DCT: An Scalable Multi-Objective Module Clustering Tool

Ana Paula M. Tarchetti*, Luís Amaral*, Marcos C. Oliveira†, Rodrigo Bonifácio*, Gustavo Pinto‡ and David Lo§
*University of Brasília. Emails: aptarchetti@gmail.com, luis.amaralh@gmail.com, rbonifacio@unb.br
†Brazilian Ministry of Economy. Email: mail@mcesar.dev
‡Federal University of Pará. Email: gpinto@ufpa.br
§Singapore Management University. Email: davidlo@smu.edu.sg

***Abstract*—Maintaining complex software systems is a time-consuming and challenging task. Practitioners must have a general understanding of the system's decomposition and how the system's developers have implemented the software features (probably cutting across different modules). Re-engineering practices are imperative to tackle these challenges. Previous research has shown the benefits of using software module clustering (SMC) to aid developers during re-engineering tasks (e.g., revealing the architecture of the systems, identifying how the concerns are spread among the modules of the systems, recommending refactorings, and so on). Nonetheless, although the literature on software module clustering has substantially evolved in the last 20 years, there are just a few tools publicly available. Still, these available tools do not scale to large scenarios, in particular, when optimizing multi-objectives. In this paper we present the *Draco Clustering Tool* (DCT), a new software module clustering tool. DCT design decisions make multi-objective software clusterization feasible, even for software systems comprising up to 1,000 modules. We report an empirical study that compares DCT with other available multi-objective tool (HD-NSGA-II), and both DCT and HD-NSGA-II with mono-objective tools (BUNCH and HD-LNS). We evidence that DCT solves the scalability issue when clustering medium size projects in a multi-objective mode. In a more extreme case, DCT was able to cluster Druid (an analytics data store) 221 times faster than HD-NSGA-II.**

***Index Terms*—Software Module Clustering, Multi-Objective Optimization, Genetic Algorithms**

I. Introduction

With increasing complexity of modern software, there is an increased demand for automated tools to support the maintainability and scalability of those systems (Dahiya et al. [1]). Fundamental contributions to this subject include, for instance, the introduction of the automated Software Module Clustering (SMC) tool by Mitchell and Mancoridis [2]. This appliance began with the purpose to offer techniques to reveal the structure of a software system by grouping its modules into clusters. They based their algorithm on the principle of "low coupling and high cohesion". The input of the algorithm is a set of modules and dependencies between them. Typically, these modules correspond to files (or classes, in object-oriented programming languages), and the dependencies correspond to function/method calls or variables/fields access. While this kind of modules and dependencies are common, other representations are useful too, such as methods/fields as modules and co-change dependencies [3].

Revealing the software structure by using SMC tools can help to overcome complications related to misleading or insufficient documentation. The problem with documentations is accurately comprehended in Lethbridge et al. [4], this study is consisted of interviews with software engineers, and the general answers about documentation was the following: documentation is frequently out of date, often poorly written, challenging in terms of finding useful content and has a considerable untrustworthy fraction. In this context, it becomes very meaningful the chase for computational mechanisms such as SMC so that the documentation gap could be filled, hence, making it possible to support the six main aspects of software development pointed out by Garlan [5]: understanding, reuse, construction, evolution, analysis, and management.

Besides software structure recovering, SMC techniques can also be used to: (a) recommend or reveal alternative decompositions [3], (b) recommend refactorings in order to conform to some alternative decomposition [6], and (c) detect anomalies in the software design [3], [6].

Past researches have proposed many alternative SMC approaches [7]–[13], however, they failed to provide publicly available tools that use multi-objective genetic algorithms in their designs. One of the primary benefits of multi-objective algorithms is that it outputs a set of best solutions in contrast with mono-objective where there is only one "best" solution. The problem with pursuing only one solution is that we have to chose between conflicting objectives. For example, it is hard to chose between a solution with better cohesion or other with better coupling; i.e. for a SMC tool to find the best solution among several candidate solutions they have to decide about questions like "which is better: coupling or cohesion?" [16], [17].

In order to overcome this problem, in this paper we present the Draco Clustering Tool (DCT), a public tool that performs automated SMC using multi-objective genetic algorithms.

