Code Similarity in TouchDevelop: Harnessing Clones

Marat Akhin Nikolai Tillmann Manuel Fahndrich
Jonathan de Halleux Michal Moskal

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{t-marata,nikolait,maf,jhalleux,micmo}@microsoft.com
Microsoft Research Tech Report MSR-TR-2011-103

Abstract. The number of applications available in mobile marketplaces is increasing rapidly. It's very easy to become overwhelmed by the sheer size of their codebase. We propose to use code clone analysis to help manage existing applications and develop new ones.
First, we propose an automatic application ranking scheme based on (dis)similarity. Traditionally, applications in app stores are ranked manually, by user or moderator input. We argue that automatically computed (dis)similarity information can be used to reinforce this ranking and help in dealing with possible application cloning.
Second, we consider code snippet search, a task commonly performed by application developers. We view it as a special instance of the clone detection problem which allows us to perform precise search with little to no configuration and completely agnostic of code formatting, variable renamings, etc.
We built a prototype of our approach in TouchDevelop, a novel application development environment for Windows Phone, and will use it as a testing ground for future evaluation.

1 Introduction

In the last decade mobile devices like smartphones and tablets slowly, but steadily became the main electronic devices people use in their everyday life, taking place of traditional desktops and laptops. As a result of that, mobile application development is also increasing its market share, populating so called "app stores" or "marketplaces"[1] with dozens of applications every day.

TouchDevelop[2] [21, 20] is a novel mobile application development environment that allows users to write applications – called "scripts" – on their mobile devices, without the need to use external computers for development purposes. Instead of a traditional marketplace, TouchDevelop uses a cloud-based "script bazaar" which contains user-submitted scripts and their source codes.

One of the distinguishing features of TouchDevelop is that, unlike most other development environments, it targets novice programmers who have little or no experience

[1] In the rest of the report, we will use the notion "marketplace".
[2] TouchDevelop was previously known as TouchStudio [20].

in application development. Because of that, many people who write scripts make extensive use of Copy-and-Paste Programming [23]. Typical TouchDevelop script development lifecycle may be represented as follows:

- The user starts with an idea and a set of basic script examples relevant to his idea.
- The user creates a new script by combining and modifying some of the basic scripts until the intended idea has been successfully implemented.
- The user publishes this new script to the script bazaar, and it may serve as a new basic script for other developers.

It is obvious that often a derivative script would be very similar to its base scripts due to copy-and-pasting. In essence, it means that many scripts in the bazaar would be clones of other scripts, and this presents different problems to TouchDevelop users (and, in part, to users of other traditional marketplaces). We will focus on two of the resulting problems:

First, we propose an automatic script ranking scheme based on script similarity. In most marketplaces, applications are ranked according to user or moderator input. In the presence of clones, this approach may become difficult or even impossible to perform completely manually. To cope with this problem in the TouchDevelop setting (where application source codes are fully available) script similarity information may be used to assign higher rankings to highly distinct scripts. This is based on a simple intuition that distinct scripts have little in common with other scripts in the marketplace, meaning that they were developed mostly from scratch and should have a somewhat unique functionality which should be explored by the marketplace users.

Second, we address the problem of code snippet search. When developing an application, programmers often need to find a given code fragment, to see the context in which it is used. This is even more so in TouchDevelop because of user inexperience and the example-based development model. We argue that code snippet search can be viewed as a special instance of the clone detection problem where we have a particular clone instance that we want to find given beforehand. This approach, unlike traditional search tools used in integrated development environments (IDE), allows the user to find a code snippet with no configuration (compared, for example, to regular expressions) and completely agnostic of variable names, possible statement additions/removals, etc.

We've implemented a prototype of our system in TouchDevelop – our automatic ranking technique is used to generate the list of "featured" scripts on the web [21] and in the TouchDevelop app start with version 2.0, and our code snippet search will be included in the app starting with version 2.3. We plan to do an extensive evaluation of our approach to test its effectiveness and usability and guide our future design decisions. We also consider the possibility of implementing our approach for other marketplace based environments, to assess its limitations and level of generality.

