CONCENTRATION OF KNOWLEDGE IN SOFTWARE PROJECTS: AN EMPIRICAL ASSESSMENT

MÍVIAN MARQUES FERREIRA

CONCENTRATION OF KNOWLEDGE IN SOFTWARE PROJECTS: AN EMPIRICAL ASSESSMENT

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Tulio de Oliveira Valente
Coorientadora: Kecia Aline Marques Ferreira

Belo Horizonte

Junho de 2017

MÍVIAN MARQUES FERREIRA

CONCENTRATION OF KNOWLEDGE IN SOFTWARE PROJECTS: AN EMPIRICAL ASSESSMENT

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Tulio de Oliveira Valente
Co-Advisor: Kecia Aline Marques Ferreira

Belo Horizonte

June 2017

© 2017, Mívian Marques Ferreira.
Todos os direitos reservados.

Marques Ferreira, Mívian

D1234p Concentration of Knowledge in Software Projects: An Empirical Assessment / Mívian Marques Ferreira. — Belo Horizonte, 2017
xxv, 67 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Marco Tulio de Oliveira Valente
Coorientadora: Kecia Aline Marques Ferreira

1. Computação — Teses. 2. Engenharia de Software — Teses. I. Orientador. II. Título.

CDU 519.6*82.10

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Concentration of knowledge in software projects: an empirical assessment

MÍVIAN MARQUES FERREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. KÉCIA ALINE MARQUES FERREIRA - Coorientadora
Departamento de Computação - CEFET-MG

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

PROF. RICARDO TERRA NUNES BUENO VILLELA
Departamento de Ciência da Computação - UFLA

Belo Horizonte, 05 de junho de 2017.

To my mother, Alzira. A brilliant woman.

Acknowledgments

I have so much to thank! I could never complete this work without the support I received during these two long years. Special thanks:

To my family for always impelling me to seek new paths, to support me at all times and in every possible way. This achievement is also yours.

To Rubico, for love and affection, for understanding my absences, taking my own dream as his, supporting me, and making me happy.

To my advisors, Marco Tulio and Kecia. Thank you for your patience and commitment to my work.

To the friends of aSERG, for showing me good coffee and making this journey much more joyful.

To my friends of LBS lab. Thank you for welcoming me. Guys, you are amazing! This journey would be completely bland without you!

To CAPES, for funding the execution of this work.

To DCC/PPGCC, for all support provided.

To all those who were part of this moment, thank you very much!

"No man is an island,
Entire of itself.
Every man is a piece of the continent,
A part of the main."
(John Donne)

Resumo

Desenvolvimento de software é uma indústria altamente baseada em conhecimento. Esse fato faz com que a concentração de conhecimento seja um risco grande para projetos de software. Sendo assim, torna-se crucial obter informações sobre como o conhecimento acerca do código-fonte está distribuído entre os membros de uma equipe. *Truck Factor* é um conceito proposto por profissionais para indicar o número mínimo de desenvolvedores que devem se ausentar para que a continuidade de um projeto seja impossibilitada. Logo, essa métrica indica a concentração de conhecimento e os desenvolvedores chave de um projeto. Devido à importância dessas informações para gerentes de projeto, foram propostos alguns algoritmos na literatura para calcular o *Truck Factor* automaticamente, por meio de dados de atividades de manutenção extraídos de sistemas de controle de versão. No entanto, ainda faltam estudos que avaliam os resultados desses algoritmos. Esta dissertação de mestrado tem por objetivo avaliar distribuições de conhecimento em projetos de software por meio de uma análise profunda da métrica *Truck Factor*. Para tanto, foram realizados três estudos empíricos. No primeiro estudo, foram validados os resultados de *Truck Factor* produzidos por três algoritmos propostos na literatura. Foi constituído um oráculo de *Truck Factor* por meio de um *survey* com equipes de 35 projetos de código aberto. Os resultados revelaram o melhor algoritmo para cálculo de *Truck Factor*. No segundo estudo, buscou-se identificar a relação entre *Truck Factor* e *Core Developers*. Os resultados obtidos indicam que os desenvolvedores pertencentes ao conjunto *Truck Factor* são, na maior parte dos casos, um subconjunto de *Core Developers*. No último estudo, analisou-se a relação entre *Truck Factor* e arquitetura de software. Como resultado, mostrou-se a adoção da importância das classes como um fator a ser analisado em estimativas de *Truck Factors*.

Palavras-chave: *Truck Factor*; Propriedade de código; *Core Developers*; Rotatividade de desenvolvedores; Mineração de repositórios de software.