II. Background and Related Work

The re-engineering process in large scale software projects require appropriate and scalable techniques. With the focus

on software module clustering (SMC) techniques, the work of Anquetil and Lethbridge [14], for instance, compares different strategies for using SMC as a software remodularization recommender. More recently, Maqbool and Babri [15] investigate the use of hierarchical clustering algorithms for architecture recovering.

Given this context, it is noticeable that the majority of SMC approaches use mono-objective algorithms. Praditwong et al. [16] proposed to represent the SMC problem as a multi-objective search problem. They formulated the problem representing separately several different objectives (including cohesion and coupling). The rationale of this proposal is that it is not always possible to capture the relative importance of some desirable properties (for example, it is hard to decide if cohesion is more important than coupling or vice-versa).

Candela et al. [17], investigated which properties developers consider relevant for a high-quality software remodularization. To be able to compare different properties, they had to use a multi-objective genetic algorithm to compute the software module clusters. Accordingly, they presented to the developers several recommendations of remodularization, and investigated which property (e.g. cohesion or coupling) the developers regards most. This kind of study was only possible by using a multi-objective SMC tool.

Other works are also worth mention here, because provide different SMC implementations. First, M. Barros discusses the effects of using the MQ metric as an extra objective on a multi-objective SMC tools [18]. Second, Monçores et al. present a large study addressing a heuristic based on the mono-objective *Large Neighborhood Search* algorithm, applied to SMC problems [19]. Both works publish tools that we explored in this paper. Finally, in a recent work, work [6] we leveraged a multi-objective software module clustering tool to produce a set of alternative decompositions of a software. Our needs to use a multi-objective approach to find these alternative decompositions, and the lack of scalable multi-objective SMC tools, motivated us to implement DCT.

III. Draco Clustering Tool

Draco Clustering Tool (DCT) is a command line interface (CLI) tool, that reads a *Module Dependency Graph* (MDG) [20] from the standard input and writes a clustered graph represented as a DOT[1] file in the standard output. It was implemented in Go[2] programming language, and is publicly available.[3] A typical invocation of the tool looks like this:

```
$ clustering < software.mdg > software.dot
```

The main use case of the tool is to run experiments involving multi-objective SMC computation. Accordingly, the following principles guided the design of DCT:

- **An easy to use interface.** While a Graphical User Interface potentially could be more intuitive, it makes experiments automation more difficult;
- **Minimal memory usage.** DCT users might want to run the tool in parallel, so its memory consumption must be minimal;
- **Runtime efficiency.** Similarly, the time spent running a experiment must be minimal;
- **Extensible.** To experiment with multiple scenarios, it must be possible to replace portions of the clustering algorithm or to tune its parameters values;
- **Standard formats.** To make comparisons of DCT with other tools easier, DCT must adopt well-known file formats, both for input (MDG) and output (DOT);

In order to address this principles, we chose Go as programming language. Go programs are compiled ahead of time to native machine code, therefore compiled programs can execute efficiently. Furthermore, this property makes the use of CLI tools more convenient, since they would not require a virtual machine to run. In addition, we address the extensibility principle using Go interfaces. For instance, we have a Go interface to abstract the random number generator (see more details bellow).

In DCT we used the definition of the SMC problem as a multi-objective optimization problem, using the same set of objects recommended by Praditwong et al. [16]. The input is a MDG represented by a graph $G = (V, E)$ from a set of modules V and a set of dependencies $E \subseteq V \times V$; and the output is a set of solutions. A solution is a partition of a MDG that corresponds to a set of clusters. Although the original design of DCT uses a multi-objective genetic algorithm (GA) [21] to compute optimal partitions, it is also possible to extend DCT to use mono-objective algorithms.

To use a genetic algorithm, it is necessary to precisely define the concept of *individuals* and *fitness functions* for the problem domain. A typical GA executes as follows:

1) It first generates an initial population (i.e., a set of individuals) randomly;
2) It repeatedly produces a new population, by (a) selecting individuals from the previous population using the fitness values and (b) combining them using the genetic operators *crossover* and *mutation*;
3) It proceeds until a stop condition is met.

In DCT, each GA component (e.g., the fitness function or the crossover operator) is defined as Go interfaces, which enables the replacement for other implementations. The default implementations of these interfaces are specified next.

The default DCT implementation relies on the multi-objective genetic algorithm NSGA-II [22], responsible to implement the selection operator of the GA.

