable the user to directly leverage the results in formulating the next query. CodeExchange does this with a set of four features supporting the programmer to use characteristics of the results to find other code with or without those characteristics. CodeLikeThis supports simply selecting an entire result to find code that is analogous, to some degree, to that result. We evaluated how these approaches were used along with two approaches not explicitly supporting iteration, a baseline and Google, in a user study among 24 developers. We find that search engines that support using results to form the next query can improve the programmers’ search experience and different approaches to iteration can provide better experiences depending on the task.

CCS CONCEPTS
• Software and its engineering → Software notations and tools;

KEYWORDS
Code search, internet-scale, iterative

1 INTRODUCTION

Unlike when people search on the Internet to be aware of the latest news or current temperature, a distinctly informational activity [60], programmers routinely search for source code on the Internet [13, 68, 72] when they are looking for solutions to aid in their current programming problem [19]. Sometimes the programmer uses a search engine to find one specific code snippet. For example, the programmer might need to remember how to write a few lines of code (e.g., the code to open a database in PHP) [15, 25]. Other times, the programmer uses a search engine when she is not quite sure what she is searching for and there is not exactly one code snippet she has in mind. For example, the programmer might need to learn a concept [15, 66], such as database transactions, and needs to look at multiple examples illustrating different aspects of databases and alternative examples to clarify her understanding [15, 25, 36, 61, 66]. For another example, the programmer might need to get ideas [72], such as when designing a new game and wants to see, and sometimes compare [25], how other code handles game characters, board state, or visualization.

Given how important searching for code on the Internet is to programmers, researchers are investigating how to improve code search engines. Some, for instance, have been investigating how to support more expressive queries (e.g., searching by test case or method signatures) that afford more precise matching of code compared to keywords (e.g., [1, 10, 17, 37, 41, 45, 54, 59, 70, 75]). Others have investigated new matching and ranking algorithms (e.g., ranking code higher with method names or class names matching the keywords) so that more results presumed to better match the topic described by the keywords are returned and appear towards the top of the list (e.g., [14, 20, 27, 29, 35, 42, 43, 49, 79]).

While many different approaches for improving code search exist, these approaches are generally similar in one very visible design decision: they are non-iterative approaches. They expect a query and optimize on returning the best matching results for the query, occasionally offering filters to help scope the results (e.g., programming language or file type filters) [2, 5]. This focus on a non-iterative design for search engines is mirrored in how search engines are evaluated [46]. Typically, a group of experts score the performance of search engines by the results returned for some representative set of queries, with the score reflecting how on topic the results are.

While a search engine that returns the code the programmer is looking for after the first query appears ideal, many times the programmer is not sure what she is looking for and does not search for code with a single query. Instead, the programmer issues multiple queries [12, 15, 34, 67, 71], where, after receiving results, the
programmer modifies their query by removing keywords, adding keywords, or some combination of both, and repeats this process multiple times [12, 34, 67]. That is, search looks like an iterative process where programmers often submit a query, get results, reflect on and learn from the results, submit a modified query in response to the results, get new results, and so on, until the programmer stops searching.

Cognitive processes in which programmers engage possibly explain why code search is often iterative. Particularly, when programmers are working on a programming problem, what they are working on, a solution, is often not immediately understood [19, 32, 69]. However, as programmers begin to look at some code or consider possible ideas, they are faced with constraints or different perspectives not previously considered, changing their understanding, and a new understanding will often change the next code and ideas considered [19, 21, 44, 53]. The implication of this on code search is that, when programmers search for code not clearly understood (e.g., cases when learning or needing ideas) code results can cause them to change their understanding of what they are looking for and, thus, the next code searched for – making the search iterative.

Our research investigates what happens when programmers are explicitly supported in searching iteratively for code. It particularly answers the following research question:

**What is the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process on the Internet?**

The key insight we explore to support iteration is that the code returned for a query tend to serve as inspiration or triggers for the next queries issued. We introduce two search engines, Code-Exchange (CE) and CodeLikeThis (CLT), specifically aimed to enable the user to directly leverage the results in formulating the next query. CE [47], previously developed but now built on the Specificity ranking algorithm [43], provides a set of four features supporting the programmer to use characteristics of the results to find other code with or without those characteristics. For example, if a result is undesirable because it is too complex, then the user can refine her query to find code that is less complex than the undesirable result. Rather than using particular characteristics, CLT supports simply selecting an entire result to find code that is analogous, to some degree, to that result. For example, if the user receives an implementation of an AI for chess but wants to see other similar approaches to learn from, then she can select the entire result to find other similar approaches.

We conducted a user study with 24 developers evaluating the iterative approaches with two non-iterative approaches (a baseline and Google). The baseline was a control used to measure the impact of a lack of iteration support, while maintaining the same code index used in the iterative approaches. As such, the baseline was created by removing the iterative features from CE, leaving a traditional looking search engine. While Google is not a code search engine per se and indexes a much greater amount code on web pages than our iterative approaches, it is important to evaluate the most popular form of search today [6] to gain an understanding of how developers iteratively search with it.