The rest of the report is organized as follows. We briefly describe the TouchDevelop environment and programming language in section 2. We outline clone detection techniques used in our approach in section 3. Automatic script ranking and code snippet search are described in sections 4 and 5 respectively. We present our evaluation plan in section 6. Related work is discussed in section 7. Finally, we summarize our results and outline possible future work in section 8.

string ::= `"Hello world!\n"` | ...
number ::= `0` | `1` | `3.1415` | ...
local ::= `x` | `y` | `z` | `foo bar` | ...
action ::= `frobnicate` | `go draw line` | ...
global ::= `phone` | `math` | `maps` | `ok button` | `chat roster` | ...
property ::= `vibrate` | `sin` | `add pushpin` | `disable` | `add` | ...
exprs ::= *expr* | *exprs* `,` *expr*
parameters ::= | `(` *exprs* `)`
op ::= `+` | `-` | `/` | `*` | $\leq$ | $\geq$ | `>` | `<` | `=` | $\neq$ | **and** | **or** | `||`
expr ::= *local* | *string* | *number* | **true** | **false**
| `-` *expr* | **not** *expr* | *expr op expr*
| *action parameters* | *expr* $\rightarrow$ *property parameters*
block ::= *stmt* | *block stmt*
locals ::= *local* | *local* `,` *locals*
stmt ::= *expr* | *locals* `:=` *expr* | **do nothing**
| **if** *expr* **then** *block* **else** *block*
| **while** *expr* **do** *block* | **foreach** *local* **in** *expr* **do** *block*
| **for** `0` $\leq$ *local* < *expr* **do** *block*
type ::= **Number** | **String** | **Song** | **Nothing** | **Phone** | ...
formals ::= *local* `:` *type* | *local* `:` *type* `,` *formals*
formals-opt ::= | *formals*
returns-opt ::= | **returns** *formals*
action-def ::= **action** *action* `(` *formals-opt* `)` *returns-opt block*

Fig. 1. Abstract syntax of TouchDevelop actions.

2 TouchDevelop

TouchDevelop has been developed with a single main principle in mind: *all programs are created on a mobile device and are consumed on a mobile device* [20]. This principle influences all main design decisions which, in turn, influence our approach.

2.1 TouchDevelop Language

TouchDevelop uses a statically typed programming language that takes its traits from imperative, object-oriented and functional paradigms. Static typing allows highly useful code completion options to be presented to the user in the code editor, and it simplifies development, because types can be inferred, thus lifting the need for most type annotations. See Figure 1 for the language grammar.

For our approach static typing means that we can analyze similarity and detect clones w.r.t. types. This increases precision and effectiveness due to more fine-grained clone clusters.

2.2 TouchDevelop Environment

For editing scripts, the TouchDevelop environment offers an easy-to-use touch-based editor that operates on statements making it impossible to make a syntactically incorrect

```
var loc : Location;
action park() {
  loc := senses→current_location;
}
action find() {
  loc→postto_wall;
}
```

Fig. 2. Example of a complete TouchDevelop script in textual form.

program at the statement level. Figure 2 shows an example of a TouchDevelop script. A textual format is used to persist scripts, and to view scripts on the web. From this example, it may seem that the user edits script code as text, but the TouchDevelop editor actually works on abstract syntax tree (AST) level at all times. For example, in the editor, copy-and-paste operations do not copy-and-paste chunks of text, but the corresponding AST nodes.

This TouchDevelop feature allows us to do clone detection at statement level, without the need to do expression level analysis. Moreover, all editor operations can be easily mapped to the AST level which makes it easy to do AST based code analysis in TouchDevelop.

2.3 TouchDevelop Script Bazaar

Script sharing is done via the script bazaar which is a cloud-based storage to which all users can publish the source codes of their scripts. (The centralized nature of the script bazaar serves more purposes than just storage; for example, every submitted script is analyzed by an automated information-flow analysis in order to let TouchDevelop handle potential private concerns that could arise when another user would later run the script [22].) There are two main types of scripts users can publish to the TouchDevelop script bazaar:

- Root scripts that are entirely new and were written from scratch
- Derivative scripts that were created by combining and modifying other already published scripts

The TouchDevelop script bazaar may be viewed as a coarse-grained version control system (VCS) where revisions are different versions of a given root script. Moreover, usually different script versions do not differ much and contain a lot of very similar (if not equal) code.

