Shuffling and Randomization for Scalable Source Code Clone Detection

±Iman Keivanloo, *Chanchal K. Roy, ±Juergen Rilling, ×Philippe Charland

±*Department of Computer Science*
Concordia University, Canada
{i_keiv, rilling@cse.concordia.ca}

**Department of Computer Science*
University of Saskatchewan, Canada
croy@cs.usask.ca

×*System of Systems Section*
Defence R&D Canada – Valcartier
philippe.charland@drdc-rddc.gc.ca

***Abstract*—In this research, we present a novel approach that allows existing state of the art clone detection tools to scale to very large datasets. A key benefit of our approach is that the improved tools scalability is achieved using standard hardware and without modifying the original implementations of the subject tools. We use a hybrid approach comprising of shuffling, repetition, and random subset generation of the subject dataset. As part of the experimental evaluation, we applied our shuffling and randomization approach on two state of the art clone detection tools. Our experience shows that it is possible to scale the classical tools to a very large dataset using standard hardware, and without significantly affecting the overall recall while exploiting all the strengths of the original tools including the precision.**

***Keywords*-Clone detection; scalability; shuffling; sampling**

I. Introduction

SeCold (secold.org) is an online Linked Data project that provides a hub for source code related facts using Semantic Web technologies. For its first release, we extracted about two billion facts from our IJaDataset, which contains about 1.5 million Java files (266 MLOC) from 18,000 projects.

The objective of SeCold is to establish a "DBpedia" equivalent (service) for the software mining community. Recently, we started to enrich the SeCold repository with additional *clone facts,* by publishing result sets obtained from several state of the art clone detection tools. The input is the source code from all indexed projects (i.e., the IJaDataset) and the output, all the clones it contains. However, key challenges for applying some of the existing clone detection tools on our SeCold dataset are: (1) The size of the dataset itself, which we expect to significantly grow in the next releases, with the inclusion of additional open source code repositories, and (2) enabling existing clone detection tools to scale to our SeCold dataset using standard hardware (one multi-core CPU and 24 GB of RAM), without modifying the original implementations of the subject tools.

II. Solution

While scalable, distributed clone detection approaches (e.g., [1, 2]) require significant computational resources and attempt to achieve 100% recall (tool specific gold standards). In contrast, our objective is to allow existing clone detection tools to scale to very large datasets, while using limited computational resources and achieving an acceptable overall recall. For the SeCold dataset and its global code analysis context, we consider 85% recall as an acceptable objective. Our approach is based on a simplified *divide and conquer* (DC) paradigm without feedback or recursion loop used by other approaches (e.g., MapReduce). We (1) split the corpus into n subsets, which are manageable in size by existing clone detection tools, (2) execute the detection tools on each subset independently, and (3) upload the results from the subsets into our central clone pair repository (i.e., SeCold).

In what follows, we assume that all subsets, denoted by s, are of equal size. Therefore, if C represents the total corpus size, the relation $C = n \times s$ must be satisfied. We further assume that the default *unit of measurement* for n is the number of files. However, if a tool requires a complete compilation unit (i.e., a software project) as the smallest input, then the unit (n) corresponds to the number of projects. This approach will generate the highest recall only if $n = 1$ (i.e., the corpus is not split in subsets). That is, the complete corpus is treated as a uniform dataset, containing all potential clones, and resulting in the highest recall of the individual tools. If $n \geq 2$, recall decreases, since the tool has no longer access at detection time to all potential clone pairs, as they might be located in different subsets (e.g., *Subset* i and *Subset* j, where $i \neq j$).

We require large n values to reduce the size of the individual subsets so that they can be handled by existing clone detection tools (i.e., $n = 15$ for the IJaDataset). However, achieving an acceptable overall recall (i.e., ~85%) becomes a challenge for a large number of subsets (n).

As part of our research, we have derived a heuristic that repeats several times the same three-step process for partitioning the dataset, executing the clone detection tools, and uploading the results to our repository. As part of our heuristic, for each round (r) we apply a *shuffling function* to create random subsets that differ from the subsets of the previous rounds. Figure 1 illustrates the shuffling and randomization approach. Rather than using recursion typically used by *divide and conquer* algorithms (e.g. MapReduce), we rely on repetition (up to r rounds), to increase the overall *cumulative recall.*