When using multi-objective GAs, each individual has a vector of fitness values [21]. To compare two individuals, we use the concept of *Pareto Dominance*: a vector v dominates another vector u if no value v_i is smaller than the value u_i, and at least one v_j is greater than u_j [21] (this applies to optimizations where the goal is to maximize the objective values, if the goal is the opposite, we must invert the comparisons).

As such, we represent the individuals as a mapping from a module to the cluster it belongs to (typically a module

[1] https://graphviz.org/doc/info/lang.html
[2] https://golang.org
[3] https://github.com/project-draco/tools/tree/master/clustering

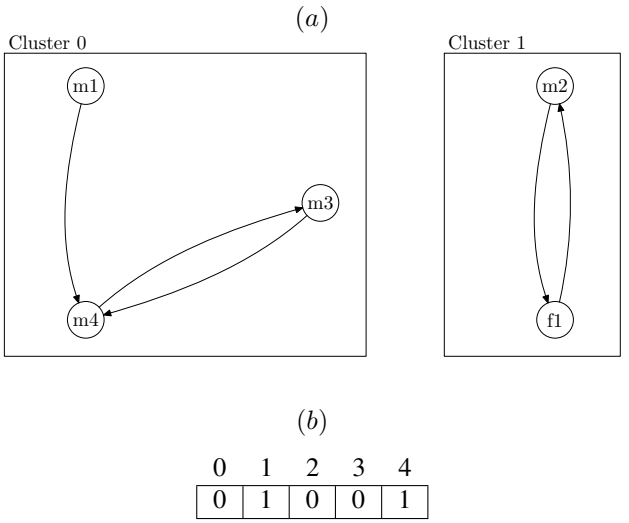

Fig. 1. Individual representation.

represents a file or class). Technically, an individual is an array where each position corresponds to a module, and each value corresponds to a cluster. Two different modules belong to the same cluster when they refer to the same value. Figure 1-(a) illustrates this representation, showing four modules (m1, m2, m3, m4, f1). All modules belong to the cluster C_0, except for m2 that belongs to the cluster C_1 (together with module f1).

Differently from previous works [19], [20], [23], DCT saves computer's main memory since the array is codified as a binary string (i.e., as a sequence of bits), as we can see in Figure 1-(b). The maximum number of clusters is set to $\frac{|V|}{2}$, and each element of the array occupies $\left\lceil \log_2 \frac{|V|-1}{2} \right\rceil$ bits of the binary string—where V is the set of vertices of the MDG. Previous works represent the individual as an array of "integers" [19], [20], [23], which could place a toll on today processors that take 64 bits. For example, if we have a MDG with 10,000 vertices, one element of the array will occupy 13 bits, while the state of the art would occupy 64 bits.

The genetic operators transform the population through successive generations, maintaining the *diversity* and *adaptation* properties from previous generations. In this work, we use the *one-point crossover operator*, which takes two binary strings (parents) and a random index as input, and produces two new binary strings (offspring) by swapping the parents' bits after that index. For example, if we have the parent binary strings $p_1 = 101010$ and $p_2 = 001111$, and an index $i = 1$, the offspring will be $c_1 = 101111$ and $c_2 = 001010$. We also used a *mutation operator* that can flip any bit of the individual's binary string at a specified probability. That is, given a mutation probability p and a binary string $s = b_1 b_2 \dots b_n$, we produce a random number $0 \leq r_i < 1$ for each bit b_i, flipping b_i in the cases where $r_i < p$. For example, if we have a binary string $s = 10011$, a mutation probability $p = 0.1$, and a sequence of random numbers $r = (0.9, 0.3, 0, 0.6, 0.5)$, the algorithm will produce a *mutant binary string* $s' = 10111$. In DCT we used the Xorshift algorithm in order to generate random numbers; which is a known fast algorithm [24]. To the best of our knowledge, no other SMC tool uses this algorithm.

As mentioned before, we setup the GA to optimize five objectives [16]:

- maximize *Modularization Quality* (MQ);
- maximize intra-edge dependencies;
- minimize inter-edge dependencies;
- maximize number of clusters;
- minimize the the difference between the maximum and minimum number of source-code entities in a cluster.

