Detecting and Measuring Similarity in Code Clones

Randy Smith and Susan Horwitz
Department of Computer Sciences, University of Wisconsin–Madison
{smithr,horwitz}@cs.wisc.edu

Abstract

Most previous work on code-clone detection has focused on finding identical clones, or clones that are identical up to identifiers and literal values. However, it is often important to find similar clones, too. One challenge is that the definition of similarity depends on the context in which clones are being found. Therefore, we propose new techniques for finding similar code blocks and for quantifying their similarity. Our techniques can be used to find clone clusters, sets of code blocks all within a user-supplied similarity threshold of each other. Also, given one code block, we can find all similar blocks and present them rank-ordered by similarity. Our techniques have been used in a clone-detection tool for C programs. The ideas could also be incorporated in many existing clone-detection tools to provide more flexibility in their definitions of similar clones.

1 Introduction

Identifying code clones serves many purposes, including studying code evolution, performing plagiarism detection, enabling refactoring such as procedure extraction, and performing defect tracking and repair.

Most previous work on code-clone detection has focused on finding *identical* clones, or clones that could be made identical via consistent transformations of identifiers and literals. However, code segments that are similar but not identical occur often in practice, and finding such non-identical clones can be as important as finding identical code segments. For example, while automated code compaction may require finding identical clones, studies of the evolution of a codebase over time require finding clones that vary in their similarity.

One of the central issues with finding non-identical clones is assessing when two pieces of code are close enough to be considered "similar" [10, 24]. Because this is likely to depend on the context in which the clone-detection tool is used, we believe that such tools should provide a quantitative measure of clone similarity, leaving the ultimate decision of classification to the user of the tool.

In this paper, we advocate the use of explicit similarity-based clone detection, and we present techniques that can be tuned to find clones of varying degrees of similarity. Our techniques identify clones at the *block* level, based on *fingerprints* computed at the statement level. At the statement level, different fingerprinting algorithms can be used so that, at one extreme, only identical statements have the same fingerprint, while at the other extreme, the same fingerprint may be given to many minimally similar statements. At the block level, sequences of statement fingerprints are grouped into syntactically-valid hierarchical blocks that reflect the structure of the source code. We can then compute the *similarity score* and *similarity distance* (defined in Section 3) for a pair of blocks as a function of the number of statements in one block whose fingerprints match those in another block.

Our techniques are largely language-independent, requiring only a language lexer. We have implemented a clone-detection tool for C programs based on these ideas that can be tuned by the user to find clones with varying degrees of similarity. This explicit notion of similarity enables interesting queries to be performed. First, our tool can be used to find *clone clusters*: sets of clones such that the similarity distance between any pair of clones in a cluster is less than a user-specified maximum. Second, for a code segment S, the tool can be used to find all clones within a given similarity distance of S and will present them rank-ordered by decreasing similarity.

To illustrate, Figure 1 provides an example of four similar code segments taken from the GNU *DAP* program. All four code fragments involve finding the next token in the input and include code to print an error message for tokens that are too long. However, the kinds of tokens found are different, there are two different error messages, and just two of the four seg-

```
for ( ; alphanum(c); c = dgetc(dotc, dapc, out)) {
  if (t < TOKLEN) token[t++] = c;
  else {
    token[t] = '\0';
    fprintf(stderr, "dappp:%s:%d: token too long: %s\n", dotname, lineno, token);
    exit(1);
  }
}
unget1c(c, dotc, (out ? dapc : NULL));
```

Segment 1

```
for ( ; num(c); c = dgetc(dotc, dapc, out)) {
  if (t < TOKLEN) token[t++] = c;
  else {
    token[t] = '\0';
    fprintf(stderr, "dappp:%s:%d: token too long: %s\n", dotname, lineno, token);
    exit(1);
  }
}
unget1c(c, dotc, (out ? dapc : NULL));
```

