ISSN 0361-7688, Programming and Computer Software, 2018, Vol. 44, No. 5, pp. 335–343. © Pleiades Publishing, Ltd., 2018.
Original Russian Text © D.V. Luciv, D.V. Koznov, G.A. Chernishev, A.N. Terekhov, K.Yu. Romanovsky, D.A. Grigoriev, 2018, published in Programmirovanie, 2018, Vol. 44, No. 5.

Detecting Near Duplicates in Software Documentation[1]

D. V. Luciv[a],*, D. V. Koznov[a],, G. A. Chernishev[a],***, A. N. Terekhov[a],****, K. Yu. Romanovsky[a],*****, and D. A. Grigoriev[a],********

[a]*Saint Petersburg State University, St. Petersburg, 199034 Russia*

**e-mail: d.lutsiv@spbu.ru*

***e-mail: d.koznov@spbu.ru*

****e-mail: g.chernyshev@spbu.ru*

*****e-mail: a.terekhov@spbu.ru*

******e-mail: k.romanovsky@spbu.ru*

*******e-mail: d.a.grigoriev@spbu.ru*

Received August 8, 2017

Abstract—Contemporary software documentation is as complicated as the software itself. During its lifecycle, the documentation accumulates a lot of "near duplicate" fragments, i.e. chunks of text that were copied from a single source and were later modified in different ways. Such near duplicates decrease documentation quality and thus hamper its further utilization. At the same time, they are hard to detect manually due to their fuzzy nature. In this paper we give a formal definition of near duplicates and present an algorithm for their detection in software documents. This algorithm is based on the exact software clone detection approach: the software clone detection tool Clone Miner was adapted to detect exact duplicates in documents. Then, our algorithm uses these exact duplicates to construct near ones. We evaluate the proposed algorithm using the documentation of 19 open source and commercial projects. Our evaluation is very comprehensive — it covers various documentation types: design and requirement specifications, programming guides and API documentation, user manuals. Overall, the evaluation shows that all kinds of software documentation contain a significant number of both exact and near duplicates. Next, we report on the performed manual analysis of the detected near duplicates for the Linux Kernel Documentation. We present both quantative and qualitative results of this analysis, demonstrate algorithm strengths and weaknesses, and discuss the benefits of duplicate management in software documents.

Keywords: software documentation, near duplicates, documentation reuse, software clone detection

DOI: 10.1134/S0361768818050079

1. INTRODUCTION

Every year software is becoming increasingly more complex and extensive, and so does software documentation. During the software life cycle documentation tends to accumulate a lot of duplicates due to the copy and paste pattern. At first, some text fragment is copied several times, then each copy is modified, possibly in its own way. Thus, different copies of initially similar fragments become "near duplicates". Depending on the document type [1], duplicates can be either desired or not, but in any case duplicates increase documentation complexity and thus, maintenance and authoring costs [2].

Textual duplicates in software documentation, both exact and near ones, are extensively studied [2–5]. However, there are no methods for detection of near duplicates, only for exact ones and mainly using software clone detection techniques [2, 3, 6]. In our previous studies [7, 8] we presented a near duplicate detection approach. Its core idea is to uncover near duplicates and then to apply the reuse techniques described in our earlier studies [4, 5]. Clone detection tool Clone Miner [9] was adapted for detection of exact duplicates in documents, then near duplicates were extracted as combinations of exact duplicates. However, only near duplicates with one variation point were considered. In other words, the approach can detect only near duplicates that consist of two exact duplicates with a single chunk of variable text between them: $exact_1$ $variable_1$ $exact_2$.

In this paper we give the formal definition of near duplicates with an arbitrary number of variation points, exhibiting the following pattern: $exact_1$ $variable_1$ $exact_2$ $variable_2$ $exact_3$ ... $variable_{n-1}$ $exact_n$. Our definition is the formalized version of the definition given in the reference [10]. We also present a generalization of the algorithm described in [7, 8]. The algorithm is implemented in the Documentation Refactoring Toolkit

[1] The article is published in the original.

[11], which is a part of the DocLine project [4]. In this paper, an evaluation of the proposed algorithm is also presented. The documentation of 19 open source and commercial projects is used. The results of the detailed manual analysis of the detected near duplicates for the Linux Kernel Documentation [12] are reported.

2. RELATED WORK

Let us consider how near duplicates are employed in documentation-oriented software engineering research. Horie et al. [13] consider the problem of text fragment duplicates in Java API documentation. The authors introduce a notion of crosscutting concern, which is essentially a textual duplicate appearing in documentation. The authors present a tool named CommentWeaver, which provides several mechanisms for modularization of the API documentation. It is implemented as an extention of Javadoc tool, and provides new tags for controlling reusable text fragments. However, near duplicates are not considered, facilities for duplicate detection are not provided. Nosál and Porubän [14] extend the approach from [13] by introducing near duplicates. In this study the notion of documentation phrase is used to denote the near duplicate. Parametrization is used to define variative parts of duplicates, similarly to our approach [4, 5]. However, the authors left the problem of near duplicate detection untouched.