This additional information may be used to guide clone analysis and do advanced post-processing of the results. For example, we may find it sufficient to analyze only the most recent scripts in a given script branch, because they represent the latest (most advanced) versions of it. This allows us to reduce the effective code size we have to analyze, which in turn allows us to analyze bigger codebases.

3 Clone Detection

Our approach is based on clone detection, and in this section we describe clone detection techniques that we use for automatic script ranking and code snippet search. We use a tree-based approach based on characteristic vectors to represent source code features, and for clustering we use Locality Sensitive Hashing (LSH) that performs well for our purposes. All in all, our clone detection method is very similar to DECKARD [11] with several differences we describe in more detail below.

3.1 Tree Characteristic Vectors

Tree characteristic vectors capture the structure of a given tree by mapping it to some numerical vector. For example, in [11] characteristic vectors are calculated using *q-level atomic patterns* which are pattern trees of height no more than q, and a characteristic vector captures the number of occurrences of q-level patterns in a tree.

These characteristic vectors have the nice property that Euclidean or Manhattan distance between them can be used as a lower bound on edit distance between corresponding trees. Therefore, the problem of tree similarity is reduced to the problem of finding close (w.r.t. given distance metric) q-level vectors.

3.2 Locality Sensitive Hashing

LSH is a recent addition to the family of dimension reduction methods that is based on a very simple idea: if two points p, q in high-dimensional space M are close, then their "projections" $h(p), h(q)$ should also remain close to each other [7]. Projections are defined via LSH functions $F = \{h_k\}$ that should satisfy the following three properties (given a threshold R and an approximation factor c):

- if $d(p,q) \leq R$, then $Pr[h(p) = h(q)] \geq P_1$
- if $d(p,q) \geq cR$, then $Pr[h(p) = h(q)] \leq P_2$
- $P_1 > P_2$ (in order for LSH functions to be useful)

To amplify the gap between P_1 and P_2, LSH usually uses not one, but k LSH functions, and repeats hashing L times. This allows one to achieve an arbitrarily low LSH failure probability as needed by the application.

In this work we use the stable distribution LSH function family which is described as follows [5]:

$$h(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor,$$

where a is a vector with components selected independently from some stable distribution (e.g., Gaussian distribution $N(0,1)$), w is the width of hashing buckets, and b is a number from $U(0,w)$. If you use s-stable distribution corresponding to your distance metric (e.g., Cauchy distribution for l_1 norm), you can ensure that the probability of LSH failing is no more than δ by setting L to be [19]:

$$L = \left\lceil \frac{|log\ \delta|}{log(1 - P_1^k)} \right\rceil.$$

LSH can be applied to different tasks, one of its main applications is efficient nearest neighbor search in high-dimensional space, e.g. clone detection using characteristic vectors. The basic algorithm is trivial [1]:

- For every LSH function group $g_j = (h_1, \ldots, h_k)$ calculate hash value $g_j(v)$ for the query vector v
- For every $g_j(v)$ find all vectors q_r with the same hash value and calculate $d(v, q_r)$
- If $d(v, q_r) < R$ then the corresponding trees are similar and can be considered to be clones

3.3 Characteristic Vector Generation

We use 1-level patterns to generate tree characteristic vectors, akin to DECKARD [11]. The abstract syntax tree (AST) structure of the TouchDevelop language is relatively simple and has about 20 different node types corresponding to assignments, loops, if-then-else statements, method calls, etc. We map the number of occurrences of each node type to a separate component in a characteristic vector.

To increase precision, we capture type information for variables and constants in the AST by representing them as separate components in characteristic vectors. Static typing ensures that every value has a single type, and the number of types is always finite, because TouchDevelop does not allow users to create new types, and all built-in types are non-recursive. Therefore, characteristic vectors have a finite number of components.