Segment 2

```
for (t = 0; c == '.' || ('0' <= c && c <= '9'); c = sbsgetc(sbsfile)) {
  if (t < TOKENLEN) token[t++] = c;
  else {
    token[t] = '\0';
    fprintf(stderr, "sbstrans: before %d: token too long: %s\n", sbslineno, token);
    exit(1);
  }
}
```

Segment 3

```
for (t = 0; ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z') || ('0' <= c && c <= '9') ||
        c == '_' || c == '.'; c = sbsgetc(sbsfile))
{
  if (t < TOKENLEN) token[t++] = c;
  else {
    token[t] = '\0';
    fprintf(stderr, "sbstrans: before %d: token too long: %s\n", sbslineno, token);
    exit(1);
  }
}
```

Segment 4

Figure 1. A clone cluster from the GNU *DAP* project.

ments end with a call to `ungetlc`. Our tool can find these four fragments as a clone cluster, or, given one fragment, can identify the other three as being similar.

Further details about fingerprinting and how fingerprints are used by our clone-detection tool to compute similarity scores and similarity distances are given in Sections 2 and 3. Section 4 describes how to use these similarity measures to find clone clusters, and to rank-order clones according to their similarity to a given block of code. Section 5 discusses related work, and Section 6 concludes.

In summary, this paper makes the following contributions:

1. we advocate the use of explicit similarity-based mechanisms for clone detection;
2. we present a clone-detection technique that can be tuned to find blocks of code with varying degrees of similarity;
3. we introduce the similarity score and distance as a means for gauging similarity, and present clustering and rank-ordering applications built from them;
4. we propose a fingerprinting scheme for finding identical or similar statements, providing another source of tunability in our approach to clone detection.

2 Similarity-Preserving Fingerprints

In this section we discuss how to use fingerprinting to identify statements that are similar but not necessarily identical. One approach would be to use a measure like edit distance: two statements would be considered similar if they were within some threshold edit distance of each other. This approach would have the advantage of being tunable, but the strong disadvantage of being prohibitively expensive when a large number of statements must be compared pairwise. Furthermore, edit distance alone may not best capture statement similarity. For example, consider the two pairs of if-conditions shown below (as sequences of tokens):

Pair 1: A "function-call" condition and a "less-than" condition

```
if ( id ( id ) )
if ( id < id )
```

Pair 2: Two "bitwise-and" conditions

```
if ( ( id + id ) & ( id * id ) )
if ( id & id )
```

The first pair has a closer edit distance than the second pair: to transform the "function-call" condition to the "less-than" condition requires just two deletions (of the two parentheses) and one insertion (of the `<`), while for the second pair the transformation requires eight deletions. However, it might make more sense to consider the two "bitwise-and" conditions to be more similar than the first pair, based on the fact that they share the same relatively rare operator. This observation is supported by information theory: for a probability distribution $Pr(T)$ over a domain T, the information content of an element $t \in T$ increases as its probability of occurrence decreases; *i.e.*, the more infrequent a domain element is, the more important or characteristic it becomes.

With these observations in mind, we use a fingerprinting technique that is both more efficient than edit distance and also respects the idea that a sequence of tokens may be best represented by its *characteristic components*: those that capture its essence and distinguish it from other, dissimilar sequences. Given this representation, two sequences are deemed *similar* if they share the same characteristic components.

The fingerprinting algorithm that we use is an adaptation of one that has been used successfully to identify similar English sentences in text documents [5,22]. For each statement S, the algorithm computes all n-grams, the sequences of tokens of length n that occur in S, then produces the fingerprint for S by concatenating the k least-frequent n-grams.[1] By using the *least frequent* n-grams, we take into account the importance of the presence of a rare token; because each n-gram represents n consecutive tokens, we take into account the fact that similar token ordering should affect the perceived similarity of two statements; and finally, by selecting only k n-grams to represent a statement, we allow similar but non-identical statements to have the same fingerprint.