The rest of the paper is organized as follows. Section 2 presents relevant background in code search. Section 3 introduces each of the search engines used in the user study. In Section 4 we discuss the design of the experiment. Section 5 presents the results in detail. Section 6 presents threats to validity. In Section 7 we conclude with the implications of the results and future work.

## 2 BACKGROUND

Previous research in code search can be divided into empirical studies on how developers search for code on the Internet and tool research that seeks to provide new ways of supporting code search on the Internet. In this section, we present a summary of both groups of research.

### 2.1 Empirical Studies on Internet Code Search

Several types of studies (e.g., surveys, search log analysis, field studies, and lab studies) have been conducted to understand why and how developers search for code. Surveys asking programmers why they search for code on the Internet have been conducted by Sim et al. [66], Sadowski et al. [61], Stolee et al. [72], and Hucks and Graham [36]. Overall, these survey studies find that the motivations to search for code on the Internet include getting ideas/inspiration, learning, remembering, clarifying knowledge, and finding code to reuse as-is.

Lab studies have been conducted looking at how programmers search for code by Scott Henninger [30], Sim et al. [67], and Brandt et al. [15]. Henninger and Sim et al.’s findings suggest that searching for code is highly iterative, spanning a sequence of queries (from 2.38 to 7.25 on average), where each new query is often a modification of the previous (often making it more specific, called a refinement, or less specific, called a generalization). Further, Brandt et al.’s findings confirm the motivation to search found in the survey studies above.

Studies by Bajracharya et al. [12], Brandt et al. [15], and Holmes [34] examined patterns in search engine logs to understand how and why programmers search for code. Overall, these studies confirm that programmers submit multiple queries on average and some of their motivations include learning new concepts or being reminded of how to accomplish a programming task.

Lastly, Rosalva Gallardo-Valencia et al. [25] conducted a field study onsite at a company in Peru where they observed employees search for code on the Internet. The study found that the majority of searches were concerned with learning, remembering, gaining a deeper understanding, solving a bug, translating from English to Spanish, and comparing candidate solutions.

### 2.2 Tool Support for Code Search on the Internet

Research has explored a wide range of tools and techniques to support code search on the Internet. These tools are organized in the following six subsections.

#### 2.2.1 More Expressive Queries

Recognizing that keywords do not allow developers to easily target their search to the content
of code, several search tools support structural queries, usually submitted with an advanced query form. These approaches support search by the code’s method signature [65], packages [33], framework [77], and language constructs (e.g., if statements) and relationships (e.g., one method calls another) [43].

Rather than searching for functionality based on the structure of the code, several approaches have investigated how to support developers in more meaningfully searching for specific functionality by supplying semantic queries. A common approach is to use test cases as queries to find code passing them [37, 41, 59, 78].

Several approaches support the user to write part of the code she needs (at the method or statement granularity) and to submit it as a query to get results that complete it. This approach is intended to more seamlessly go from code to results with no keyword query in between [16, 17, 51, 54, 62].

2.2.2 Better Ranking Algorithms. Other research has investigated how to improve the ranking of the code that is returned. To return more on topic results, one approach automatically adds related words to the current keywords to match topically related code that would be missed by matching only against the programmer’s keywords. The terms added can come from a variety of thesauruses [42], rule systems mapping keywords to related terms [20], related Java documentation [28], or code the developer is currently writing [14]. A similar, but different approach is to index code in the search engine not only with terms occurring in it, but also with descriptive terms elsewhere [18, 33, 77, 82].

Linstead et al. found that it is possible to improve ranking performance for code by matching keywords against the most qualified parts of fully qualified type names in code (e.g., class names) in an algorithm called Specificity [43].

2.2.3 Query Creation Support. Several approaches attempt to eliminate the effort of formulating queries by automatically constructing queries on behalf of the user and continuously pushing results to the user. The insight behind these approaches is that the context (e.g., opened projects) of the developer can determine, in part, some initial queries [8, 27, 35, 79].

While not Internet code search approaches, local code search approaches, for various maintenance tasks (e.g., finding a method to change in a local project), support the programmer in replacing or completing their keyword queries when results are too few or deemed inadequate for the maintenance task. These approaches utilize issue tracking systems [29] and various statistics in the project [26, 58].

2.2.4 Result Usability. While returning topically related code to a query is crucial, other research has noted that the usability of the results in terms of their quality, understandability, and ease of integration are also important for search. To control quality, several approaches match against code that is more popular. Measuring popularity has been done by counting the number of times code is used by other code [43, 49] and extracting high level patterns from the code indexed and counting how often those patterns occur in the search engine [22, 39, 50].