We also distinguish methods by their number of arguments and map method calls to different vector components based on this information. The TouchDevelop language does not allow overloading methods, and this allows us to further increase analysis precision.

These design decisions were made to maximize characteristic vector efficiency by making them expressive enough while, at the same time, bounding their size. Of course, our general approach is oblivious to a particular vector generation scheme and will work for any scheme, but a good vector generation scheme allows for higher precision and recall than a bad one.

4 Automatic Application Ranking

Our first contribution is an automatic application ranking schema that is based on application similarity. We propose to assign rankings according to application distinctiveness w.r.t. some similarity measure. This is backed up by a simple intuition that highly distinct applications are different from other applications and should have some unique functionality, therefore they should be explored by the users.

4.1 Ranking Schema

To measure application distinctiveness, we use cluster size as a main component of our ranking. The less populated a cluster is, the more distinct its applications are from other

applications under consideration. This approach can be viewed as an adaptation of the approach of Das et. al. to news filtering [4].

We consider LSH buckets to be our clusters of interest, akin to subspace clustering. Every projection by LSH function can be seen as a separate random subspace we analyze, and by aggregating over all L LSH functions we can make this approximation as close to actual clusters as needed (by setting L to high enough value) w.r.t. LSH error probability.

The algorithm proceeds as follows, given a LSH database for analyzed applications:

- For each projection from L we take top T least populated clusters and assign rankings to the corresponding applications as $r = \frac{W}{|C_t|}$ where W is a ranking multiplier that captures additional information about application distinctiveness (e.g., code size) and $|C_t|$ is the size of t-th cluster
- After that we summarize application rankings for all projections, getting the total application ranking which is used for the actual application ordering

By using this schema, we give higher ranking to applications which do not have many neighbours in a single projection and which maintain this property across many different projections.

We could use another ranking schema, for example, average distance from all cluster points, which also captures application distinctiveness. But our approach allows for both horizontal (addition of new computational nodes) and vertical (addition of computational resources to a single node) scalability and is expressive enough to capture application distinctiveness. Therefore, it is suitable for our task, and we anticipate it to prove its effectiveness in the future evaluation.

4.2 Post-processing

After computing application distinctiveness rankings, we post-process the results to improve their relevance. The main problem is as follows:

> If we have a highly distinct cluster with two very similar applications, both applications will get a high ranking

While this is completely correct w.r.t. objective measure, subjectively it is strange for the end-user to see two or more very similar applications that are recommended in the list of "featured" applications to try out.

That is why we filter out these clones using additional knowledge about applications (in our setting, information from the TouchDevelop script bazaar). We use the following strategy:

- If two scripts have a parent-child relation, the parent script is removed from the results
- If two scripts are very similar (distance between them is less than some threshold), one of them is removed from the results

These simple rules practically guarantee that ranking results are free from duplications w.r.t. our similarity measure. If two applications are semantically similar, but differ

syntactically, our approach cannot do anything to filter one of them from the results, because it cannot test applications for semantic similarity (which is, in general, undecidable).

4.3 Combining Rankings

Our similarity based application ranking can be used in conjunction with other rankings, e.g. user rankings, download counts, etc. They can be combined using simple linear ranking models or via machine-learned ranking (MLR, [9, 15]).

Combining rankings can help to further increase ranking quality from the end-user perspective, because users are generally interested not only in application uniqueness w.r.t. similarity, but also in its quality and interestingness. The natural way of doing this is combining several rankings that capture different aspects of application identity together. In this work, due to time limitations, we have not implemented any combination of rankings, and it remains one of the possible future works.

4.4 Limitations

Of course, our approach is not without any remaining problems. For example, because we analyze only syntactic similarity, it is quite easy to ”fool” the system by scattering some “dead code” across an otherwise cloned application. In this case two identical applications (with and without dead code) will appear as very distinct ones and receive a high ranking.

The only way to deal with this is to use not syntactic, but semantic similarity to rank applications. But this problem is undecidable in general, and there are no known high-performance algorithms that can solve it efficiently for any interesting particular cases.

