Automated Software Engineering manuscript No.
(will be inserted by the editor)

Exact Search-space Size for the Refactoring Scheduling Problem

Rodrigo Morales · Francisco Chicano · Foutse Khomh · Giuliano Antoniol

Received: date / Accepted: date

Abstract Ouni et al. "Maintainability defects detection and correction: a multi-objective approach" proposed a search-based approach for generating optimal refactoring sequences. They estimated the size of the search space for the refactoring scheduling problem using a formulation that is incorrect; the search space is estimated to be too much larger than it is. We provide in this paper the exact expression for computing the number of possible refactoring sequences of a software system. This could be useful for researchers and practitioners interested in developing new approaches to automate refactoring.

Keywords Software maintenance · Search-based software engineering · Refactoring

1 Introduction

Refactoring is a software maintenance activity that aims to improve code design, while preserving behavior [9]. In the last decade, many works have reported that refactoring can reduce software complexity, improve developer comprehensibility and also improve memory efficiency and startup time [2, 14]. Hence, developers are advised to perform refactoring operations on a regular basis [3]. However, manual refactoring is a complicated task, as there could be more than one correct solution depending on the design attributes that one is interested in improving. Moreover, the order in which a set of candidate refactorings should be applied is uncertain, and can lead to different designs; some

Rodrigo Morales, Foutse Khomh, Giuliano Antoniol
DGIGL
Polytechnique Montréal, Canada
E-mail: rodrigomorales2@acm.org, foutse.khomh@polymtl.ca,antoniol@ieee.org

Francisco Chicano
Dept. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Andalucía Tech,
Spain E-mail: chicano@lcc.uma.es

refactorings can have sequential dependencies that requires a specific order to enable further refactorings, and other refactorings can be mutually exclusive (i.e., incompatible refactorings). Finding the right sequence of refactorings to apply on a software system is usually a hard task for which no polynomial-time algorithm is known. However, knowing the size of the search space (the possible refactoring sequences) helps to determine the best technique to solve the problem in a reasonable amount of time.

Recently, researchers have formulated the problem of refactoring as an optimization problem and suggested different metaheuristics techniques to solve it [4–8, 10–12]. The goal is to find a sequence of refactoring operations that most improves the design quality of a software system. The concept of "quality" here can be interpreted in many different ways. We can reduce the number of design defects, a.k.a., anti-patterns in the software, or improve some desirable quality attributes like maintainability and reusability. We assume that the potential refactoring operations that can be applied in the sequence are determined before the search starts. This is a big assumption, since new refactoring opportunities could arise as a consequence of a change in the code. However, the search for new refactoring opportunities after every change in the code is a costly operation. Thus, most (if not all) the works on automatic refactoring assume that there is a list of refactoring opportunities at the beginning of the search and the optimization algorithm simply selects which of them will be applied and their order.

Ouni et al. [10] provided a formula for the size of the search space of the automatic refactoring problem. The expression is $NS = (n!)^n$, where NS is the number of refactoring sequences (size of the search space), and n is the number of available refactoring operations (the list of refactoring operations available at the beginning of the search). This formula, however, is not correct. To illustrate this, let us suppose that $n = 3$, then the number of refactoring sequences should be $(3!)^3 = 6^3 = 216$, according to Ouni et al. [10]. A simple manual enumeration, shown in Table 1, proves that the number of refactoring sequences is, in fact, 16. Thus, we wish to find a closed form expression for the number of ordered subsets of a set of size n. The On-Line Encyclopedia of Integer Sequences, sequence A000522 [1] gives values of S for low values of n. In the rest of this paper, we derive asymptotic and exact closed-form formula for these numbers. The asymptotic result seems to have been known to Ramanujan (Notebook II) [13], while the exact result has been discussed in the community more recently. We present them with proof here for completeness and ease of reference for the reader.

2 Asymptotic behaviour of the number of refactoring sequences

In order to provide an exact expression for the number of refactoring sequences, we need to formally describe how to count the number of possible combinations from a set of refactoring operations.

Table 1 Number of possible refactoring sequences for the set of refactoring operations {r1, r2, r3}.