Abstract

Software development is a knowledge-intensive industry. For this reason, concentration of knowledge in software projects tends to be very risky. This risk makes crucial to obtain reliable data on how the source code knowledge is distributed among team members. Truck Factor is a concept proposed by practitioners to indicate the minimal number of developers that have to be hit by a truck (or leave) before a project is incapacitated. Therefore, it is a measure that reveals the concentration of knowledge and the key developers in a project. Due to the importance of this concept to project managers, algorithms have been proposed to automatically compute Truck Factors, using maintenance activity data extracted from version control systems. However, we still lack large studies that assess the results of Truck Factor algorithms. This master dissertation aims to assess the distribution of knowledge in software projects through an in-depth analysis of the metric Truck Factor. To achieve this goal, we conducted three empirical studies. In the first study, we validate the results produced by three recent algorithms proposed in the literature to estimate Truck Factors. We built an oracle of Truck Factor, gathered via a survey with 35 open-source project teams. The results reveal the best algorithm proposed in the literature so far to estimate Truck Factor. In the second study, we conducted a comparison between the concepts of Truck Factor and Core Developers, a related concept proposed in the literature. We aim to reveal the relationship between these concepts. Our results indicate that Truck Factor developers are in most cases a subset of Core Developers. In our last study, we analyze the relation between Truck Factor and software architectures. As a result, we recommend that measures of Truck Factor should consider the importance of the classes in a software project.

Palavras-chave: Truck Factor; Code ownership; Core Developers; Developer turnover; Mining software repositories.

List of Figures

4.1 AVL Dataset: distribution of number of developers, commits, and files (Avelino et al. [2016]) 24

5.1 Distribution of the system's Truck Factor 31
5.2 Absolute error results 32
5.3 Precision 35
5.4 Recall 35
5.5 F-measure 36
5.6 Number of systems with $error = 0$ and $|error| \leq 1$, after varying AVL's threshold on abandoned files. 37
5.7 Dispersion of RIG errors, considering 30 runs. The highlighted value (in red) is the median error. 38
5.8 Number of systems with $error = 0$ and $|error| \leq 1$, after varying RIG's threshold on the number of tested samples. 39

6.1 Error 46
6.2 Precision 48
6.3 Recall 48
6.4 F-measure 49

List of Tables

4.1 AVL dataset (Avelino et al. [2016]) 23

5.1 Oracle of TF results. Systems in the first table's section are reused from Avelino et al. [2016]. Age is measured in years. 31
5.2 Error measures for outlier systems 33
5.3 Percentage of results with *error* = 0 per group of systems. Size is the number of systems in each group. 33
5.4 Number of systems with *error* = 0 and *error* $\leq$ 1, when CST is configured to use LCTA and MCEC metrics. 40

6.1 Percentage of results with *error* = 0 per group of systems. Size is the number of systems in each group. 47
6.2 Summary of intersection results. 51

7.1 *Android Annotations* results 57
7.2 *Titan* results 57
7.3 *Dropwizard* results 58
7.4 *Glide* results 58

Contents

Acknowledgments **xi**

Resumo **xv**

Abstract **xvii**

List of Figures **xix**

List of Tables **xxi**

1 Introduction **1**
1.1 Aims 2
1.2 Contributions 4
1.3 Publications 4
1.4 Outline of the Dissertation 5

2 Background **7**
2.1 Truck Factor 7
2.1.1 Algorithms for Estimating Truck Factor 8
2.2 Core Developers 12
2.2.1 Heuristics for Identifying Core Developers 12
2.3 Model Software Architecture 14
2.3.1 Little House 14
2.3.2 Previous Evaluation of the Little House Model 15

3 Related Work **17**
3.1 Authorship and Ownership 17
3.2 Developer Categorization 18
3.3 Experimental Studies on Source Code Knowledge 18
3.4 Experimental Studies on Truck Factor 19

3.5 Final Remarks . . . 20

4 Study Design **21**
4.1 Research Questions . . . 21
4.2 Dataset . . . 23
4.2.1 AVL Dataset . . . 23
4.2.2 Oracle of Truck Factors . . . 24
4.3 Handling Third-Party Libraries and Aliases . . . 26
4.4 Implementing, Configuring, and Executing the Algorithms . . . 26

5 Comparison of Algorithms for Computing Truck Factor **29**
5.1 Motivation . . . 29
5.2 Oracle of Truck Factors . . . 30
5.3 Results . . . 32
5.4 Discussion . . . 40
5.4.1 What is the runtime performance of each algorithm? . . . 40
5.4.2 Why and when the algorithms fail? . . . 40
5.4.3 Are Truck Factor scenarios realistic? . . . 41
5.5 Threats to Validity . . . 42
5.6 Final Remarks . . . 43