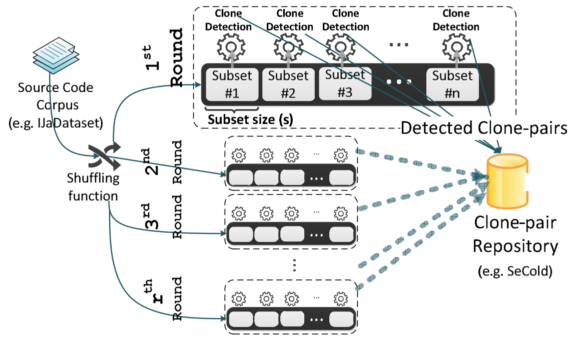
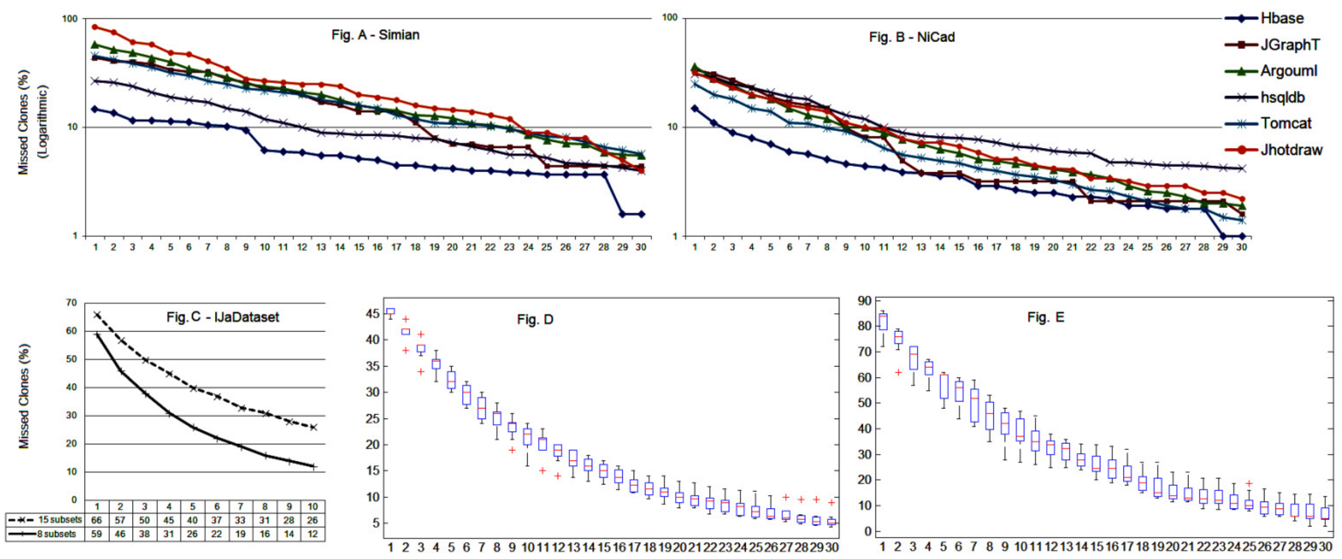


Figure 1. Overview of the shuffling approach. Arrows represent data flow. There is no ordering or dependency between rounds and subsets.



	1	2	3	4	5	6	7	8	9	10
15 subsets	66	57	50	45	40	37	33	31	28	26
8 subsets	59	46	38	31	26	22	19	16	14	12

Figure 2. Experimental results overview. The horizontal and vertical axes are the *number of rounds* and the *missed clones (%) respectively*.

III. Experimental Results

We conducted an experimental evaluation to determine the actual recall of our shuffling approach for a given dataset (i.e., IJaDataset), using the following three steps: (1) execute each detection tool on the complete (single) dataset to create a gold standard for the number of clones it can detect; (2) perform the randomize shuffling and divide approach on the complete dataset to create a fixed number of subsets; and (3) execute the clone detection tool on all subsets and calculate the total recall (tr) achieved, with $tr(r,n) = \frac{\sum_1^r(\sum_1^n detected\ clones)}{clones\ in\ oracle}$. Steps 2 and 3 are repeated for a fixed number of times, to determine the actual improvement in recall and therefore, the reduction in % of missed clones.

Preliminary observations. For the initial experiment, we applied the random shuffling approach on six small datasets. We used two clone detection tools, Simian [3] (line-level) and NiCad [4] (method-level granularity) on them (no subsets) to establish the tool specific gold standard. For steps 2 and 3, we created 15 subsets. We performed a total of 30 shuffling rounds (Fig 2-A and 2-B), with NiCad detecting 97.8% and Simian 96.6% of total recall. Moreover, NiCad achieved ~90% recall after 10 shuffling rounds, whereas Simian needed 20 rounds. This variation can be attributed to the different granularity of the tools.

IJaDataset. For the final part of the evaluation, we repeated the experiment on our complete IJaDataset (1.5 million files). Given our limited hardware, Simian was the only tool to complete the actual clone detection on the IJaDataset and create its gold standard. We further differentiate two setups, one with 15 and the other with 8 subsets of the IJaDataset. For the 15-subset experiment, the run-time for each round was approximately 32 hours. In both cases, we performed 10 rounds of shuffling (Fig. 2-C) and achieved 74% and 88% recalls for 15 and 8 subsets respectively. Note that, we believe that higher recall is achievable using our shuffling idea and NiCad according to our *preliminary observations* (Fig 2-A and 2-B). Since they show using a medium-granularity tool, we can achieve higher recall in less number of rounds (50% less).

Randomization effect. Since our shuffling approach is based on a random creation of subsets, we analyzed the effect of this random selection on the overall performance (recall) of our approach. We extracted two datasets, with the second dataset being three times the size of the first one. We then created 15 subsets per dataset and executed 30 shuffling rounds. The randomization effect was analyzed by repeating each shuffling round 10 times. Fig. 2-D (large dataset) and Fig. 2-E (small dataset) show the fluctuation due to the randomization. The result clearly shows that for a larger dataset (e.g., IJaDataset), the fluctuation has almost no effect and can therefore be neglected.

IV. Conclusion and Future Work

Our experience shows that the proposed shuffling approach can be effectively used to scale existing clone detection tools for large data using limited hardware with an acceptable loss of recall (without sacrificing precision). The approach may outperform (run-time) existing tools even in cases when they process IJaDataset without partitioning. As future work, we plan to (1) apply shuffling to other tools and devise a heuristic to estimate the recall, and (2) combine our approach with the sampling technique earlier [5].

References

[1] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable", Proc. ICSM, 2010, pp. 1-9.

[2] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder", Proc. ICSE, 2007, pp. 106-115.

[3] Simian, http://www.harukizaemon.com/simian/, Visited Jan. 2012.

[4] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. ICPC, 2008, pp. 172-181.

[5] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale Real-time Code Clone Search via Multi-level Indexing", Proc. WCRE, 2011, pp. 23-27.