1.	None	9.	r3, r1
2.	r1	10.	r3, r2
3.	r2	11.	r1, r2, r3
4.	r3	12.	r1, r3, r2
5.	r1, r2	13.	r2, r1, r3
6.	r1, r3	14.	r2, r3, r1
7.	r2, r1	15.	r3, r2, r1
8.	r2, r3	16.	r3, r1, r2

Let us denote with R the set of refactoring opportunities and with $n = |R|$ its cardinality. The number of refactoring sequences can be found by (1) selecting a subset of refactorings from R and, (2) finding all the possible permutations of the selected subset of refactorings. Let S be the set of sequences of refactorings in R. Following the previous idea, the cardinality of S can be computed as:

$$\begin{aligned} |S| &= \sum_{R' \subseteq R} |R'|! = \sum_{k=0}^{n} \binom{n}{k} k! = \sum_{k=0}^{n} \frac{n!}{(n-k)!k!} k! = \sum_{k=0}^{n} \frac{n!}{(n-k)!} \\ &= n! \sum_{k=0}^{n} \frac{1}{(n-k)!} = n! \sum_{k=0}^{n} \frac{1}{k!} \leq n! \cdot e, \end{aligned} \tag{1}$$

where the last inequality follows from the Taylor expansion of e^x. That is, $e^x = \sum_{k=0}^{\infty} x^k/k!$, and if we evaluate the previous expression in $x = 1$ we have $e = \sum_{k=0}^{\infty} 1/k!$.

As a consequence, we have $|S| \in O(n!)$, that is, the number of refactoring sequences (assuming there are no destructive conflicts among them) is $O(n!)$. But, on the other hand, $|S| \geq n!$, because we can select all the refactoring operations for the sequence and build at least $n!$ different sequences. Thus, $|S| \in \Theta(n!)$. That is, the size of the set S grows as $n!$. This is already much smaller than Ouni et al.'s result, by a power of n. The asymptotic behavior obtained in this section could be enough in many contexts to characterize the growth of the search space. In the next section, we provide an exact expression, valid for all the values of n.

3 The exact expression

Let us call E_n the difference between the value of $|S|$ and $e \cdot n!$, that is: $E_n = e \cdot n! - |S|$. Then, according to (1) we have:

$$E_n = e \cdot n! - |S| = n! \sum_{k=n+1}^{\infty} \frac{1}{k!}, \tag{2}$$

which can be considered a series in n.

Lemma 1 *The series E_n fulfills the following recurrence:*

$$E_{n+1} = (n+1)E_n - 1, \tag{3}$$

where $E_0 = e - 1$.

Proof The proof is a simply algebraic manipulation:

$$\begin{aligned} E_{n+1} &= (n+1)! \sum_{k=n+2}^{\infty} \frac{1}{k!} = (n+1)! \sum_{k=n+1}^{\infty} \frac{1}{k!} - 1 \\ &= (n+1) \left(n! \sum_{k=n+1}^{\infty} \frac{1}{k!} \right) - 1 = (n+1)E_n - 1 \end{aligned}$$

Lemma 2 *The following equality holds for E_n:*

$$E_n = e \int_0^1 t^n e^{-t} dt. \tag{4}$$

Proof We will prove this simply by checking that the integral fulfills the recurrence equation (3) with the initial value $E_0 = e - 1$. Let us start with the explicit expression for E_{n+1} and let us integrate by parts:

$$\begin{aligned} E_{n+1} = e \int_0^1 t^{n+1} e^{-t} dt &= \left[-e^{-t+1} t^{n+1} \right]_0^1 + e \int_0^1 (n+1) t^n e^{-t} dt \\ &= -1 + (n+1) e \int_0^1 t^n e^{-t} dt = -1 + (n+1) E_n. \end{aligned}$$

Now we can easily see that:

$$E_0 = e \int_0^1 t^0 e^{-t} dt = e \int_0^1 e^{-t} dt = \left[-e^{-t+1} \right]_0^1 = -1 + e.$$

Since both, the initial value and the recurrence equation holds for (4), and the recurrence equation is order 1, then the equality holds for all $n \geq 0$.

Theorem 1 *The exact expression for $|S|$ is $\lfloor e \cdot n! \rfloor$ for $n \geq 1$ and $|S| = 1$ for $n = 0$.*

Proof The case $n = 0$ is trivial (if we don't have refactoring operations, there is only one way to build a sequence of refactoring operations).

For $n \geq 1$, first we notice that E_n is a strictly decreasing series for $n \geq 0$. The reason is that for any $t \in (0,1)$ we have that the integrand in the expression for E_n, $t^n e^{-t}$ is greater than the integrand for E_{n+1}, since $t^n > t^{n+1}$. Then, we have, $E_{n+1} < E_n$, for $n \geq 0$. Using the recurrence equation we can easily see that $E_1 = e - 2 < 1$, and thus, $E_n < 1$ for all $n \geq 1$.