To illustrate our fingerprinting algorithm, Figure 2(a) shows the 4-grams of the statement `if (signum<0 || sigstr(signum, signame)!=0) break;` represented as the sequences of tokens `if ( id < intlit || id ( id , id ) != intlit ) break ;` with their frequency rankings. The rankings are taken from the frequency distribution of all 4-grams in the GNU *bash* shell source code.[2] Figure 2(b) shows the three least-frequent 4-grams for the example statement. Each of those 4-grams is shown with its frequency and

[1] We mean the elements that have the lowest probability of occurrence over all sequences, not the elements that occur least frequently in S itself.

[2] For moderate to large codebases, we find that n-gram frequency distributions are consistent: n-grams that are infrequent in one codebase are typically infrequent in other codebases.

its representation as a sequence of four token numbers (*e.g.*, 31 is the token number for the `id` token). Figure 2(c) gives the statement's fingerprint: the concatenation of the representations of the three 4-grams.

Choices for k and n can be used to tune the algorithm according to the desired level of similarity: the larger the values, the more likely that only identical statements will have the same fingerprint (because the fingerprint will encode the exact sequence of tokens). However, larger values for k and n also increase the cost of computing a fingerprint. Therefore, if the goal is to have only identical statements map to the same fingerprint, we suggest more traditional techniques such as Rabin fingerprints [11] or MD5 hashing [19].

To evaluate the effectiveness of statement fingerprinting, we tokenized and fingerprinted the entire bash shell codebase using the following three-step process to compute a fingerprint for each statement S in the codebase:

1. For each n-gram in S, look up $frequency(S)$ in a pre-computed database of n-gram frequencies.

2. Select the k least frequently occurring n-grams in S.

3. Concatenate the k n-grams in occurrence order in S.

We found k=3 and n=4 to be effective for finding similar statements: not too many statements have the same fingerprint, and those that do are likely to be considered to be similar by a human programmer. For example, we show below two sets of non-identical statements (expressed as sequences of tokens) where for each set all statements have the same fingerprint.

1. Two for-loops that differ only in the initialization.

```
for (id=intlit; id[id] && id(id[id]); id++);

for (id=id; id[id] && id(id[id]); id++);
```

2. Three assignment statements with function calls that differ only in the number of parameters.

```
id = id ( id->id->id , id ) ;

id = id ( id->id->id , id , id ) ;

id = id ( id->id->id , id , id , id ) ;
```

4-gram	Freq	4-gram	Freq
`if ( id <`	8614	`( id < intlit`	6988
`id < 0 ||`	984	`< intlit || id`	1008
`intlit || id (`	420	`|| id ( id`	3050
`id ( sid ,`	117967	`( id , id`	77927
`id , id )`	64951	`, id ) !=`	532
`id ) != intlit`	1111	`) != intlit )`	1722
`!= intlit ) break`	102	`inlit ) break ;`	734

(a) all 4-grams and their frequencies

4-gram	Freq	Seq of Token Numbers
`intlit || id (`	420	45:10:31:08
`, id ) !=`	532	02:31:09:27
`!= intlit ) break`	102	27:45:09:16

(b) the 3 least frequent 4-grams in order of occurrence

`45:10:31:08:02:31:09:27:27:45:09:16`

(c) the final fingerprint

Figure 2. Fingerprint construction for the tokenized statement
`if ( id < intlit || id ( id , id ) != intlit ) break ;`

3 Block-level Similarity

Our tool uses the fingerprints computed for each statement to compute similarity scores and distances for pairs of *blocks* of code. This means that we identify code clones at the block level. One issue with this approach is that duplicated code that constitutes a small part of otherwise disparate blocks may not be readily identified. There are several advantages, though. First, blocks provide natural and reasonable boundaries intrinsic to the source language for comparing code. Other such boundaries tend to be either too coarse (*e.g.*, function bodies) or too fine (*e.g.*, statements). Second, blocks reflect code structure without the need to invoke full parsing. Finally, blocks provide a well-defined structure over which we can quantify similarity and provide tunable measures.