Another critical part of code search is the ability of the user to understand the code results. One very early approach supported the user to select parts of the code to issue “why” questions to retrieve manually created documentation explaining the selected parts [24]. Another approach collapses code results into groups by different functionalities in the code and supports annotations [63, 64]. Some other methods include documentation and comments from the web with the results [57, 80], summarizations [81], or examples of usage [52].

2.2.5 Result Navigation. Traditionally, navigation of code search engine results is done by paging through 10 ranked results at a time. However, some code search engines, many commercial, support navigating the results by scoping them with descriptive fields called filters. For example, the commercial search engine Krugle [3] supports scoping results by known projects, file types, and authors of the code indexed.

2.2.6 Iteration Support. Little preexisting work supports how developers search. In particular, little work exists on using the results to create the next query. Henninger’s work around 1994 was first to present iteration as an issue and demonstrated in CodeFinder that it is possible to recommend keyword refinements from the results by using the spreading activation algorithm on a LISP repository of 1800 snippets [31]. Henninger showed the recommendations helped find code for ill-defined code search tasks. Mica [74] offers Java SDK specific refinement recommendations by recommending keywords in the results that also occur in the Java SDK libraries (shown to be used in half the queries in a field study). More recently, Bajracharya et al. presented ideas, yet to be evaluated, on recommending query refinements by function calls and types used in functions occurring in the results [11]. However, much more research can be done beyond refinement recommendations, and it is important to do so given that in 100% of the empirical studies that looked at the behavior of code search suggest that code search on the Internet is often iterative.

3 SUMMARY OF SEARCH ENGINES

In this section, we describe the details and rationale behind the iterative approaches and briefly introduce the baseline approach. Since Google is well known in general, we do not include a description.

3.1 CodeLikeThis (CLT)

Sometimes it is easier for people to recognize something that resembles what they want rather than to say what they want [38, 56]. To support using a result that does or does not resemble helpful code in order to find other helpful code, we designed a new method of search by similarity, implemented in CLT. After the first keyword query, CLT presents the user with a diverse set of results, using the Hybrid diversity ranking algorithm [48], where the results are on topic, but diverse across other characteristics (e.g., libraries used, authors, and implementations). From this diverse set of results, the developer has different kinds of results to select as a query (called a like-this query) to find code that is more, somewhat, or less similar to the selected result. Code returned from the like-this query can continually be used to issue another like-this query to support iterative search.

A screen-shot of CLT is presented in Figure 1 illustrating how to use a quick sort example to find many different kinds of quick sort implementations and other ways of sorting. On the bottom
left in Figure 1 is the main page showing the top 10 results after a developer issued the keyword query quick sort. At the bottom of each code result are buttons to issue a query to find other code that is less, somewhat, or more similar to that result. Shown above and to the right of the main page are the top two results after clicking on each of “like this” buttons for the highlighted implementation of quick sort on the bottom left. When the programmer selects the “More Like This” button, she gets results (A) that are also quick sort implementations, but use different styles and methods to implement quick sort (i.e., similar quick sort implementations but not exact clones). When the programmer clicks “Somewhat Like This” on the quick sort implementation, she gets results (B) that rely more on other classes (e.g., extending parent classes to implement quick sort) or include comments in other human languages. Lastly, when the programmer clicks “Less Like This”, she gets results (C) that are no longer quick sort implementations, but are examples of other kinds of sorting algorithms (in this case merge sort and heap sort).

3.1.1 Like-This Ranking Algorithm. To process a like-this query \(Q\) on selected result \(R\), CLT performs the following algorithm:

1. Find and order the top \(N\) code snippets by their similarity to \(R\), where similarity is calculated with the \(\text{Sim}^{ST}_2\) function [48].
2. If \(Q\) is a more-like-this query, return the top 10 of \(N\).
3. If \(Q\) is a somewhat-like-this query, return the 10 code snippets that are an average distance away from \(R\).
4. If \(Q\) is a less-like-this query, return 10 at the tail of \(N\).

In our implementation, we set \(N = 300\) in an attempt to limit completely off topic results from a less-like-this query and to speed up our processing of queries.

3.2 CodeExchange (CE)

Sometimes particular characteristics of a code result appear helpful and sometimes not. To support using characteristics of the results to find other code with or without those characteristics, CE [47] provides four features. Each of these features are presented in Figure 2 (a partial screen-shot) in the context of CE after the user has been searching for an implementation of an HTTP servlet. Each feature is discussed next.

3.2.1 Language Constructs Feature. Language constructs (A) support the developer in searching by structural characteristics she likes about a code result. In particular, language constructs highlight the structural properties of the code results so that they can be clicked on to refine the query by that structural property. For example, the user can click on a method call (e.g., setContentType) or an import (e.g., javax.servlet) in a result to refine the query to find all code that also has that method call or import.