In [3] Nosál and Porubän present the results of a case study in which they searched for exact duplicates in internal documentation (source code comments) of an open source project set. They used a modified copy/paste detection tool, which was originally developed for code analysis and found considerable number of text duplicates. However, near duplicates were not considered in this paper.

Wingkvist et al. adapted a clone detection tool to measure the document uniqueness in a collection [6]. The authors used found duplicates for documentation quality estimation. However, they did not address near duplicate detection.

The work of Juergens et al. [2] is the closest one to our research and presents a case study for analyzing redundancy in requirement specifications. The authors analyze 28 industrial documents. At the first step, they found duplicates using a clone detection tool. Then, the authors filtered the found duplicates by manually removing false positives and performed a classification of the results. They report that the average duplicate coverage of documents they analyzed is 13.6%: some documents have a low coverage (0.9%, 0.7%, and even 0%), but there are ones that have a high coverage (35, 51.1, 71.6%). Next, the authors discuss how to use discovered duplicates and how to detect related duplicates in the source code. The impact of duplicates on the document reading process is also studied. Furthermore, the authors propose a classification of meaningful duplicates and false positive duplicates. However, it should be noted that they consider only requirement specifications and ignore other kinds of software documentation. Also, they do not use near duplicates.

Oumaziz et al. [15] analyze the API documentation of several well-known projects that was generated using the Javadoc technology. Their approach is based on employing a software clone detector for detecting duplicates in tags and methods. The authors provide a classification of duplicate types and explore the possibility of documentation reuse. However, near duplicates are not considered in this work, although it is mentioned that they are very common and important in practical tasks.

Rago et al. [16] apply natural language processing methods for near duplicate detection in textual descriptions of use cases. However, it should be noted that their work is dedicated to a rather peculiar type of requirement specifications that is rarely used in the industry. It is unclear how to apply this method to other types of software documentation.

Duplicate detection algorithms are developed in other research areas as well. The information retrieval community considers a number of tasks related to detection of near duplicates: detection of similar documents on the Internet [17, 18], detection of (local) reuse [19, 20], extraction of textual templates in web page collections [21, 22]. The plagiarism detection community is also studying duplicates in detail, and, in particular, near duplicates in documents [23–25]. However, these approaches are aimed at detecting similarity of whole documents and attaining high performance during processing large collections.

Traditionally, source code clone detectors are used for duplicate detection in software documentation [2, 3, 6, 15]. Let us note that the area of code clone detection has near duplicate detection tools. SourcererCC itesajnani2016 makes finding duplicate code fragments possible using the *static bag-of-tokens* strategy, which is insensitive to insignificant differences in these fragments. DECKARD [26] computes characteristic vectors of code for approximation of the structure of abstract syntax trees in an Euclidean space. NICAD [28] is a tool for detecting near duplicates for pretty-printing and transformation/filtering of source code. Other similar works include [29, 30]. It should be noted that such methods are not applicable for text documents since they employ syntactic code analysis. Concerning software documentation, these methods are used solely for exact duplicate detection, and only with the condition that they employ a token-based search. Nevertheless, using code clone detection as a base method for duplicate detection is appealing due to the following reasons:

- there is a large variety of readily available tools,

• the possibility of adding near duplicate detection,

• this approach is consistent with the archetype/delta concept [10] that we use for formalizing the definition of a near duplicate.

Following [16, 31], let us note that natural language processing methods are promising for our task. The following techniques could be useful:

• the N-gram model [31],

• topic modeling (allows to attribute text fragments to a particular area [32]),

• extraction of facts from free-form texts [33],

• text normalization techniques.

The application of these methods to our task is going to be the subject of our future research.

3. EXACT DUPLICATE DETECTION AND CLONE MINER

Not only documentation, but also software itself is often developed with a lot of copy/pasted information. To cope with duplicates in the source code, software clone detection methods are used. This area is quite mature; a systematic review of clone detection methods and tools can be found in [34]. In this paper, the Clone Miner [9] software clone detection tool is used to detect exact duplicates in software documentation. Clone Miner is a token-based source code clone detector. A token in the context of text documents is a single word separated from other words by some separator: '.', '(', ')', etc. For example, the following text fragment consists of 2 tokens: "FM registers". Clone Miner considers input text as an ordered collection of lexical tokens and applies suffix array-based string matching algorithms [35] to retrieve the repeated parts (clones). In this study we use the Clone Miner tool. We have selected it for its simplicity and its ability to be easily integrated with other tools using a command line interface.

4. NEAR DUPLICATE DEFINITION

Let us define the terms necessary for describing the proposed algorithm. We consider document D as a sequence of symbols. Any symbol of D has a coordinate corresponding to its offset from the beginning of the document, and this coordinate is a number belonging to $[1, length(D)]$ interval, where $length(D)$ is the number of symbols in D.