For C programs, a block is a sequence of statements delimited by a matching pair of curly braces. If one block contains another, we consider both as distinct objects. To avoid the fruitless task of comparing a block with its own components, we keep track of each block's starting and ending position in the source code.

The *similarity score* for a pair of blocks is an ordered pair containing the number of fingerprints common to both blocks divided by the size of each block, respectively; i.e., for two blocks of code S_1 and S_2 it is defined as follows:

$$sim(S_1, S_2) = \langle \frac{|S_1 \cap S_2|}{|S_1|}, \frac{|S_1 \cap S_2|}{|S_2|} \rangle$$

The first term in the ordered pair is the fraction of fingerprints in the first block that is common to both blocks, and the second term is the fraction of fingerprints in the second block common to both blocks. We argue that similarity as defined here is naturally binary-valued and presents a better picture of the relationship between code blocks than a single number does. Specifically, the binary-valued structure of the similarity score can be interpreted as assessing the relative containment of one block inside another. For example, a score of $\langle 1.0, .333 \rangle$ indicates that the first block is wholly contained within the second, and that roughly a third of the second is contained in the first.

Nevertheless, binary-valued similarity scores are not appropriate when a total ordering of scores is needed. This is the case, for example, when the goal is to find all blocks that are similar to a given block, and to present them rank-ordered by similarity. Simply retaining one part of the similarity score is not a good solution when one block is much larger than the other. For example, retaining only the first component of the score $\langle 1.0, .333 \rangle$ fails to distinguish it from the score $\langle 1.0, 1.0 \rangle$. Therefore, we transform a similarity score to a single-valued *similarity distance* when necessary. The similarity distance for a similarity score $\langle s_1, s_2 \rangle$ is defined as follows:

$$sim\_dist(\langle s_1, s_2 \rangle) = 1 - \sqrt{\frac{(s_1)^2 + (s_2)^2}{2}}$$

Geometrically, s_1 and s_2 are treated as the two sides of a right triangle with *sim_dist* related to the length of the hypotenuse, normalized to a value between 0 and 1. Other measures such as a simple arithmetic mean may be used, but in practice we prefer the proposed definition, since it results in a lower (better) similarity distance for similarity scores with at least one large component than the arithmetic mean does.

One of the consequences of our definitions of similarity score and distance is that statement order is irrelevant to clone identification. This has both positive and negative consequences: clones whose statements have been reordered are readily identified at the possible cost of introducing false matches due to blocks whose statements match but logically are not clones. One could use order-preserving techniques [20], but in practice, we have found our lack of ordering to have negligible effect in clone identification.

4 Code Clustering and Rank-Ordering

We briefly describe how our tool uses similarity scores and distances to perform code clustering and rank ordering.

4.0.1 Clone Clusters

It is often useful to find clusters of clones that are mutually similar. Formally, a set of blocks forms a cluster if and only if every pair of blocks in the set is within some user-supplied similarity threshold. Thus clustering requires an $O(n^2)$ step to compute the similarity score for each pair of blocks. Fortunately, we can significantly reduce the runtime of this step if we avoid computing similarity scores for blocks with no fingerprints in common. This is done by maintaining an inverted index keyed on fingerprint values that maps them to the blocks in which they are found. For a block B with fingerprints f_B, we compute similarity scores only for those blocks that also contain some fingerprint in f_B.

Once the set of similarity scores is computed, we convert the results to a graph and find all maximal cliques. Blocks are graph nodes, and an edge is placed between two nodes if the similarity distance for the corresponding two blocks is less than the threshold. We then find the maximal cliques using the algorithm in [23], which produces sets of blocks in which each block in a set is within the threshold of every other block in the set. Figure 1 shows a cluster of four similar clones found with a threshold of 0.5.