6 Truck Factor vs. Core Developers **45**
6.1 Motivation . . . 45
6.2 Results . . . 46
6.3 Discussion . . . 52
6.4 Final Remarks . . . 53

7 Knowledge Distribution in Software Architectures **55**
7.1 Study Design . . . 55
7.2 Results . . . 56
7.2.1 Android Annotations . . . 56
7.2.2 Titan . . . 57
7.2.3 Dropwizard and Glide . . . 57
7.3 Discussion . . . 58
7.4 Threats to Validity . . . 59
7.5 Final Remarks . . . 59

8 Conclusion **61**

Bibliography **65**

Chapter 1

Introduction

Software systems have a major impact on the daily lives of our society, and we are observing everyday the rise of companies totally centered on software and, by consequence, on their developers. By this own nature, software development is a knowledge-intensive industry, which makes developers one of the most important resources of software organizations.

In this context, developer turnover is a serious risk to software projects [Mockus, 2010; Williams and Kessler, 2003]. To mitigate this risk, project managers should measure and monitor the concentration of knowledge in specific team members. An indicator of this concentration is a measure known as Truck Factor (TF), which is defined as the minimal number of developers that have to be hit by a truck (or leave the team) in so that the project will be in trouble [Williams and Kessler, 2003; Zazworka et al., 2010; Ricca and Marchetto, 2010; Ricca et al., 2011; Mens, 2016; Avelino et al., 2016]. This measure is also known as Bus Factor or Lottery Number. Essentially, a low Truck Factor means that the knowledge about the project is concentrated in few team members and, therefore, the project faces a serious risk of discontinuation in case these developers leave. On the other hand, a high Truck Factor means that many developers are contributing to the project in similar terms. Then, a high Truck Factors may be the result of collective code-ownership practices, commonly advocated by agile software development methodologies [Williams and Kessler, 2003; Beck and Andres, 2004].

In the case of open-source software projects, Truck Factor has additional relevance, since these projects are commonly maintained by volunteer developers. These developers do not have legal contracts with the project and usually do not have financial benefits, which makes turnover risks higher than in commercial software development. An example of a "Truck Factor episode" faced by an open-source project is illustrated

by the following mail, recently posted to the FindBugs[1] mailing list.[2]

> *I'm really sorry to say, but FindBugs project in its current form is dead. [...] It looks like the project leader is not interested in the project anymore, and we can't reach him. [...] We requested his help for the project many times (via direct mails, postings to the list and to the GitHub issues) but haven't received any sign of life from him since a year.*

Given the relevance of Truck Factor, some algorithms have been proposed to estimate this measure. Nevertheless, we still lack studies to assess the differences between these algorithms and identify if the results presented by each algorithm are compatible with real scenarios of software project development. There is also the concept of Core Developers, which is also close to Truck Factor. Both concepts aim to identify the most important developers of a software development project. However, there are no works, to the best of our knowledge, that carried out empirical investigation to reveal the relationship between these concepts.

1.1 Aims

This master dissertation aims to assess the distribution of knowledge in software projects through an in-depth analysis of the Truck Factor metric. Therefore, we conducted three empirical studies, described as follows.

Study #1: Comparison of Algorithms for Computing Truck Factor

In our first study, we validate the results produced by three recent algorithms proposed in the literature to estimate Truck Factor: AVL Algorithm [Avelino et al., 2016], RIG Algorithm [Rigby et al., 2016], and CST Algorithm [Cosentino et al., 2015]. To achieve this goal, we first built an oracle of Truck Factor by means of a survey with the developers of 35 open-source systems hosted on GitHub. Besides, we compare these results and discuss the main benefits and limitations of each algorithm. To support this study, we investigate the following research questions:

- **RQ1. How accurate are the results provided by each algorithm?**

  In this research question, we aim to identify how close a Truck Factor estimated by an algorithm (AVL, CST or RIG) is to the true value reported in the constructed oracle.

[1] Findbugs is a popular open-source bug finding tool for Java systems.
[2] https://mailman.cs.umd.edu/pipermail/findbugs-discuss/2016-November/004321.html

- **RQ2. How accurate is the identification of Truck Factor developers by each algorithm?**

  This research question investigates how often occur the cases where an algorithm correctly estimates the Truck Factor, but not the list of key authors.