5 Code Snippet Search

Our second contribution is an approach to code snippet search. We consider the similar code search problem as an instance of the clone detection problem: given a code fragment which the user wants to find (user query), we find all its clones, i.e. similar code snippets, and present them to the user.

If the user query consists of a single AST subtree, this problem is reduced to a very simple problem of finding all similar subtrees(w.r.t. characteristic vectors) using LSH. But if the user query corresponds to an AST forest, we can’t use this naive approach.

We can use the technique from [11] to calculate characteristic vectors not only for AST nodes, but also for AST forests up to some limited width. While this approach is reasonable for clone detection itself (it just limits clone detection ability to clones up to a particular size), in our settings we don’t want to limit the end-user to be able to search only for fragments of a limited length. Moreover, we want to allow the user to search for similar fragments with arbitrary gaps in them which is not supported by the DECKARD approach.

5.1 Algorithm

We propose a different approach to deal with this problem that is inspired by sequence alignment algorithms from bioinformatics [17]. It proceeds as follows:

- For every node in the AST, in addition to its characteristic vector, we store its breadth-first search (BFS) index and BFS index of its parent node.
- We find similar nodes for every query node independently, getting a list of applications and BFS indexes of possible matchings within these applications.
- For every application from this list we build a sequence of matching nodes (using BFS indexes).
- These sequences are matched to the original query, all matching sequences are reported as search results.

Let us explain the sequence matching algorithm in more detail. We define the following properties for an AST node n:

- $v(n)$ – characteristic vector of n
- $index(n)$ – BFS index of n
- $parent(n)$ – BFS index of n's parent

Both the user query and matching sequences can be considered as sequences of AST nodes $p = p_1, \ldots, p_i, \ldots, p_m$.

First, we try to align the user query p and a given sequence a of matching nodes, i.e. find all subsequences s of a that satisfy the following properties:

- $d(v(s_i), v(p_i)) \leq \sigma$
- $id(s_i) \neq id(s_{i+1})$

A naive implementation of alignment algorithm is $O(m^2)$, by using dynamic programming we can achieve subquadratic performance.

To determine whether a candidate subsequence s matches the user query p we use the following rules that must hold for all $s_i, s_{i+1}, p_i, p_{i+1}$:

- $parent(p_i) = parent(p_{i+1}) \rightarrow parent(s_i) = parent(s_{i+1})$
- $(id(s_{i+1}) - id(s_i)) - (id(p_{i+1}) - id(p_i)) \leq \theta$

The first rule ensures that we do not match code fragments with mismatching parent-child hierarchies, the second one prevents gaps in matchings from becoming too large.

All matching sequences (and their corresponding scripts) are presented to the user as the search result. As in section 4.2, we post-process the search results and filter out parent scripts if their children are also present in the results.

5.2 Example

To better illustrate our approach, let us consider a very simple example. The user wants to find the following code snippet:

```
1 board → update on wall    a/7/2
2 board → evolve            a/8/2
3 phone → vibrate (0.1)     c/9/2
```

We write the corresponding AST nodes as $v(n)/id(n)/parent(n)$. In our example, `update on wall` and `evolve` method calls are the same w.r.t. similarity measure, therefore, they are mapped to the same characteristic vector.

After querying the LSH database, we get the following results:

```
a matches FOO at 13/5  c matches FOO at 15/6
  matches FOO at 14/5    matches BAR at 60/3
  matches BAR at 45/3
  matches BAR at 46/3
  matches BAR at 58/3
  matches BAR at 59/3
```

We build matching sequences for `FOO` and `BAR` scripts by sorting the LSH data on $id(n)$:

```
FOO: a/13/5  BAR: a/48/3
     a/14/5       a/49/3
     c/15/6       a/58/3
                  a/59/3
                  c/60/3
```

When trying to align the query sequence `aac` to matching subsequences, `FOO` has a single possible alignment, and `BAR` has three possible alignments:

```
FOO: a/13/5 -> a/7/2  BAR: a/48/3 -> a/7/2
     a/14/5 -> a/8/2       a/49/3 -> a/8/2
     c/15/6 -> c/8/2       c/60/3 -> c/9/2
BAR: a/49/3 -> a/7/2  BAR: a/58/3 -> a/7/2
     a/58/3 -> a/8/2       a/59/3 -> a/8/2
     c/60/3 -> c/9/2       c/60/3 -> c/9/2
```

Of these subsequences, only the last one satisfies both matching rules (the first subsequence violates the parent-child hierarchy rule, and the second and the third ones violate the gap rule) and is presented to the user in the search results.