Definition 1. For D we define a **text fragment** as an occurrence of some text substring in D. Hence, each text fragment has a corresponding integer interval $[b, e]$, where b is the coordinate of its first symbol and e is the coordinate of its last symbol. For text fragment g of document D, we say that $g \in D$.

Let us introduce the following sets: D^* is a set of all text fragments of D, I^D is a set of all integer intervals within interval $[1, length(D)]$, S^D is a set of all strings of D.

Also, let us introduce the following notations:

• $[g] : D^* \to I^D$ is a function that takes text fragment g and returns its interval.

• $str(g) : D^* \to S^D$ is a function that takes text fragment g and returns its text.

• $\overline{I} : I^D \to D^*$ is a function that takes interval I and returns corresponding text fragment.

• $\|[b,e]\| : I^D \to [0, length(D)]$ is a function that takes interval $[g] = [b, e]$ and returns its length as $\|[g]\| = e - b + 1$. For simplicity, we will use $|g|$ notion instead of $\|[g]\|$.

• For any $g^1, g^2 \in D$ we consider their intersection $g^1 \cap g^2$ as intersection of corresponding intervals $[g^1] \cap [g^2]$, and $g^1 \subset g^2$ implies $[g^1] \subset [g^2]$.

• We define the binary predicate *Before* on $D^* \times D^*$, which is true for text fragments $g^1, g^2 \in D$, iff $e^1 < b^2$, when $[g^1] = [b^1, e^1]$, $[g^2] = [b^2, e^2]$.

Definition 2. Let us consider a set G of text fragments of D such that $\forall g^1, g^2 \in G\ (str(g^1) = str(g^2)) \wedge (g^1 \cap g^2 = \emptyset)$. We name those fragments as **exact duplicates** and G as **exact duplicate group** or **exact group**. We also denote number of elements in G as $\#G$.

Definition 3. For ordered set of exact duplicate groups $G_1, \ldots, G_N$, we say that it forms **variational group** $\langle G_1, \ldots, G_N \rangle$ when the following conditions are satisfied:

1. $\#G_1 = \ldots = \#G_N$.

2. Text fragments having similar positions in different groups, occur in the same order in document text: $\forall g_i^k \in G_i\ \forall g_j^k \in G_j\ ((i < j) \Leftrightarrow Before(g_i^k, g_j^k))$, and

$$\forall k \in \{1, \ldots, N-1\} Before(g_N^k, g_1^{k+1}).$$

We also say that for any G_k of this set $G_k \in VG$.

Note 1. According to condition 2 of definition 3, $\forall g_i^k \in G_i, \forall g_j^k \in G_j\ (i \neq j \Rightarrow g_i^k \cap g_j^k = \emptyset)$.

Note 2. When $VG = \langle G_1, \ldots, G_N \rangle$ and $VG' = \langle G'_1, \ldots, G'_M \rangle$, are variational groups, $\langle VG, VG' \rangle = \langle G_1, \ldots, G_N, G'_1, \ldots, G'_M \rangle$ is also a variational group in case when is satisfies definition3.

For example, suppose that we have $VG = \langle G_1, G_2, G_3 \rangle$ and each of G_i consists of three clones $g_i^k, i \in \{1, 2, 3\}$. Then, these clones appear in the text in the following order: $g_1^1 \dots g_2^1 \dots g_3^1 \dots \dots g_1^2 \dots g_2^2 \dots g_3^2 \dots \dots$ $g_1^3 \dots g_2^3 \dots g_3^3$. Next, it should be possible to compute the distance between variations or exact duplicate groups. This is required to support group merging inside our algorithm which selects several closest groups to form a new one. Thus, a distance function should be defined.

Definition 4. Distance between text fragments for any $g^1, g^2 \in D$ is defined as follows:

$$dist(g^1, g^2) = \begin{cases} 0, & g^1 \cap g^2 \neq \emptyset, \\ b^2 - e^1 + 1, & Before(g^1, g^2), \\ b^1 - e^2 + 1, & Before(g^2, g^1), \end{cases} \quad (1)$$

where $[g^1] = [b^1, e^1]$ and $[g^2] = [b^2, e^2]$.

Definition 5. Distance between exact groups G_1 and G_2, having $\#G_1 = \#G_2$, is defined as follows:

$$dist(G_1, G_2) = \max_{k \in \{1, \dots, \#G_1\}} dist(g_1^k, g_2^k). \quad (2)$$

Definition 6. Distance between variational groups VG_1 and VG_2, when there are $G_1 \in VG_1$, $G_2 \in VG_2 : \#G_1 = \#G_2$, is defined as follows:

$$dist(VG_1, VG_2) = \max_{G_1 \in VG_1, G_2 \in VG_2} dist(G_1, G_2). \quad (3)$$

Definition 7. Length of exact group G is defined as follows: $length(G) = \sum_{k=1}^{\#G} (e^k - b^k + 1)$, where $g^k \in G$, $[g^k] = [b^k, e^k]$.