- **RQ3. What is the impact of different thresholds and configurations in the results of each algorithm?**

  In this research question, we explore the impact of different thresholds on the results of the algorithms. We ranged the following thresholds of the algorithms: percentage of abandoned files, in AVL Algorithm; number of random samples, in RIG Algorithm; and authorship metrics, in CST Algorithm.

Study #2: Truck Factor vs. Core Developers

In our second study, we conducted a comparison between the concepts of Truck Factor and Core Developers. We seek to reveal the relationship between these concepts, since both aim to identify developers who play a central role in software development projects. Therefore, we compare the results of two heuristics for Core Developers estimation with the Truck Factor values provided by the oracle that we build. To guide this comparison, we investigate the following research questions:

- **RQ4. How accurate are the results provided by each heuristic?**

  This research question aims to identify how close is the number of Core Developers, estimated by the heuristics, to the Truck Factors reported in the oracle.

- **RQ5. How accurate is the identification of Truck Factor developers by each heuristic?**

  This research question aims to identify how often the Core Developers indicated by the studied heuristics are the same authors reported in the oracle of Truck Factor.

- **RQ6. What is the correlation between Truck Factor and the Core Developers sets?**

  In this research question, we investigate the correlation between the Truck Factor and the Core Developers, by analyzing the intersection of these sets.

Study #3: Knowledge Distribution in Software Architectures

In our last study, we investigate whether the set of developers indicated by a Truck Factor algorithm are those who have knowledge on the most complex components of a system architecture. For this purpose, the architecture was represented

by mean of the Little House [Ferreira, 2011], a generic architectural model, previously proposed in the literature. To support this study, we investigate the following research question:

- **RQ7. Do Truck Factor developers have knowledge on the most complex components of a system?**

  This research question aims to investigate whether the Truck Factor developers are those who actually have knowledge on the most complex architectural components of the software system.

1.2 Contributions

We highlight the following contributions of the studies presented in this master dissertation:

(i) A Truck Factor oracle obtained through a survey with the leading developers of 35 open-source projects hosted on GitHub;

(ii) A comprehensive study that identifies the best algorithm in the literature to estimating Truck Factor;

(iii) A second study that compares Truck Factor with Core Developers; and

(iv) A third study that interprets Truck Factor considering software architecture aspects.

1.3 Publications

This master dissertation generated the following publications and therefore contains material from them:

- Mívian Ferreira, Marco Tulio Valente, and Kecia Ferreira. A Comparison of Three Algorithms for Computing Truck Factors. In 25th International Conference on Program Comprehension (ICPC), pages 1-11, 2017. (*Qualis* A2)

- Mívian Ferreira, Guilherme Avelino; Marco Tulio Valente, and Kecia Ferreira. A Comparative Study of Algorithms for Estimating Truck Factor. X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), p. 91-100, 2016. (*Qualis* B4)

- Mívian Ferreira, Kecia Ferreira, and Marco Tulio Valente. Distribuição de Conhecimento de Código em Times de Desenvolvimento – uma Análise Arquitetural. IV Workshop Brasileiro em Visualização, Evolução e Manutenção de Software (VEM), p. 89-96, 2016.

1.4 Outline of the Dissertation

The remaining of this dissertation is organized as follows.

- **Chapter 2** covers the main concepts related to this master dissertation. We introduce the concepts of Truck Factor and Core Developers, and the main algorithms defined in the literature to estimate them. We also present the Little House model, used in this work to represent software architectures.
- **Chapter 3** presents related work. First, we present the authorship and ownership concepts which are related to Truck Factor. We also discuss the categorization of developers according to their contributions to the source code of software projects. Finally, we present works that apply Truck Factor in experimental studies.
- **Chapter 4** describes the method used to conduct the studies presented in this work. Fist, we present our dataset and the steps for cleaning unnecessary files. Second, we describe the configuration, implementation, and execution of the algorithms compared in this work. Further, we present the research questions.
- **Chapter 5** presents a comparative study of three algorithms to calculate Truck Factor. First, we compare the algorithms accuracy, analyzing the error measure. Second, we compare the algorithms using the precision, recall, and F-measure measures. We also analyze the algorithm results when their thresholds are modified.
- **Chapter 6** describes a comparative study between the concepts of Truck Factor and Core Developers. Heuristics for estimating Core Developers are analyzed taking into account their accuracy in relation to the oracle of Truck Factor. We then sought to identify the relation between Truck Factor and Core Developers.
- **Chapter 7** presents an empirical study that investigates whether the set of developers indicated by a Truck Factor algorithm are those who have knowledge on the most complex components of the system.
- **Chapter 8** presents the conclusion of this dissertation, outlining its main contributions and future work.