4.0.2 Rank Ordering

One common use of clone-detection tools is to find all clones of a given code segment S. With similarity scores we can generalize this operation to produce a rank-ordered sequence of *similar* clones ordered by decreasing similarity. Operationally, the procedure is similar to that for clone clusters, except that an all-pairs comparison is replaced with an $O(n)$ comparison of S to all other blocks. As before, we use an inverted index to significantly reduce the number of comparisons. After the similarity scores are computed, we rank-order the blocks according to decreasing similarity (*i.e.*, increasing similarity distance). Figure 3 gives a partial example: Figure 3(a) contains a code segment S, and Figures 3(b), (c), and (d) show successively less-similar clones with their similarity scores and similarity distance values computed with regard to segment S in Figure 3(a).

5 Related Work

Many techniques have been proposed for performing code clone detection, ranging from lightweight line- and token-based syntactic techniques [2, 7–9, 17] to heavier-weight approaches that emphasize semantics [3, 12, 16], to metric-based techniques that indirectly measure similarity [13, 18]. Due to limited space, we refer to existing summaries [4, 14] and focus on related work as it pertains explicitly to similarity.

```
if (!recursive(_NL_CURRENT (LC_TIME, D_FMT))) {
  if (*decided == loc) return NULL;
  else rp = rp_backup;
} else {
  if (*decided == not &&
    strcmp(_NL_CURRENT(LC_TIME, D_FMT), HERE_D_FMT))
    *decided = loc;
  want_xday = 1;
  break;
}
*decided = raw;
```

(a)

```
if (!recursive(_NL_CURRENT(LC_TIME, T_FMT_AMPM))) {
  if (*decided == loc) return NULL;
  else rp = rp_backup;
} else {
  if (*decided == not &&
      strcmp (_NL_CURRENT(LC_TIME, T_FMT_AMPM),
              HERE_T_FMT_AMPM))
    *decided = loc;
  break;
}
*decided = raw;
```

(b) sim((a), ·) = $\langle 0.83, 0.71 \rangle$, sim_dist=0.22

```
if (!recursive(_NL_CURRENT (LC_TIME, T_FMT))) {
  if (*decided == loc) return NULL;
  else rp = rp_backup;
} else {
  if (strcmp(_NL_CURRENT(LC_TIME, T_FMT),
             HERE_T_FMT))
    *decided = loc;
  break;
}
*decided = raw;
```

(c) sim((a), ·) = $\langle 0.67, 0.57 \rangle$, sim_dist=0.38

```
const char *fmt = _NL_CURRENT(LC_TIME, ERA_T_FMT);

if (*fmt == '\0') fmt = _NL_CURRENT(LC_TIME, T_FMT);
if (!recursive(fmt)) {
  if (*decided == loc) return NULL;
  else rp = rp_backup;
} else {
  if (strcmp(fmt, HERE_T_FMT)) *decided = loc;
  break;
}
*decided = raw;
```

(d) sim((a), ·) = $\langle 0.50, 0.33 \rangle$, sim_dist=0.58

Figure 3. (a) shows a user-supplied code segment. (b)-(d) show successively less-similar clones.

To the best of our knowledge, similarity-preserving fingerprints and similarity scores were first used for detecting and quantifying similarity in text documents [5, 22]. That work applied similarity fingerprints to individual sentences and calculated similarity scores between documents represented as sequences of fingerprints. At a high level, our work can be thought of as an application of those ideas to source code.

Early syntactic techniques such as Dup [2] defined parameterized clones, allowing for variation between code sequences as long as a consistent substitution of identifiers existed. Token-based techniques operate on lexemes and are thus resilient to differences in white space. However, these techniques do not quantify similarity as we do.

Semantics-based techniques use representations such as abstract syntax trees [3] and program dependence graphs (PDGs) [12,16] to find semantically identical clones whose original source code may contain extraneous or reordered statements. These techniques cannot quantify similarity as we can.