Definition 8. Length of variational group $VG = \langle G_1, \dots, G_N \rangle$ is defined as follows:

$$length(VG) = \sum_{i=1}^{N} length(G_i). \quad (4)$$

Definition 9. Near duplicate group is such a variational group $\langle G_1, \dots, G_N \rangle$ that satisfies following condition for $\forall k \in \{1, \dots, \#G_1\}$:

$$\sum_{i=1}^{N-1} dist(g_i^k, g_{i+1}^k) \leq 0.15 * \sum_{i=1}^{N} \left| g_i^k \right|. \quad (5)$$

This definition is constructed according to the near duplicate concept from [36]: variational part of near duplicates with similar information (*delta*) should not exceed 15% of their exact duplicate (*archetype*) part.

Note 3. An exact group G can be considered as a variational one formed by itself: $\langle G \rangle$.

Definition 10. Consider near duplicate group $\langle G_1, G_2 \rangle$, where G_1 and G_2 are exact groups. We assume that this group contains a single **extension point**, and the text fragments contained in positions $[e_1^k + 1, b_2^k - 1]$ are called fextension point values. In the general case, a near duplicate group $\langle G_1, \dots, G_N \rangle$ has $N - 1$ extension points.

Definition 11. Consider two near duplicate groups $G = \langle G_1, \dots, G_n \rangle$ and $G' = G_1', \dots, G_m'$. Suppose that they form a variational group $\langle G_1, \dots, G_n, G_1', \dots, G_m' \rangle$ or $\langle G_1, \dots, G_n, G_1', \dots, G_m' \rangle$, which in turn is also a near duplicate group. In this case, we call G and G' **nearby groups**.

Definition 12. Nearby duplicates are duplicates belonging to nearby groups.

Note 4. Due to remark 3, definition 12 is applicable to both near and exact duplicates.

5. NEAR DUPLICATE DETECTION ALGORITHM

The algorithm that constructs the set of near duplicate groups (*SetVG*) is presented below. Its input is the set of exact duplicate groups (*SetG*) belonging to document *D*. It employs an interval tree — a data structure whose purpose is to quickly locate intervals that intersect with a given interval. Initially, the *SetG* set is created using the Clone Miner tool. The core idea of our algorithm is to repeatedly find and merge nearby exact groups from *SetG*. At each step, the resulting near duplicate groups are added to *SetVG*. Let us consider this algorithm in detail.

Algorithm 1: Near duplicate groups construction

```
   Input data: $SetG$
   Result: $SetVG$
1  $SetVG \leftarrow \emptyset$
2  $Initiate()$
3  repeat
4      $SetNew \leftarrow \emptyset$
5      foreach $G \in SetG \cup SetVG$ do
6          $SetCand \leftarrow NearBy(G)$
7          if $SetCand \neq \emptyset$ then
8              $G' \leftarrow GetClosest(G, SetCand)$
9              $Remove(G, G')$
10             if $Before(G, G')$ then
11                 $SetNew \leftarrow SetNew \cup \{\langle G, G' \rangle\}$
12             else
13                 $SetNew \leftarrow SetNew \cup \{\langle G', G \rangle\}$
14             end if
15         end if
16     end foreach
17     $Join(SetVG, SetNew)$
18 until $SetNew \neq \emptyset$
19 $SetVG \leftarrow SetVG \cup SetG$
```

The initial interval tree for *SetG* is constructed using the *Initiate*() function (line 2). The core part of the algorithm is a loop in which new near duplicate groups are constructed (lines 3–18). This loop repeats until we can construct at least one near duplicate group, i.e. the set of newly constructed near duplicate groups (*SetNew*) is not empty (line 18). Inside of this loop, the algorithm cycles through all groups of $SetG \cup SetVG$. For each of them, the *NearBy* function returns the set of nearby groups *SetCand* (lines 5, 6), which is then used for constructing near duplicate groups. Later, we will discuss this function in more detail and prove its correctness, i.e. that it actually returns groups that are close to *G*. Next, the closest group to *G*, denoted G', is selected from *SetCand* (line 8) and a variational group $\langle G, G'\rangle$ or $\langle G', G\rangle$ is created. This group is added into *SetNew* (lines 10–14). Since *G* and *G*' are merged and therefore cease to exist as independent entities, they are deleted from *SetG* and *SetVG* by the *Remove* function (line 9). Next, the *Join* function adds *SetNew* to *SetVG* (line 17). It is essential to note that the *Remove* and *Join* functions perform some auxiliary actions described below.

In the end of the algorithm *SetG* is added to *SetVG*. The result – *SetVG* – is presented as the algorithm's output. This step is required in order for the output to contain not only near duplicate groups, but also exact duplicate groups which havenot been used for creation of near duplicate ones (line 19).