3.2.2 Critiques Feature. Critiques (B) support the developer to search by what she does not like about a code result. In particular, language constructs highlight the structural properties of the code results so that they can be clicked on to refine the query by that structural property. For example, the user can issue a query that the results need to have more/less size, complexity, or number of imports than the result she does not like. This is done by clicking an up/down arrow above the size, complexity, or number of imports appearing above the result, and when clicked will refine the query to find results with more/less
size, complexity, or imports. For example, if a code result is simply too long for a user to want to read through, she can click the down arrow below the size value for that result and the next results will be shorter in size compared to that result. Clicking the down arrow below the size value for the snippet in Figure 2 would refine the query to find code less than 3155 characters.

### 3.2.3 Refinement Recommendations Feature

Refinement Recommendations (C) use the results after each query to present ways for refining the current query by domain related keywords or common imports, parent classes, and interfaces implemented. These recommendations alleviate the work for the developer trying to see what is commonly used or related in the entire set of results returned (often in the thousands). For example, if the developer issues the keyword query HTTP servlet, she gets keyword refinement recommendations request and response (both having a domain specific meaning in HTTP servers) and parent class recommendation HttpServlet. When the recommendation for the parent class HttpServlet is taken, the query will be refined for code that extends the class HttpServlet and the recommendations are updated using the latest results. In this way, the developer can iteratively take recommendations to refine her query to steer the search engine toward desired results.

### 3.2.4 Query Parts Feature

Query parts (D) decompose the query by the result characteristics and keywords used to refine the query and can be toggled on/off. When a part is toggled off, it acts to generalize the query by deactivating the refinement and when toggled on reactivates it to refine the query. When a query part is on, it appears appear yellow, and when it is off, it appears white. In Figure 2, the user has generalized the query by toggling the keyword query part off. In this way, the programmer can quickly modify her query by previous result characteristics to try different combinations in response to the current results.

CE’s features are orthogonal to any ranking algorithm for keywords. In this study, it uses the Specificity ranking algorithm [43] because Specificity is a successful code ranking algorithm compared to more basic ranking algorithms (e.g., TF-IDF).

### 3.3 Baseline

The baseline search engine (shown in Figure 3) was constructed to resemble a traditional search engine, while controlling for confounding factors. Our method to do so was to remove all the iterative features of CE, leaving basic search features, but preserving the same ranking algorithm of CE and the same code index used by CE and CLT. The basic features included a keyword text box with autocomplete, a list of code results, and a paging mechanism. We called our baseline search engine “SearchIt” when introducing it to participants in order to hide the fact we were using it as a baseline.

### 4 EXPERIMENT DESIGN

To answer our research question, what is the impact of explicitly supporting developers in searching iteratively, we conducted a user study measuring the experience, time, and success of each participant in searching for code with each approach [23, 40]. Further, we logged how each approach was used as an indication of what features helped, and we collected reasons why code was chosen to give insight into why code is used on the Internet when the search for code is initially uncertain.

The participants in the study consisted of 24 developers who reported to have approximately 4 years of professional development experience on average (s = 2.67), above intermediate Java skill level (median of 5 skill level on an ordinal scale of 1 as beginner and 7 as expert and with s = 1.03), an average age of 26.2 (s = 3.61), and a 20/4 male-female ratio. We recruited the participants by sending out advertisements to industry affiliated mailing lists targeted towards developers [4].

The user studies were held in a closed lab setting where the participants sat alone in a room completing eight different and independent search tasks, in sequence. Each participant was assigned two search engines for the entire experiment and completed each search task using only one. We chose to set the number of search engines to two per person to reduce the learning curve effect with using all four search engines, but still allowing them to make comparisons between the search engines in their feedback.

We designed the experiment as follows:
We used the Latin Square [73] design for our experiment to evenly distribute the tasks among the search engines and participants. As such, each search engine was used in 48 tasks in total and used for each task six times, which yielded a total of 192 data points.

To address ordering effects, the search engines alternated with each task and all the tasks came in a random order for each participant. This was accomplished by assigning each search engine to a task (done with Latin Square) creating pairs like $(S_1, T_1), (S_2, T_2), (S_1, T_3), \ldots, (S_2, T_8)$, and when a participant finished a task, then a pair with the other search engine was chosen at random and used as the next task and search engine to use.

Each participant received a different task for each search engine, so each participant never repeated a task on two different treatments — making the experiment a between subject design to reduce carry over effects. In addition, each participant used different search engines, making it also a within subject design. As such, we had a mixed design.

To control for differences in code indexed by the search engine, CE, CLT, and the baseline search engine all index the same 10 million Java classes mined from github.com. With respect to Google, in Section 6 we discuss implications about its index.

Lastly, to push participants to give each task some thought and effort, we asked them to include explanations for what the code does and why they chose it. This was a tactic to get the participants to genuinely attempt each task.