As seen from this example, our approach is limited by construction in what kind of similar code it can find. It is highly dependent on what definition of similarity is used – for example, in the current work we define two method calls to be similar if the types of callee objects and method signatures match. If we have several different methods with this property, then they are considered similar by our approach.

This notion of similarity is sufficient in our setting (similar script search for possible code reuse). If, for example, someone needs to find exact matches, then he can redefine the similarity measure (by changing characteristic vector generation scheme and/or sequence matching rules) and achieve his goal using our approach. In a sense, our approach can be seen as parametrized by the similarity measure which gives it a nice flexibility in possible applications.

6 Evaluation Plan

We plan to conduct several user studies, both explicitly and implicitly, to evaluate the efficiency of our approach. We have built a prototype system and integrated it as a part of TouchDevelop, so that the user studies could be done in the most user-friendly way. We briefly describe our evaluation plan below.

6.1 Ranking Efficiency Study

We plan to collect two statistics to evaluate our automatic ranking efficiency. First, we will directly survey TouchDevelop users to measure ranking quality by asking whether they agree or not with this ranking. This will show us if our ranking approach is good enough from an average end-user's point of view.

Second, we will collect information about how often users actually install scripts from the top of our ranking, as opposed to other script lists which the user can browse, e.g. "new" scripts, "top" scripts that are most popular, etc. We argue that similarity can be used to distinguish interesting scripts, and if this claim is true, then there should be a large number of installations from our ranked script list.

6.2 Similarity Search Study

To assess efficiency of our similar script search, we will use almost the same approach. We will survey the users and ask them to measure search relevance, i.e. if the search did find some useful code fragments or not.

We will also analyze how often the users actually copy the code from search results. The main intended use for similar script search is to allow users to find code snippets that they can reuse. If we achieve this goal, then there should be a large number of "copy-and-paste" operations from the search results.

7 Related Work

In this section we briefly discuss related work in the fields of application ranking and similarity analysis.

7.1 Automatic Application Ranking

Ranking is a way of comparing two items together w.r.t. some property. In this work we talk about marketplace application ranking that compares applications by their "interestingness level" to the end-user. To our knowledge, there are no works that propose to do this ranking automatically, by using code similarity analysis information, – traditionally this is done by manual voting or statistical analysis.

As mentioned before, our work is similar to the approach of Das et. al. which uses click statistics for news recommendation [4]. In our case, we use similar methods (LSH, MLR) but apply them to a completely different domain (application ranking based on similarity). We also left some aspects to future work (for example, MLR) and we do not yet have any experimental evaluation. That is why we cannot make any final statements about the efficiency of our approach until we do that.

7.2 Similarity Analysis

Similarity analysis is usually considered in the context of clone detection when you find software clones and eliminate them (e.g., refactor them away) to improve software maintainability and quality. Many definitions of similarity have been proposed in the literature, most of them can be summarized as follows [18]:

- Type I – identical code fragments modulo formatting
- Type II – type I with possible variations with variable naming, types, etc.
- Type III – type II with further modifications (e.g., insertion or removal of statements)
- Type IV – semantically equivalent, but syntactically different code fragments

The ability to detect different clone types is highly dependent on what algorithms are used to do it. In this work we focus on Type III similarity, and text- [2] and token-based [14, 12, 10] approaches are not suitable for this kind of task.

The tree characteristic vector approach [11] that we use as a basis in this work corresponds both to statistical- and tree-based approaches. Most tree-based approaches support detection of Type III clones, because they work on an AST level that can represent addition/deletion of statements in a simple and natural way. We extended this approach to support gaps of arbitrary size by sacrificing some performance which, in our setting, is mitigated by the fact that we use it for similar code search per user query and not for full codebase clone detection.