Let us describe the functions employed in this algorithm.

The *Initiate*() function builds the interval tree. The idea of this data structure is the following.

Suppose we have *n* natural number intervals, where b_1 is the minimum and e_n is the maximum value of all interval endpoints, and *m* is the midpoint of $[b_1, e_n]$. The intervals are divided into three groups: fully located to the left of *m*, fully located to the right of *m*, and intervals containing *m*. The current node of the interval tree stores the last interval group and references to its left and right childnodes containing the intervals to the left and to the right of *m* respectively. This procedure is repeated for each child node. Further details regarding the construction of an interval tree can be found in [37, 38].

In this study, we build our interval tree from the extended intervals that correspond to the exact duplicates found by CloneMiner. These extended intervals are obtained as follows: original intervals belonging to exact duplicates are enlarged by 15%. For example, if $[b, e]$ is the initial interval, then an extended one is $[b - 0.15 * (e - b + 1),\ e + 0.15 * (e - b + 1)]$. We will denotethe extended interval that corresponds to the exact duplicate *g* as $\updownarrow g$. We also modify our interval tree as follows: each stored interval keeps the reference to the corresponding exact duplicate group.

The *Remove* function removes groups from sets and their intervals from the interval tree. The interval deletion algorithm is described in references [37, 38].

The *Join* function, in addition to the operations described above, adds intervals of the newly created near duplicate group $G = \langle G_1, \ldots, G_N\rangle$ to the interval tree. The standard insertion algorithm described in references [37, 38] is used. Extended intervals added to the tree of each near duplicate $g^k = (g_1^k, \ldots, g_N^k)$, where $k \in \{1, \ldots, \#G_1\}$, have the form of $[b_1^k - x^k, e_N^k + x^k]$, where $x^k = 0.15 * \sum_{i=1}^{N} \left|g_i^k\right| - \sum_{i=1}^{N-1} dist(g_i^k, g_{i+1}^k)$. We will denote this extended interval of g^k (now, a near duplicate) as $\updownarrow g^k$ as well.

The *NearBy* function selects nearby groups for some group *G* (its parameter). To do this, for each text fragment from *G* a collection of intervals that intersect with its interval is extracted. *Text fragments that correspond to these intervals turn out to be neighboring to the initial fragment*, i.e. for them, condition (5) is satisfied. The retrieval is done using the interval tree search algorithm [37, 38]. We construct the GL_1 set, which contains groups that are expected to be nearby to *G*:

$$GL_1(G) = \{G' | (G' \in SetG \cup SetVG) \wedge \exists g \in G, \quad g' \in G' : \updownarrow g \cap \updownarrow g' \neq \emptyset. \tag{6}$$

That is, the GL_1 set consists of groups that contain at least one duplicate that is close to at least one duplicate from *G*. Then, only the groups that can form a variational group with *G* are selected and placed into the GL_2 set:

$$GL_2(G) = \{G' | G' \in GL_1 \wedge (\langle G, G'\rangle \text{ or } \langle G', G\rangle \text{is variational group})\}. \tag{7}$$

Finally, the GL_3 set (the *NearBy* function's output) is created. The only groups placed in this set are those from GL_2 whose all elements are close to corresponding elements of *G*:

$$GL_3(G) = \{G' | G' \in GL_2 \wedge \forall k \in \{1, \ldots, \#G\} : \updownarrow g^k \cap \updownarrow g'^k \neq \emptyset\}. \tag{8}$$

Theorem 1. *Suggested algorithm detects near duplicate groups that conform to definition* 9.

It is easy to show by construction of *NearBy* that for some group *G* it returns the set of its nearby groups (see definition 11). That is, each of these groups can be used to form a near duplicate group with *G*. Then for the set the algorithm selects the group closest to *G* and constructs a new near duplicate group. The correctness of all intermediate sets and other used functions is immediate from their construction methods.

Table 1. Near-duplicate groups detected

Document	Size, Kb	Time, S	Total near dup groups	Exact duplicate groups, %	1-ext. pt. groups, %	2-ext. pt. groups, %	3-ext. pt. groups, %
1	892	46	1291	93.9	5.1	0.9	0.2
2	2924	188	6056	93.3	5.3	0.9	0.2
3	1810	134	4220	95.9	3.4	0.5	0.2
4	686	32	1500	96.5	3.3	0.2	0.1
5	1311	110	4688	95.9	3.3	0.5	0.0
6	3136	321	6587	93.9	5.1	0.7	0.2
7	1491	173	4537	92.0	6.0	1.2	0.4
8	3160	218	7804	95.6	3.5	0.5	0.2
9	1104	16	152	93.4	5.3	0.7	0.7
10	1800	186	4685	92.7	5.8	0.9	0.3
11	1056	66	2436	91.5	6.7	1.1	0.3
12	36	3	59	83.1	11.9	1.7	0.0
13	166	8	392	89.5	9.7	0.5	0.3
14	103	4	208	91.3	7.7	0.5	0.0
15	98	4	117	94.0	4.3	0.9	0.9
16	241	12	394	88.8	9.1	0.8	0.3
17	43	2	16	81.3	12.5	6.3	0.0
18	50	2	77	88.3	11.7	0.0	0.0
19	167	7	145	88.3	9.7	0.7	1.4