The search tasks were designed to cover a space of tasks that are broad to more focused. The broad tasks were designed to model situations when the programmer is looking for multiple and possibly different examples that can help. The more focused tasks were designed to model situations when the programmer is looking for one example or examples of a particular kind. This space of tasks is presented in a 2x2 matrix shown in Table 1. The broader tasks are found in the “Find 4” row and the “No Specific Role for Code” column. The more focused tasks are found in the “Find 1” row and the “Algorithm/Data Structure” column. The tasks in the upper right and lower left are a mix of the broad and focused tasks. The participants had 20 minutes for the “Find 4” tasks and 10 minutes for “Find 1” tasks. We set the time limits for completing tasks based on our pilot studies, where we found our participants could finish tasks in the given time limits.

The topics of the tasks were created to mirror real-world topics for code search. The topics were derived by reverse engineering plausible topics from real queries (tic tac toe - T1, mail sender - T2, AWT events - T3, combinations n per k - T4, array multiplication - T5, database connection manager - T6, JSpinner - T7, and binary search tree - T8) found across four different code search engine logs [48]. With the topics, we created the tasks in a style similar to those used in other code search studies [31, 45], where the tasks do not give the participant a query or the actual code snippet to find (as is usually not done in the real-world) and are composed only of one or two sentences expressing a high-level problem for which the participant needs to find code to help solve.

To start the experiment, the participants watched a tutorial video on each of their assigned search engines that explained all features (including the advanced search feature for Google). Further, to warm up, each participant had a few minutes to play around with the search engines as they pleased. Once done, each participant used a survey system, which presented the time she had for a question, a search engine hyper-link indicating which search engine to use (when clicked opened the search engine), the search task, five tabs each containing an editor to paste the code she finds for the search task, a text box to explain what the code does, and a text box to explain why she chose the code. We gave the participants five tabs in case they wanted to find more than the required number of snippets. When a participant hit the done button to indicate finishing or when time ran out, the participant then rated her experience, from 1 to 7, for using the assigned search engine for the given task, where 1 was labeled “bad”, 4 was labeled “neutral”, and 7 labeled “great”. Once the participant was done giving her experience rating, she hit submit and received the next task and treatment. Finally, once the participant finished all eight tasks, she filled out a questionnaire about the treatments used and then had an open interview with us.

From the entire experiment, we collected a total of 192 experience scores, 192 time durations to find one snippet (also counting time to find first snippet in the “Find 4” tasks), 96 time durations to find four snippets, and 24 interviews. Further, we logged the search behavior across all the search engines, giving us data for what features were used and how often. We also collected 463/480 reasons why code was chosen (each participant was asked to find 20 snippets in total and give a reason why she chose each snippet).

5 RESULTS

In this section, we examine the impact of supporting iterative search on the experience, time, and success of searching for code. Further, we look at how the search engines were used and the reasons people reported for searching for code. For significance tests, we set alpha to 0.1 as done in smaller-scale experiments [7].

5.1 Experience, Time, and Success

Our first analysis compared the experience scores for using each search engine for each search task. To compare experience scores, we observed when one approach had a higher median score than others. It is standard practice to use the median and not mean when the data is ordinal, such as the experience scores. We created the
Table 1: Task Matrix

| No Specific Role for Code — Broader | Algorithm/Data Structure — More Focused |
|-------------------------------------|----------------------------------------|
| **T3**                             | **T1**                                  |
| **Scenario:** You are building a sketching application. Task: Find 4 snippets of Java source code that you think will help. | **Scenario:** You are making your favorite video game. Task: Find 4 snippets of Java source code that you think will help implement the algorithms and data structures. |
| **T7**                             | **T5**                                  |
| **Scenario:** You are building a program to survey people’s preferences. Task: Find 4 snippets of Java source code that you think will help. | **Scenario:** You are building a program to help teach students basic algebra. Task: Find 4 snippets of Java source code that you think will help implement the algorithms and data structures. |
| **T2**                             | **T4**                                  |
| **Scenario:** You are building a new email client. Task: Find Java source code that counts new messages. | **Scenario:** You are teaching a class and need to split your class up into groups of four. Task: Find Java source code that implements an algorithm and data structures to find all possible groups of students. |
| **T6**                             | **T8**                                  |
| **Scenario:** You are building a large healthcare patient record keeping system. Task: Find Java source code to add or edit records in the system. | **Scenario:** You are building the world’s first online phone book. Task: Find Java source code that implements an algorithm and data structures to retrieve phone numbers by name. |

box plot summary, shown in Figure 4, of the experience scores for each search engine by task. Each box is color coded by search engine, shows the median score with a black horizontal bar through it (sometimes on top or bottom), and has a height summarizing the spread of the scores. For each task, the search engine with the highest median scores has its plot annotated with its initials. If there are ties between search engines, then both names appear above their corresponding plots.