Cordy *et al.* [6] and Roy and Cordy [20] propose a technique for finding "near-miss"' clones, with some aspects that are similar to our own. As with us, they employ a token-based system and use lightweight mechanisms for ensuring syntactic validity of potential clones. In addition, their methods for measuring similarity between clones resembles our own similarity score, although they do not seem to utilize it to the extent that we do or to make it a user-controlled parameter. Unlike them, we do not enforce statement order when measuring similarity between blocks of code. Finally, our use of similarity fingerprints for statements is distinct.

Li *et al.* [17] have proposed CP-Miner for identifying copy-paste bugs in large systems. In their approach, code sequences are transformed so that common subsequences can be identified using data mining techniques. Like us they fingerprint statements, although their fingerprints do not preserve similarity. As with other techniques, they do not quantify the degree of similarity that exists between potential clones.

Finally, techniques for detecting plagiarism deal with code sequences that are by their nature similar but not typically identical. The Moss system [1,21], for example, uses a winnowing algorithm to select fragments of source code to be fingerprinted and then calculates a similarity percentage based on the set of common fingerprints. Because its goals are different than ours, Moss operates at a coarser level of granularity than we do and cannot produce the same level of detail that we can. On the other hand, their storage requirements are smaller than ours.

In summary, even for those techniques that can identify similar clones, there is typically no mechanism for quantifying the degree of similarity: code segments are similar or they aren't. In contrast, one of our main goals is to identify non-identical clones and quantify the amount of similarity present.

6 Conclusion

We posit that identifying and quantifying similarity is important for many applications of clone detection and that similarity should be intrinsic to the definition of clones. We have presented a framework and implementation for similarity detection that operates at two levels. At the lower level, we propose a tunable statement fingerprinting scheme that tends to preserve similarity across the fingerprinting function so that similar statements have the same fingerprint. At the higher level, we present a lightweight mechanism operating on language blocks over which we can quantify the amount of similarity. We illustrate the utility of this approach by describing two applications – code clustering and rank-ordering – that our approach enables. There is considerable future work to be done, but we are optimistic that similarity-based approaches such as ours can provide additional power to clone detection.

Acknowledgements

This work was supported by NSF grant number CCF-0701957.

References

[1] A. Aiken. Moss: A system for detecting software plagiarism. http://www.cs.stanford.edu/∼aiken/moss/.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conf. on Reverse Engineering*, 1995.

[3] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.

[5] D. M. Campbell, W. R. Chen, and R. D. Smith. Copy detection systems for digital documents. In *Advances in Digital Lib.*, 2000.

[6] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–12. IBM Press, 2004.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99*.

[8] J. Johnson. Identifying redundancy in source code using fingerprints. In *Proc. of the IBM Centre for Advanced Studies Conferences*, 1993.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, July 2002.

[10] C. Kapser, P. Anderson, M. W. Godfrey, R. Koschke, M. Rieger, F. V. Rysselberghe, and P. Weißgerber. Subjectivity in clone judgment: Can we ever agree? In Koschke et al. [15].

[11] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Jnl of Research and Development*, 31(2):249–260, 1987.

[12] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Symp. on Static Analysis*, pages 40–56, July 2001.

[13] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1–2):77–108, 1996.

[14] R. Koschke. Survey of research on software clones. In Koschke et al. [15].

[15] R. Koschke, E. Merlo, and A. Walenstein, editors. *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proc.* Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[16] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conf. on Reverse Engineering*, pages 301–309, Oct 2001.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[18] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, 1996.

[19] R. Rivest. The md5 message digest algorithm. RFC 1321, April 1992. Network Working Group.

[20] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[21] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03*, 2003.

[22] R. Smith. Copy detection systems for digital documents. Master's thesis, Department of Computer Science, Brigham Young University, 1999.

[23] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal of Computing*, 6(3):505–517, September 1977.

[24] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE*, 2003.