Let us evaluate the complexity of the proposed algorithm. Consider one iteration of the **repeat...until** loop. Let N_i be the total number of duplicates in groups from $SetG \cup SetVG$ before the i-th iteration. Remember that this search is performed with the use of an interval tree. Therefore, according to [38], the complexity of detecting duplicates that are nearby to the given duplicate is, on average, $\mathbb{O}(M + \log N_i)$, where M is the maximum number of nearby duplicates that were found in such situations during all iterations of the algorithm. Thus, the average complexity of every iteration of this loop is $\mathbb{O}(N_i * (M + \log N_i))$. The **repeat...until** loop iterates $E + 1$ times at most, where E is the maximum number of extension points of groups in $SetVG$ after the algorithm stops. In the end, the complexity of the algorithm can be estimated on average as $\mathbb{O}((E + 1) * N * (M + \log N))$, where $N = N_1$ is the number of exact duplicates in the initial document (i.e., in our case, this is the cardinality of Clone Miner's output). N fluctuates a lot for real documents — it reached tens of thousands in our experiments, while M did not exceed 10, and E did not exceed 20. Therefore, we can conclude that the average complexity of this algorithm is $\mathbb{O}(N * \log N)$. The actual execution time of the algorithm during the processing of documents of different sizeis presented in Table 1. This table shows that the complexity mainly depends on the number of exact duplicates, and not the size of the analyzed document.

6. EVALUATION

The proposed algorithm was implemented in the Duplicate Finder Toolkit [11]. Our prototype uses the intervaltree library [39] as an implementation of the interval tree data structure.

We have evaluated 19 industrial documents belonging to various types: requirement specification, programming guides, API documentation, user manuals, etc. (see Table 1). The size of the evaluated documents is up to 3 Mb.

Our evaluation produced the following results. The majority of the duplicate groups detected are exact duplicates (88.3–96.5%). Groups having one variation point amount to 3.3–12.5%, two variation points – to 0–1.7%, and three variation points – to less 1%, etc. A few near duplicates with 11, 12 13, and 16 variation points also were detected. We performed a manual analysis of the automatically detected near duplicates for Linux Kernel Documentation (programming guide, document 1 in the Table 1) [12]. We found 70 meaningful text groups (5.4%), 30 meaningful groups for the code example (2.3%), and 1191 false positive groups (92.3%). We found 21 near duplicate groups, i.e. 21% of the meaningful duplicate groups. Therefore, the share of near duplicates significantly increases after discarding false positives.

Having analyzed the evaluation results, we can make the following conclusions:

1. During our experiments we did not manage to find any near duplicates in considered documents that were not detected by our algorithm. However, we should note that the claim of the algorithm's high recall needs a more detailed justification.

2. Analyzing the Linux Kernel Documentation, we have concluded that it does not have any cohesive style: it was created sporadically by different authors. Virtually all its duplicates are situated locally, i.e. close to each other. For example, some author created a description of some driver's functionality using copy/paste for its similar features. At the same time, another driver was described by a different author who did not use the first driver's description at all. Consequently, there are practically no duplicates that are found throughout the whole text. Examples, warnings, notes, and other documentation elements that are preceded by different introductory sentences are not styled cohesively as well. Thus, our algorithm can be used for analyzing the degree of documentation uniformity.

3. The algorithm performs well on two-element groups, finding near duplicate groups with a different number of extension points. It appears that, in general, there are way fewer near duplicate groups with more than two elements.

4. Many detected duplicate groups consist of figure and table captions, page headers, parts of the table of contents and so on – that is, they are not of any interest to us. Also, many found duplicates are scattered across different elements of document structure, for example, a duplicate can be a part of a header and a small fragment of text right after it. These kinds of duplicates are not desired since they are not very useful for document writers. However, they are detected because currently document structure is not taken into account during the search.

5. The 0.15 value used in detecting near duplicate groups does not allow to find some significant groups (mainly small ones, 10–20 tokens in size). It is possible that it would be more effective to use some function instead of a constant, which could depend, for example, on the length of the near duplicate.

6. Moreover, often the detected duplicate does not contain variational information that is situated either in its end or in its beginning. Sometimes it could be beneficial to include it in order to ensure semantic completeness. To solve this problem, aclarification and a formal definition of semantic completeness of a text fragment is required. Our experiments show that this can be done in various ways (the simplest one is ensuring sentence-level granularity, i.e. including all text until the start/end of sentence).