We found that an iterative approach existed with a higher median than the baseline for six tasks and equal median for two tasks ($p < 0.1$, where $p = 0.002$ with $\chi^2$ on a $2 \times 3$ contingency table comparing best medians). These results suggest supporting iteration with the features of CE and CLT can significantly (using Bonferroni correction to set alpha to 0.05 for comparing Iterative twice) improve the developers’ experience in searching for code. We found that an iterative approach existed with a higher median than Google for three tasks, equal median for two tasks, and lower median for three tasks ($p = 1$ with $\chi^2$ on a $2 \times 3$ contingency table comparing best medians). These results suggest that by supporting the technique of iteration in a code search engine can provide an experience comparable to a large-scale web search engine like Google. However, it is also clear that the participants sometimes had a better experience with Google. Whether this is a consequence of the different indexes, kinds of content, participant familiarity, or ranking algorithms is uncertain.

Figure 4: Box Plot Comparing Experience Scores of Search Engines Across all Eight Tasks.
Interestingly, we found that CE and CLT complemented each other in the types of tasks they support. In Table 2, each cell displays which iterative approach had a higher experience median for each task. The table shows that CE provided a better experience for tasks that were broader (“Find 4” and “No Specific Role”) and that CLT provided a better experience for tasks that were more focused (“Find 1” and “Algorithm/Data Structure”). Such a complementary pattern, as shown in Table 2, only has a 16/6561 (0.2%) chance of occurring (given equal probability of CE, CLT, or ties occurring in each cell). The data in Table 2 suggests that using characteristics, supported by CE, of the results supports finding code when initially the search task is broad, but as the search task becomes more focused, then using the entire result to search with, supported by CLT, provides a better search experience.

Time differences to find code were examined by measuring how long it took to paste code for each of the tasks by search engine (time writing why/what explanations was not counted). If the task was not completed, the max time alloted was used. We used ANOVA to find significant differences among the groups of search engines for each task. If we did find a significant difference, we conducted a post hoc pair-wise analysis on the corresponding group using Tukey’s honest significant difference [76] to see what might be causing the difference.

The time analysis is presented in Table 3. In most cases, we did not find significant differences. However, we did find that Google was significantly faster than CLT for finding the first code snippet on task three and CLT was significantly faster than Google for finding the first code snippet on task seven. We were not able to find any significant differences in time for finding four snippets.

Finally, we looked at the number of task incompletions (no code found) by search engine as a measure of unsuccessful searches. We found that the iterative search approaches had less incomplete tasks (11) than the others (13) with CLT having the least number of incomplete tasks (4), baseline having 6, and CE and Google each with 7. However, we did not find these differences statistically significant with \( \chi^2 \). Further, we found that the "Find 4" tasks had higher incompletition rates (19/96) than the "Find 1" tasks (5/96). Further, the "Algorithm/Data Structure" tasks had a higher incompletition rate (16/96) than the "No Specific Role" tasks (8/96).

5.2 Feature Usage

We looked at how the features of the search engines were used and some participant explanations of the usefulness of the features. In Table 3: Mean Seconds Until First Paste.

### Table 2: CE and CLT Median Comparison.

| Find | No Specific Role | Algorithm/Data Structure |
|------|-----------------|--------------------------|
| Find 1 | (T3) | (T2) |
| Find 4 | (T6) | (T4) |

5.2.1 CodeExchange (CE). To understand which features of CE helped, we looked at how often features were used, when feature usages were used to find code that was copied, and what the participants had to say during our interview. The logs for CE recorded when the participants used iterative features (Recommendations, Critiques, Language Constructs, Query Parts), Keywords, History, and Advanced Search. Table 4 presents the usage frequency counts of features by task. The features that support iteration were used most often (50.6%) and then keyword text box (41.0%) second, advanced search (6.0%) third, and history (2.2%) fourth. Further, we found the iterative features were used significantly more often than keywords (\( p < 0.1 \), where \( p = 0.01 \) with \( \chi^2 \) on a 2 x 1 contingency table).

To get a better idea of the iterative features’ impact on searching we looked at how many copies happened after only keywords versus after queries that include a recommendation, critique, or a language construct. We separated counting query parts from the other iterative features in this part of the analysis because counting copies after query part usages also counts copies after a query composed only of keywords, which we are trying to separate out in this analysis. Further, we look at the copy/query and copy/iterative-feature ratios (these ratios measure how many copies happen per keyword query and how many copies happen per query created with an iterative feature). The ratios appear in parenthesis after the copy counts in the table. We found that when the iterative features were used to refine a query, they lead to higher number of copies on average (0.51) than when only keywords were used (0.28).

Our participants told us that, at a high level, CodeExchange was better for drilling down... CodeExchange helped me go in a particular direction, where CodeLikeThis did not tell me. They found language constructs useful, saying I liked clicking the import... If I found a project that seemed to do what I needed... I could just click... and my search is in the project. They felt query parts helped, saying query parts helped explore. They reported that refinement recommendations gave them ideas and helped them remember, saying initially I typed in SMTP and got suggestion for a mail validator, then I added import and it guided me to think of ideas and I typed just “draw” in here, and it recommended AWT... I was like “Oh yeah, it was AWT”.