MQ was defined by Mitchell and Mancoridis [20] as follows:

$$MQ = \sum_{i=1}^{k} CF_i$$

$$CF_i = \begin{cases} \dfrac{\mu_i}{\mu_i + \frac{1}{2} \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \mu_i > 0 \\ 0 & \mu_i = 0. \end{cases}$$

In this equation, k is the number of clusters, μ_i is the number of edges within the i^{th} cluster, and $\varepsilon_{i,j}$ is the number of edges between the i^{th} and the j^{th} clusters.

With relation to the parameters, we chose their values similarly to Candela et al [17]. As such, given a software module graph $G = (V, E)$, and $n = |V|$, we defined the parameters population size (*PS*), maximum number of generations (*MG*), crossover probability (*CP*), and mutation probability (*MP*) as follows:

- $PS = \begin{cases} 2n & \text{if } n \leq 300 \\ n & \text{if } 300 < n \leq 3000 \\ n/2 & \text{if } 3000 < n \leq 10000 \\ n/4 & \text{if } n > 10000 \end{cases}$
- $MG = \begin{cases} 50n & \text{if } n \leq 300 \\ 20n & \text{if } 300 < n \leq 3000 \\ 5n & \text{if } 3000 < n \leq 10000 \\ n & \text{if } n > 10000 \end{cases}$
- $CP = \begin{cases} 0.8 & \text{if } n \leq 100 \\ 0.8 + 0.2(n - 100)/899 & \text{if } 100 < n < 1000 \\ 1 & \text{if } n \geq 1000 \end{cases}$
- $MP = \frac{16}{100\sqrt{n}}$

In summary, DCT is a *full-fledged* multi-objective SMC tool written in the Go programming language, which (a) uses NSGA-II as default implementation, (b) employs a simple CLI to ease the execution of experiments, and (c) explores two optimization techniques: binary strings to represent individuals and the Xorshift random number generator algorithm.

IV. Study Settings

This empirical assessment aims to evaluate the performance of DCT for clustering software systems of different sizes and complexities. We conducted two experiments. The first compares the performance of DCT against one software clustering tool that runs in a multi-objective mode (Heuristic

Design NSGA-II [23]). The second compares the performance of DCT and HD-NSGA-II against two software clustering tools that use a mono-objective strategy (Bunch [20] and Heuristic Design LNS [19]). Although many research studies on software clustering are available in the literature, most of these publications do not provide tools we can use.

We investigate the following questions in our study:

(a) How does the complexity of the systems affect DCT performance?
(b) How does the DCT performance compare to the performance of multi-objective tools (HD-NSGA-II)?
(c) How does the performance of multi-objective tools (DCT and HD-NSGA-II) compare to the performance of mono-objective tools (Bunch and HD LNS)?

The multi-objective algorithm of DCT must explore a solution space of exponential complexity. As such, answering the first research question allows us to understand if DCT could be used to cluster software systems in real settings. Answering the second research question, allows us to understand the performance of DCT in comparison with another NSGA-II implementation. Finally, regarding the last research question, it is still unclear to what extent the use of multi-objective algorithms compromise the performance of publicly available SMC tools. Answering the last research question allows us to better estimate the effect of using a multi-objective algorithm to cluster software systems.

We leveraged three metrics to answer these research questions: **TS** is the elapsed time in seconds to cluster each studied system; **MMC** is the Maximum Memory Consumption (in KB) necessary to cluster each studied system; and **MQ** is a metric for estimating the Modularization Quality of the clusters [2], [25].

We ran Bunch and HD LNS tools with their default settings. On the other hand, HD-NSGA-II was not concluding the process even on small systems. To reduce the number of evaluations, we set the parameters *population size* and *maximum number of generations* to $2n$ and $4n$, respectively, where n is the number of vertices on the MDG. The default values of these parameters are $10p$ and $200p$, where p is the *package count*. The definition of *package* used in HD-NSGA-II corresponds to a `package` in the Java programming language. Furthermore, we had to write a tool to convert MDGs to the proprietary file format used by HD-NSGA-II. Finally, we ported the HD-NSGA-II and HD-LNS implementations to Java libraries and implemented a command line tool to execute both of them.[4]. We hope that this decision could help other researchers to experiment with these tools.