7. Processing real documents, the algorithm showed an acceptable execution time – 5.4 minutes in the worst case, with the average of 1.3 minutes.

7. CONCLUSION

In this paper the formal definition of near duplicates in software documentation is given and the algorithm for near duplicate detection is presented. An evaluation of the algorithm using a large number of both commercial and open source documents is performed.

The evaluation shows that various types of software documentation contain a significant number of exact and near duplicates. A near duplicate detection tool could improve the quality of documentation, while a duplicate management technique would simplify documentation maintenance.

Although the proposed algorithm provides ample evidence on text duplicates in industrial documents, it still requires improvements before it can be applied to real-life tasks. The main issues to be resolved are the quality of the near duplicates detected and a large number of false positives. Also, a detailed analysis of near duplicate types in various sorts of software documents should be performed.

Furthermore, a detailed analysis of near duplicate types common to different documentation types is required. Integration of documentation reuse (in particular, requirement specification) with automated test development [40, 41], and diagrammatic modeling of duplicate structure [40] could be interesting courses of development for our work.

ACKNOWLEDGMENTS

This work is partially supported by RFBR grant no. 16-01-00304.

REFERENCES

1. Parnas, D.L., Precise documentation: The key to better software, in *The Future of Software Engineering,* Berlin, Heidelberg: Springer-Verlag, 2011, pp. 125–148.
2. Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., and Streit, J., Can clone detection support quality assessments of requirements specifications?, in *Proceedings of the 32 ACM/IEEE International Conference on Software Engineering (ICSE'10)*, New York, NY, USA: ACM, 2010, vol. 2, pp. 79–88.
3. Nosál', M. and Porubän, J., Preliminary report on empirical study of repeated fragments in internal documentation, *Proceedings of Federated Conference on Computer Science and Information Systems,* 2016, pp. 1573–1576.
4. Koznov, D.V. and Romanovsky, K.Yu., DocLine: A method for software product lines documentation development, *Program. Comput. Software,* 2008, vol. 34, no. 4, pp. 216–224.

5. Romanovsky, K., Koznov, D., and Minchin, L., Refactoring the documentation of software product lines, *Lecture Notes in Compute Science,* Berlin, Heidelberg: Springer-Verlag, 2011, vol. 4980 of CEE-SET 2008, pp. 158–170.

6. Wingkvist, A., Lowe, W., Ericsson, M., and Lincke, R., Analysis and visualization of information quality of technical documentation, *Proceedings of the 4th European Conference on Information Management and Evaluation,* 2010, pp. 388–396.

7. Koznov, D., Luciv, D., Basit, H.A., Lieh, O.E., and Smirnov, M., Clone detection in reuse of software technical documentation, *International Andrei Ershov Memorial Conference on Perspectives of System Informatics, 2015,* Springer Nature, 2016, vol. 9609 of *Lecture Notes in Computer Science*, pp. 170–185.

8. Luciv, D.V., Koznov, D.V., Basit, H.A., and Terekhov, A.N., On fuzzy repetitions detection in documentation reuse, *Program. Comput. Software,* 2016, vol. 42, no. 4, pp. 216–224.

9. Basit, H.A., Puglisi, S.J., Smyth, W.F., Turpin, A., and Jarzabek, S., Efficient token based clone detection with flexible tokenization, *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers,* New York, NY, USA: ACM, 2007, pp. 513–516.

10. Bassett, P.G., Framing software reuse: Lessons from the real World, Upper Saddle River, NJ, USA: Prentice-Hall, 1997.

11. Documentation Refactoring Toolkit. http://www.math.spbu.ru/user/kromanovsky/docline/index_en.html.

12. Torvalds, L., Linux Kernel Documentation, Dec 2013 snapshot. https://github.com/torvalds/linux/tree/master/Documentation/DocBook/.

13. Horie, M. and Chiba, S., Tool support for crosscutting concerns of API documentation, *Proceedings of the 9th International Conference on Aspect-Oriented Software Development,* New York, NY, USA: ACM, 2010, pp. 97–108.

14. Nosál', M. and Porubän, J., Reusable software documentation with phrase annotations, *Central Europ. J. Comput. Sci.,* 2014, vol. 4, no. 4, pp. 242–258.

15. Oumaziz, M.A., Charpentier, A., Falleri, J.-R., and Blanc, X., Documentation reuse: Hot or not? An empirical study, *Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, 2017, Proceedings,* Botterweck, G. and Werner, C., Eds., Cham: Springer-Verlag, 2017, pp. 12–27.

16. Rago, A., Marcos, C., and Diaz-Pace, J.A., Identifying duplicate functionality in textual use cases by aligning semantic actions, *Software Syst. Model.,* 2016, vol. 15, no. 2, pp. 579–603.