### Table 4: CodeExchange Search Behavior.

| Task | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|------|----|----|----|----|----|----|----|----|
| CE   | 219| 294| 167| 392| 488| 263| 301| 372|
| CLT  | 185| 382| 359| 357| 373| 285| 103| 189|
| BL   | 263| 178| 174| 466| 354| 278| 158| 301|
| G    | 176| 244| 85 | 393| 410| 182| 349| 169|
| ANOVA| p = 0.8| p = 0.5| | p = 0.03| 0.04| p = 0.5| p = 0.2| p = 0.06| p = 2
We found that keyword queries were used (84.7%) about five times were visited a total of 529 times, with the frequency following a χ^2 with the same link multiple times). We found that 126 different domains participants (we do not double count a visit when one participant clicks a hyperlink and how often those domains were visited across participants.

However, we looked at how many copies happened after keywords versus after like-this queries and the copy/keyword and copy/like-this ratios. The ratios appear in parenthesis after the copy counts. We found that keywords and like-this queries lead to an equal number of copies on average when they were used (.42). This suggests that keywords and like-this queries both had an equally important impact in searching for code. Some example event sequences in the logs recorded using CLT were: keyword query, back button, keyword query, somewhat-like-this query, more-like-this query, code copied and keyword query, more-like-this query, code copied.

At a high level, participants explained like-this queries helped them get a new perspective. They said with CLT you get to see different codes and different ideas and it gave me a new perspective and it was sort of a way of exploring. They explained the like-this queries were helpful because I wasn’t sure exactly what I was looking for and that it works well for queries that are common and precise: quicksort, hash table, game loop or game examples for instance.

However, it is clear that like-this queries do not always work, one participant explained more like this is pretty much what you expect, but the other two doesn’t really follow the semantics in my mind and another saying I felt a little lost.

5.2.3 Google. For Google, we recorded normal keyword queries, advanced keyword queries (e.g., restricting search to a site with the “site” qualification), and domains visited by clicking hyperlinks. We found that keyword queries were used (84.7%) about five times as much advanced queries (15.2%) with p < 0.1 (p = 1.4 × 10^{-28}) with χ^2 on a 2 × 1 contingency table. This suggests either keywords sufficed or advanced search was less helpful or harder to use for the participants.

We looked at the number of domains that were visited by clicking a hyperlink and how often those domains were visited across participants (we do not double count a visit when one participant clicks the same link multiple times). We found that 126 different domains were visited a total of 529 times, with the frequency following a long tail distribution. github.com (130/529 visits) and stackoverflow.com (105/529 visits) were visited most often, but thereafter a sharp drop in number of visits to other domains appears. However, while individually the other 124 domains are visited much less frequently, collectively they are visited more (294/529) than github.com or stackoverflow.com. This suggests that the size and variety of Google’s index was helpful for participants in their search.

Participants told us they appreciated Google for the context and comments they would find on web pages. Some said sometimes I was like, oh man I wish I could use Google at this point, just to get some context so I can understand what I need to search in CodeExchange and I use Google to find best practices and Google had comments. However, there are times when Google did not help. Our participants told us sometimes they asked Why am I getting this? and Google is lacking in digging down... and for simple straight forward [questions] I felt CodeLikeThis was better.

Lastly, we looked at the query behavior of search with Google and present the results in Table 5. Each row presents on average how many queries were issued per user, the average number of terms in the keyword queries, and the average number of terms deleted and added when modifying a query. The results support the hypothesis that code search on the Internet, even with a large-scale web search engine, is iterative and can have a vast range of queries (1.67 to 11.3). These results suggest Google could benefit from features that explicitly support iterative code search.

### Table 5: CodeLikeThis Search Behavior.

| Feature               | Back Freq | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 |
|-----------------------|-----------|----|----|----|----|----|----|----|----|----|
| Keywords Deleted Freq | 25        | 6  | 12 | 5  | 11 | 1  | 21 | 12 | 9  |
| Keywords Added Freq   | 0         | 0  | 2  | 0  | 0  | 0  | 0  | 0  | 0  |
| Like-This Freq        | 14        | 5  | 15 | 4  | 14 | 4  | 10 | 8  | 74 |
| Like-This Deleted Freq| 1         | 0  | 1  | 2  | 0  | 3  | 0  | 9  |
| Like-This Added Freq  | 25        | 29 | 22 | 19 | 15 | 8  | 11 | 10 | 12 |

5.3 Why Code Was Searched For

Using the explanations given by our participants that stated why they chose code (one explanation for each snippet they found), we present categories of reasons for selecting code being searched for. To create these categories, three graduate students (two of the authors and another software engineering graduate student) participated in an affinity diagramming session to cluster the “why” explanations. Each graduate student took a why explanation and either assigned it a category if one had already been created by the group or created a new category to assign it to. The group was free to discuss names of categories and modify them if needed. From the 463 “why” explanations 28 categories emerged. The majority of reasons fell into five clusters; we show the top 17 clusters in Figure 5 (the other 11 clusters had only 1 to 3 explanations in them). Often, programmers searched for code because it