There are several other groups of approaches that can work with Type III clones. Program dependence graph (PDG) based approaches [13, 6] try to solve the subgraph isomorphism problem which is known to be NP-complete [3]. That is why all PDG based clone detection methods use some kind of approximation algorithms to make the problem feasible. While these approaches are, in theory, more precise and allow for better clone detection than AST based methods, in practice, low performance and scalability make it impossible to use them on large codebases.

Machine learning based approaches [16, 8] also support Type III clones detection and work by applying classic machine learning algorithms like Latent Semantic Indexing (LSI) or Independent Component Analysis (ICA) to source code. While they can be used for extracting and detecting similarity in the software, they cannot be easily adapted to similar code search.

8 Conclusion

Currently mobile application development is growing rapidly, and so is the number of applications. In this environment many tasks become difficult or even impossible to perform manually and require automation.

We present an approach to automate two of these marketplace-oriented tasks. First, we propose an automatic application ranking scheme based on (dis)similarity that allows to identify interesting scripts without any need for user input; this helps to manage the growing marketplace. Second, we present a similar code search technique based on an adaptation of well-known clone detection methods to code search; this gives the developer a tool to find relevant code examples.

We've implemented a prototype of our system in TouchDevelop and use its "script bazaar" as a testing ground for our approach. We plan to evaluate the efficiency and usability of our approach in our future work.

Acknowledgment

We would like to thank all researchers and developers at Microsoft Research for inspiring discussions and wholehearted support.

References

1. Alexandr Andoni and Piotr Indyk. Near-optimal Hashing Algorithms for Approximate Nearest Neighbour in High Dimensions. *Commun. ACM*, 51(1):117–122, 2008.
2. B. S. Baker. A Program for Identifying Duplicated Code. In *24th Symposium on Computing Science and Statistics*, pages 49–57, 1992.
3. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.
4. Abhinandan Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google News Personalization: Scalable Online Collaborative Filtering. In *Proceedings of the 16th International World Wide Web Conference*, WWW2007, pages 271–280, 2007.
5. Mayur Datar and Piotr Indyk. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the Symposium on Computational Geometry*, pages 253–262, 2004.
6. F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J. F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 603–612. ACM, 2008.
7. Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
8. Scott Grant. Vector Space Analysis of Software Clones. In *Proceedings of the 17th International Workshop on Program Comprehension*, pages 233–237, 2009.
9. A. Gulin, P. Karpovich, D. Raskovalov, and I. Segalovich. Yandex at ROMIP'2009: Optimization of Ranking Algorithms by Machine Learning Methods. In *Proceedings of ROMIP'2009*, ROMIP '09, pages 163–168, 2009.
10. Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Index-based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.
11. Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.
12. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

13. Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, WCRE '01, page 301, Washington, DC, USA, 2001. IEEE Computer Society.
14. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a Tool for Finding Copy-Paste and Related Bugs in Operating System Code. In *6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 20–20, Berkeley, 2004. USENIX Association.
15. Tie-Yan Liu. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.
16. Andrian Marcus and Jonathan I. Maletic. Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, page 107, Washington, DC, USA, 2001. IEEE Computer Society.
17. David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001.
18. C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University, 2007. TR 2007-541.
19. Malcolm Slaney and Michael Casey. Locality-Sensitive Hashing for Finding Nearest Neighbors. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008.
20. Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop – Programming Cloud-Connected Mobile Devices via Touchscreen. *Microsoft Technical Report MSR-TR-2011-49*, 2011.
21. TouchDevelop. `http://research.microsoft.com/TouchDevelop`.
22. Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan de Halleux, and Michal Moskal. Transparent Privacy Control via Static Information Flow Analysis. *Microsoft Technical Report MSR-TR-2011-93*, 2011.
23. G. Yarmish and D. Kopec. Revisiting Novice Programmer Errors. *ACM SIGCSE Bulletin*, 39(2):131–137, 2007.