We used the `time` Linux tool to compute the first two metrics. To calculate the MQ metric we considered the outcomes of the clustering tools (Bunch, Heuristic Design, and DCT). We used a dataset of 17 MDGs in our study. These MDGs come from a convenient sample population of open source systems we used in a previous research work [6]. These systems are from different domains and range from small to medium size systems (in terms of lines of code). Moreover, we set 48h as the maximum execution time. Table I presents some characteristics of these systems.

We executed our experiments using an Intel(R) Xeon(R) E-2124 CPU @ 3.30GHz with 32 GB of RAM, running a Linux Ubuntu distribution (18.04.4 LTS).

TABLE I
PROJECTS USED IN THE EMPIRICAL ASSESSMENTS

System	Modules	Deps.	KLOC	Commits
React Native Framework	190	1006	48	7842
Storm distributed realtime system	388	3249	213	7451
Bigbluebutton web conf. system	497	3661	82	13420
Minecraft Forge	501	3403	72	5498
CAS - Enterprise Single Sign On	513	1718	87	6268
Atmosphere Event Driven Framework	658	3523	41	5748
Druid analytics data store	668	2648	297	7452
Liquibase database source control	716	3981	77	5360
Kill Bill Billing & Payment Platform	767	5422	139	5361
Actor Messaging Platform	768	7452	157	8772
The ownCloud Android App	833	3389	36	5329
Hibernate Object-Relational Mapping	836	2935	628	7302
jOOQ SQL generator	851	4118	133	5022
LanguageTool Style/Grammar Checker	871	1931	75	19121
Bazel build system	965	3813	375	7258
H2O-3 - Machine Learning Platform	1586	27725	143	19336
Jitsi communicator	2557	6742	326	12420

V. RESULTS

In this section we highlight the main findings of our empirical study and provide answers to the research questions we introduced in Section IV.

A. How does the complexity of the systems affect the DCT performance?

To answer this research question, we first considered the complexity of the MDGs (in terms of number of modules) as a model of the log of the elapsed time (TS) to compute the clusters. That is, we expressed this model as $log(TS) \approx Modules$. Considering the adjusted R^2, this model indicates that we can explain 88.87% of the TS variance as an exponential function on the number of modules. This exponential model better explains this variance, in comparison to a quadratic model ($R^2 = 0.73$) and a linear model ($R^2 = 0.38$).

In practice, DCT finds a cluster solution to a small system with 190 modules and 48 KLOC in `00:01:57` (REACT NATIVE FRAMEWORK), to a medium size system with 767 modules and 139 KLOC in `00:23:49` (KILL BILL BILLING & PAYMENT PLATFORM), and to a large system with 2557 modules and 326 KLOC in `08:30:07` (JITSI COMMUNICATOR). That is, although we confirmed the exponential cost necessary for DCT to compute the clusters (as a function on the number of modules), we argue that it can still be used in practice, particularly for small and medium size systems. For larger systems, DCT might find a solution in an interval from hours to a few days (for extra large systems). So, regarding our first question, we found that:

> Our empirical assessment suggests that we can predict the time necessary for DCT compute a cluster using an

[4]https://github.com/project-draco/cms_runner

exponential formula on the system's number of modules.

In the longest scenario in our experiment, DCT found a cluster in 08:30:07 for a system with more than 2500 modules. We argue that this is still an reasonable time for running a SMC reengineering task on a large system using a multi-objective approach.

B. How does the DCT *performance compare to the performance of multi-objective tools (*HD-NSGA-II*)?*

Our goal to answer this question is to understand how DCT compares to another multi-objective SMC tool. Nonetheless, HD-NSGA-II only concluded the execution for seven (out of the 17 projects we consider in our study) within our maximum time threshold (48 hours). Considering only these seven projects, we realized a substantial benefit on the DCT speed-up, ranging from 2.13x to 221x (see Table II).

TABLE II
COMPARISON OF THE ELAPSED TIME TO GENERATE THE CLUSTERS (CONSIDERING THE MULTI-OBJECTIVE TOOLS DCT AND HD-NSGA-II).