- helped implement a feature the participant had in mind (e.g., A user would want to save their sketch, so we need save and load ...)
- to support a design decision the programmer made about what the code should do (e.g., I want to support 3D too)
- meets the problem specifications (e.g., ... performs the job really well and hence I choose it)

### Table 6: Google Search Behavior.

| Feature               | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|-----------------------|----|----|----|----|----|----|----|----|
| Queries per User      | 6.33 | 3  | 2.83 | 3.67 | 11.5 | 3  | 5.83 | 1.67 |
| Terms per Query       | 5.92 | 5.52 | 4.23 | 5.24 | 4.87 | 5.93 | 4.8 | 4.5 |
| Terms Deleted per Query| 2.43 | 1  | 1.47 | 1.61 | 1.42 | 1.57 | 2.03 | 0.67 |
| Terms Added per Query | 2.57 | 1.05 | 1.53 | 1.88 | 1.45 | 1.79 | 1.94 | 1  |

782
Figure 5: Why Programmers Selected Code.

- serves as a starting point to solve the programming problem (e.g., … an ideal starting point for building connect four… contains the correct sequence of gameplay…)
- serves as an example or a reference for how to solve the programming problem (e.g., … demonstrates how calculators work…)

These results suggest that when the code being searched for is not completely specified (as in our tasks), that the programmer will make decisions on what to search for as they search. Often, she makes decisions related to design (features she thinks are needed or other design decisions). Making design decisions is often argued as an iterative process [9, 55], suggesting that making decisions might play a role in making search iterative. Further, it is clear that often programmers found code simply because they felt it satisfied requirements or would serve as a useful starting place to write code that would satisfy requirements for the problem described in the search task.

6 THREATS TO VALIDITY

Several possible threats to validity exist with our study. First, while we put in our best effort to make this lab study realistic, it still lacks the realism one would find in a field study with each of the search engines. As such, further studies need to be performed to examine whether our results hold in real-world environments.

Second, as all of our participants have used Google for years, and Google indexes more code and different information than our prototypes, our experiment inherently is unbalanced. However, we felt it was important to tolerate this imbalance, since Google is so ubiquitous and represents a ‘gold standard’ for how developers search. It is not surprising to us that Google performed better on a number of cases, even though CE and CLT offer the same ability to search with keywords only. What is more important, however, is that CE and CLT did outperform Google on a sufficient number of cases to show the promise of dedicated support for iteration in search.

Third, the search tasks we used by no means cover all possible types of search tasks. We intentionally focused on broader search tasks, given the goal of this paper of addressing code search when developers do not know exactly what they want and search around more exploratory. However, even within this narrower focus, we could have chosen to use other search tasks. While we attempted to ameliorate this issue by modeling our tasks on real searches in real code search engines, a longitudinal study with CE and CLT is needed to examine if their features apply beyond the eight search tasks we used.

Fourth, it is possible that participants did not seriously attempt the tasks. Two of the authors and a colleague graduate student not involved with the research each individually inspected all snippets and explanations, assessing if they represented genuine attempts. In 97.3% of the cases, unanimous agreement was that they were genuine attempts (1 result was ranked not genuine by all 3 people, 6 results by 2, and 6 results by 1; these were spread across search engines and participants). This gives us confidence that most of our results represent genuine attempts by our participants.

7 CONCLUSION

In this paper, we investigated code search on the Internet from the point of view that it often is an iterative process, because the developer does not always know in advance what she may need. Our results suggest that providing features explicitly designed to support iterative code search can improve the search experience. Compared to the baseline, we found that both CE and CLT lead to higher median experience scores. Compared to Google, we found that that CE and CLT each can, on a number of occasions, lead to an experience that is comparable to that of using Google, even as participants were much less versed in CE and CLT than Google — which they use daily.

Our results are also nuanced, as CE and CLT provide better experiences for opposing kinds of search tasks. CE provides a better experience for broader searches looking for multiple and possibly different examples that can help the programmer. CLT tended to better support tasks that were more focused, looking for examples of a particular kind. This in many ways is surprising, as CLT was designed to support diversity in results and CE to focus more on refinement of results. Further research will be necessary to determine why the difference emerged. Regardless, it is clear that both CE and CLT offer features that assist at important times when keyword searches seem to not lead to the desired, or at least less desirable, results.

Overall, we believe that our results point at the need to explore the design of novel search engines combining traditional keyword search with the distinctive features of both CE and CLT. This is not a trivial undertaking, yet, if successful, would offer developers multiple ways of ‘jumping out’ of a current search path without having to think of how to formulate a new keyword query. That is an important ability given the premise of this paper: developers do not always know exactly what they are looking for and need help formulating what they search for as they search. Our experiment has identified two particular ways of doing so, but we expect other approaches to need to be explored as well.

ACKNOWLEDGMENTS

This work was sponsored by NSF grant CCF-1321112. Thanks to Tariq Ibrahim for his help in this project.