In order to finish the proof, we just recall that $E_n = e \cdot n! - |S|$, and we can write $e \cdot n! - |S| < 1$ for $n \geq 1$. We already had $|S| \leq e \cdot n!$. Combining both, we have:

$$|S| \leq e \cdot n! < |S| + 1, \tag{5}$$

and the result follows.

Applying the formula to the example presented in Section 1 with $n = 3$ we have $|S| = \lfloor e \cdot 3! \rfloor = \lfloor 16.3097 \rfloor = 16$, which is the correct answer, *c.f.,* Table 1.

Note that we are assuming that applying the permutation of a subset of refactoring operations always leads to a different software design. For example, consider the sequences 5 and 7 from Table 1, composed of refactorings r_1 and r_2. However, in case of independent refactorings, i.e., refactorings that target code entities which are not related at all, this assumption may not be true. Hence, there is a chance to reduce even more the search space if we remove these permutations, after a previous analysis of refactoring interdependences. Thus, the value obtained after applying the proposed formula can be freely used as an upper bound of the maximum size of the search space, as long as we assume that applying a refactoring sequence does not create new refactoring opportunities that were not in the original set. If this can happen, the number of possible refactorings can be larger than our upper bound. However, when working in an iterative mode, one expects that a software maintainer would repeat the process of detecting refactoring candidates until: (1) there are no more refactorings to apply, or (2) the software maintainer is satisfied with the achieved design quality. Hence this does not constitute a limitation for the application of the proposed formula.

4 Conclusion

We provided an exact and easy to compute formula for the number of refactoring sequences that can be scheduled from a set of refactoring candidates for a software system. The formula assumes that distinct refactoring sequences result in distinct designs. If this is not the case, the search space is smaller and the provided expression should be interpreted as an upper bound of the search space as long as we assume that applying a refactoring sequence does not lead to new refactoring opportunities in addition to the ones present in the original set.

The correct expression for the number of refactoring sequences can be used to decide which is the most suitable search algorithm to be used to generate the best refactoring sequence. In particular, the expression can be used to determine if exact methods are able to find the optimal refactoring sequence in a reasonable time.

Acknowledgements This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and *Consejo Nacional de Ciencia y Tecnologia*, México (CONACyT).

References

1. The on-line encyclopedia of integer sequences, sequence no. A000522. http://oeis.org/A000522. Accessed: 2017-01-16

2. Bois, B.D., Demeyer, S., Verelst, J., Mens, T., Temmerman, M.: Does god class decomposition affect comprehensibility? In: IASTED Conf. on Software Engineering, pp. 346–355 (2006)
3. Fowler, M.: Refactoring: improving the design of existing code. Pearson Education India (1999)
4. Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pp. 1106–1113. ACM
5. Moghadam, I.H., Cinneide, M.O.: Code-imp: A tool for automated search-based refactoring. In: Proceedings of the 4th Workshop on Refactoring Tools, pp. 41–44. IEEE Computer Society (2011)
6. Morales, R., Sabane, A., Musavi, P., Khomh, F., Chicano, F., Antoniol, G.: Finding the best compromise between design quality and testing effort during refactoring. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 24–35 (2016)
7. Morales, R., Soh, Z., Khomh, F., Antoniol, G., Chicano, F.: On the use of developers' context for automatic refactoring of software anti-patterns. Journal of Systems and Software (2016). To appear
8. O'Keeffe, M., Cinneide, M.O.: Search-based software maintenance. In: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, pp. 10 pp.–260 (2006)
9. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
10. Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M.: Maintainability defects detection and correction: a multi-objective approach. Automated Software Engineering **20**(1), 47–79 (2013)
11. Ouni, A., Kessentini, M., Sahraoui, H., Hamdi, M.S.: Search-based refactoring: Towards semantics preservation. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 347–356. IEEE (2012)
12. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S.: Improving multi-objective code-smells correction using development history. Journal of Systems and Software **105**(0), 18 – 39 (2015)
13. Ramanujan, S.: Notebooks (2 volumes). Tata Institute of Fundamental Research, Bombay **27**, 96,816–3236 (1957)
14. van Rompaey, B., Du Bois, B., Demeyer, S., Pleunis, J., Putman, R., Meijfroidt, K., Dueas, J.C., Garcia, B.: Serious: Software evolution, refactoring, improvement of operational and usable systems. In: Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conf. On, pp. 277–280 (2009)