System	DCT (TS)	HD-NSGA-II (TS)	Speed-up
React Native	117	249	2.13x
Storm	228	12448	54.60x
Big Blue Button	442	36264	82.05x
Minecraft Forge	579	54691	94.46x
CAS Single Sign On	335	39963	119.29x
Atmosphere	970	90954	93.77x
Druid	741	164428	221.90x

Regarding the other metrics (MMC and MQ), DCT improved memory consumption up to 2x (minimum gain of 1.8x — see Table III) and slightly decreased the MQ metrics in six out of the seven cases (see Table IV). Specifically, DCT presents a significant reduction on the time necessary to compute the clusters, in comparison to the HD-NSGA-II tool; however, we observed a slight reduction on the quality of the clusters. In the worst case, (Atmosphere project), DCT found a cluster with MQ = 69.64; while HD-NSGA-II found a cluster with MQ = 95.70. Altogether, we answer our second research question as follow:

Our assessment reveals that DCT scales better than HD-NSGA-II, finishing the clusterization process of the Druid tool in 741 seconds (while HD-NSGA-II needed 164 428 seconds). Considering larger projects, HD-NSGA-II did not finish the analysis within our maximum time threshold.

We observed that HD-NSGA-II clusters are slightly better than the clusters produced by DCT

*C. How does the performance of multi-objective tools (*DCT *and* HD-NSGA-II*) compares to the performance of mono-objective tools (*BUNCH *and* HD-LNS*)?*

The boxplots in Figure 2 show the performance of the tools (DCT, HD-NSGA-II, BUNCH, and HD-LNS), considering execution time (TS), memory consumption (MMC), and modularization quality (MQ). One could observe that multi-objective SMC implementations requires much more time to compute the clusters. In the worst scenario, DCT requires `00:40:48` while BUNCH required `00:00:04`, and HD-LNS requires `00:02:57` on the same comparison.

TABLE III
COMPARISON OF THE MEMORY USAGE GENERATING THE CLUSTERS (CONSIDERING THE MULTI-OBJECTIVE TOOLS DCT AND HD-NSGA-II).

System	DCT (MB)	HD-NSGA-II (MB)	Improv.
React Native	91	188	2.07
Storm	218	463	2.12
Bigbluebutton	282	511	1.81
Minecraft Forge	320	595	1.86
CAS - Enterprise Single Sign On	300	528	1.76
Atmosphere	416	741	1.78
Druid	396	713	1.80

TABLE IV
COMPARISON OF THE CLUSTERS' MQ (CONSIDERING THE MULTI-OBJECTIVE TOOLS DCT AND HD-NSGA-II).

System	DCT (MQ)	HD-NSGA-II (MQ)	Improv.
React-native	39.27	33.57	1.17
Storm	60.40	66.60	0.91
Big Blue Button	71.15	79.31	0.90
Minecraft Forge	87.94	92.76	0.95
CAS - Enterprise Single Sign On	92.77	99.07	0.94
Atmosphere	69.64	95.70	0.73
Druid	122.65	128.00	0.96
Average			0.94

Regarding memory consumption, the BUNCH tool achieved the best performance, with an average memory consumption of ∼126MB; while HD-LNS achieved an average consumption of ∼546MB. Considering the impact on the MQ metric, Figure 2 shows a (median) decreasing of 44% on the modularization quality of the clusters from multi-objective SMC tools. Differently, the mono-objective tools preserve the average quality of the clusters Altogether, we answer our second research question as follows.

The use of multi-objective SMC implementations brings a negative impact on both performance and modularization quality, in comparison with the multi-objective tools we used in our research. That is, on average, we found a central tendency of (a) increasing in 400x the time necessary to compute the cluster and (b) decreasing in 44% the modularization quality.

Comparing to HD-LNS, BUNCH brings significant improvements in two metrics (on average): time necessary to compute the clusters (up to 20x) and maximum memory consumption (up to 2x).

VI. FINAL REMARKS

In this paper we presented a new Software Module Clustering tool that address scalability issues. This property is particularly important for running experiments that uses SMC tools as part of its process. In this use case, normally is