17. Huang, T.-K., Rahman, Md.S., Madhyastha, H.V., Faloutsos, M., and Ribeiro, B., An analysis of socware cascades in online social networks, *Proceedings of the 22Nd International Conference on World Wide Web,* New York, NY, USA: ACM, 2013, pp. 619–630.

18. Williams, K. and Giles, C.L., Near duplicate detection in an Academic Digital Library, *Proceedings of the ACM Symposium on Document Engineering,* New York, NY, USA: ACM, 2013, pp. 91–94.

19. Zhang, Q., Zhang Yu., Yu, H., and Huang, X., Efficient partial-duplicate detection based on sequence matching, *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval,* New York, NY, USA: ACM, 2010, pp. 675–682.

20. Abdel Hamid, O., Behzadi, B., Christoph, S., and Henzinger, M., Detecting the origin of text segments efficiently, *Proceedings of the 18th International Conference on World Wide Web,* New York, NY, USA: ACM, 2009, pp. 61–70.

21. Ramaswamy, L., Iyengar, A., Liu, L., and Douglis, F., Automatic detection of fragments in dynamically generated web pages, *Proceedings of the 13th International Conference on World Wide Web,* New York, NY, USA: ACM, 2004, pp. 443–454.

22. Gibson, D., Punera, K., and Tomkins, A., The volume and evolution of web page templates, *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web,* New York, NY, USA: ACM, 2005, pp. 830–839.

23. Vall'es, E. and Rosso, P., Detection of near-duplicate user generated contents: The SMS spam collection, *Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents,* New York, NY, USA: ACM, 2011, pp. 27–34.

24. Barrón-Cedeño, A., Vila, M., Martí, M., and Rosso, P., Plagiarism meets paraphrasing: Insights for the next generation in automatic plagiarism detection, *Comput. Linguist.,* 2013, vol. 39, no. 4, pp. 917–947.

25. Antiplagiarism (in Russian). https://www.antiplagiat.ru/. Accessed January 16, 2018.

26. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., and Lopes, C.V., SourcererCC: Scaling code clone detection to big-code, *Proceedings of the 38th International Conference on Software Engineering,* New York, NY, USA: ACM, 2016, pp. 1157–1168.

27. Jiang, L., Misherghi, G., Su, Z., and Glondu, S., DECKARD: Scalable and accurate tree-based detection of code clones, *Proceedings of the 29th International Conference on Software Engineering,* Washington, DC, USA: IEEE Computer Soc., 2007, pp. 96–105.

28. Cordy, J.R. and Roy, C.K., The NiCad clone detector, in *Proceedings of IEEE 19th International Conference on Program Comprehension,* 2011, pp. 219–220.

29. Akhin, M. and Itsykson, V., Tree slicing in clone detection: Syntactic analysis made (semi)-semantic (in Russian), *Model. Anal. Inform. Syst.*, 2012, vol. 19, no. 6, pp. 69–78.

30. Zeltser, N.G., Automatic clone detection for refactoring, *Proc. Inst. Syst. Program.,* 2013, vol. 25, pp. 39–50.

31. Wagner, S. and Fernández, D.M., Analyzing text in software projects, *The Art and Science of Analyzing Software Data,* Elsevier, 2015, pp. 39–72.

32. Korshunov, A. and Gomzin, A., Topic modeling in natural language texts (in Russian), *Proc. Inst. Syst. Program.,* 2012, vol. 23, pp. 215–242.

33. Tomita-parser – Yandex Technologies (in Russian). https://tech.yandex.ru/tomita/. Accessed January 16, 2018.

34. Rattan, D., Bhatia, R., and Singh, M., Software clone detection: A systematic review, *Inform. Software Technol.*, 2013, vol. 55, no. 7, pp. 1165–1199.

35. Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E., Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algorithms,* 2004, vol. 2, no. 1, pp. 53–86.

36. Bassett, P.G., The theory and practice of adaptive reuse, *SIGSOFT Software Eng. Notes,* 1997, vol. 22, no. 3, pp. 2–9.

37. de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M., *Computational Geometry,* Berlin: Springer, 2008, pp. 220–226.

38. Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction,* Berlin: Springer-Verlag, 1985, pp. 359–363.

39. PyIntervalTree. URL: https://github.com/chaimleib/intervaltree.

40. Kolchin, A.V., Kotljarov, V.P., and Drobincev, P.D., The method of test scenariogeneration in the environment of the insertion modeling, *Control Syst. Mach.,* 2012, no. 6, pp. 43–48, 63.

41. Pakulin, N.V. and Tugaenko, A.N., Model-based testing of Internet Mail Protocols, *Proc. Inst. Syst. Program.,* 2011, vol. 20, pp. 125–141.

42. Kudryavtsev, D. and Gavrilova T., Diagrammatic knowledge modeling for managers: Ontologybased approach, *Proceedings of the International Conference on Knowledge Engineering and Ontology Development,* 2011, pp. 386–389.