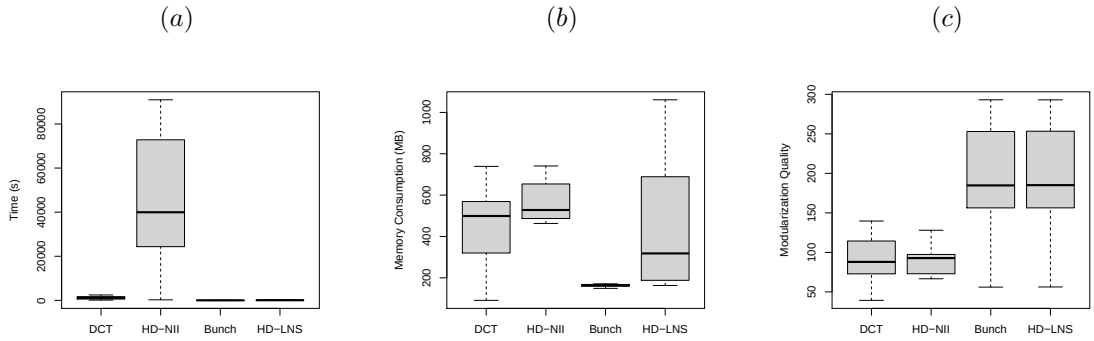

Fig. 2. Performance comparison of SMC tools. (a) Compares TS, (b) compares MMC, and (c) compares MQ. We removed the outliers in the boxplots.

required to repeatedly run several instances of the experiment. Accordingly, the tools' runtime efficiency is critical. We reported an comparison with other multi-objective SMC tool where we shown that our tool speeds up the elapsed time from 2 to 220 times, while using 2 times less memory and with a slightly decrease in MQ (6%). In the future, we will explore additional genetic algorithms, such as, NSGA-III, and pursue further optimizations.

REFERENCES

[1] S. S. Dahiya, J. K. Chhabra, and S. Kumar, "Use of genetic algorithm for software maintainability metrics' conditioning," in *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, 2007, pp. 87–92.

[2] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, March 2006.

[3] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, and M. Ribeiro, "Unveiling and reasoning about co-change dependencies," in *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, L. Fuentes, D. S. Batory, and K. Czarnecki, Eds. ACM, 2016, pp. 25–36. [Online]. Available: https://doi.org/10.1145/2889443.2889450

[4] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: the state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 35–39, 2003.

[5] D. Garlan, "Software architecture: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 91–101.

[6] M. C. de Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo, "Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings," *Journal of Systems and Software*, vol. 158, p. 110420, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219301943

[7] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[8] J. Huang, J. Liu, and X. Yao, "A multi-agent evolutionary algorithm for software module clustering problems," *Soft Computing*, vol. 21, no. 12, pp. 3415–3428, 2017.

[9] J. K. Chhabra *et al.*, "Many-objective artificial bee colony algorithm for large-scale software module clustering problem," *Soft Computing*, vol. 22, no. 19, pp. 6341–6361, 2018.

[10] A. Prajapati and J. K. Chhabra, "Madhs: Many-objective discrete harmony search to improve existing package design," *Computational Intelligence*, vol. 35, no. 1, pp. 98–123, 2019.

[11] J. Sun and B. Ling, "Software module clustering algorithm using probability selection," *Wuhan University Journal of Natural Sciences*, vol. 23, no. 2, pp. 93–102, 2018.

[12] M. Bishnoi and P. Singh, "Modularizing software systems using pso optimized hierarchical clustering," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, 2016, pp. 659–664.

[13] V. Singh, "Software module clustering using metaheuristic search techniques: A survey," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2016, pp. 2764–2767.

[14] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proceedings of the Sixth Working Conference on Reverse Engineering*, ser. WCRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 235–.

[15] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.

[16] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar. 2011.

[17] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 24:1–24:28, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2928268

[18] M. de Oliveira Barros, "Evaluating modularization quality as an extra objective in multiobjective software module clustering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 267–267.

[19] M. C. Monçores, A. C. Alvim, and M. O. Barros, "Large neighborhood search applied to the software module clustering problem," *Computers & Operations Research*, vol. 91, pp. 92–111, 2018.

[20] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[21] D. E. Goldberg, "E. 1989. genetic algorithms in search, optimization, and machine learning," *Reading: Addison-Wesley*, 1990.

[22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[23] M. d. O. Barros, "An analysis of the effects of composite objectives in multiobjective software module clustering," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012, pp. 1205–1212.

[24] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software, Articles*, vol. 8, no. 14, pp. 1–6, 2003. [Online]. Available: https://www.jstatsoft.org/v008/i14

[25] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*. IEEE Computer Society, 1999, p. 50. [Online]. Available: https://doi.org/10.1109/ICSM.1999.792498