Chapter 2

Background

This chapter presents the main concepts and algorithms previously defined in the literature and used in this work. First, we present the Truck Factor concept and the main algorithms proposed in the literature to compute this metric (Section 2.1). Second, we describe the Core Developer concept and two heuristics defined in the literature to calculate the Core Developers of a system (Section 2.2). Finally, we present the software architecture model used in this work (Section 2.3).

2.1 Truck Factor

Truck factor is a software metric that aims to identify the distribution of knowledge among development team members. Also known as Bus Factor, Bus Number, Truck Number or Lottery Factor, the metric is defined as "*the minimum number of people on your team who must be hit by a truck, so that your project gets into serious trouble*" [Bowler, 2005]. Naturally, being "hit by a truck" is a metaphor used to represent the unexpected absence of a team member. As reported by Ricca et al. [2011], in addition to identifying the distribution of knowledge among team members, Truck Factor may also be used to identify other possible risks to the project by noting how much the project is dependent on certain developers.

High values for Truck Factor indicates that there is a homogeneous distribution of knowledge among team members and that if a team member leaves the project may suffer impact, but it will not be dramatic. On the other hand, low values for this metric indicate a concentration of knowledge on individual members of the team. In this case, if one of those members leaves the project, it may make difficult or impossible to evolve the project.

2.1.1 Algorithms for Estimating Truck Factor

In this section, we present the main algorithms proposed in the literature to estimate Truck Factor: ZWK Algorithm, proposed by Zazworka et al. [2010], CST Algorithm proposed by Cosentino et al. [2015], AVL Algorithm proposed by Avelino et al. [2016], and RIG Algorithm proposed by Rigby et al. [2016]. These algorithms are evaluated and compared in this master dissertation.

2.1.1.1 ZWK Algorithm

Zazworka et al. [2010] presentedtThe first description of an algorithm for estimating Truck Factor. They used the metric to point out differences between the distribution of knowledge in projects that make use of Extreme Programming (XP) and projects that use other software development methodologies. Zazworka et al. [2010] describe the logic applied in the experiments execution presented in their work. However, they do not provide a concrete algorithm structure. Ricca et al. [2011] provide the first formalization of this algorithm, as shown in Algorithm 1.

Algorithm 1: ZWK Algorithm

```
   Input: coverage, devs (List of Developers), files (List of Files)
   Output: TF (Truck Factor)
 1 begin
 2     for j = 1 to devs.size() do
 3         minimalCoverage ← 100;
 4         for each comb ∈ Combination(j, devs) do
 5             covered ← 0;
 6             for each file ∈ files do
 7                 if getFileDevs(file)-comb ≠ null then
 8                     covered ← covered + 1;
 9                 end
10             end
11             currentCoverage = covered/files.size() * 100;
12             if currentCoverage ≤ minimalCoverage then
13                 minimalCoverage ← currentCoverage;
14             end
15         end
16         minimalCoverageArray[j] ← minimalCoverage;
17         if minimalCoverageArray[j] < coverage then
18             break;
19         end
20     end
21     return j;
22 end
```

ZWK Algorithm receives as input the *coverage*, i.e., percentage of files that should be abandoned to configure a disaster scenario; the list of the system developers (*devs*), i.e., developers who have executed at least one commit in the system; and the list of system files (*files*). Given the list of developers, the algorithm generates the combination of n developers taken j at the time, where $1 \leq j \leq$ n (line 2). For each combination, it calculates whether the number of files that would remain with authors (*covered*) considering the exit of the developers present in the set (lines 4-10). Then, it verifies if the current coverage (*currentCoverage*) is the minimal coverage for the combination size (j) (lines 11-14). The Truck Factor is given by the number of developers present in the first combination whose coverage (*minimalCoverage*) is less than the value of variable *coverage* (line 17-19).

2.1.1.2 AVL Algorithm

Avelino et al. [2016] proposed the AVL algorithm that relies on the Degree-of-Authorship (DOA) measure to define the *authors*, i.e., the key developers, of each file in a system [Fritz et al., 2010, 2014]. DOA values are computed from commit histories as follows: the creation of a file f by a developer d initializes the value of $DOA(d, f)$; further commits on f by d increase $DOA(d, f)$; finally, commits by other developers decrease $DOA(d, f)$. The weights used to increase/decrease the DOA values were defined after empirical experiments performed with developers of two proprietary software [Fritz et al., 2014]. As the last step, DOA values are normalized per file; the developer with the highest DOA in a file f has its normalized DOA equal to 1. A developer is considered an author of a file if its normalized DOA is greater than 0.75. This threshold was defined after an empirical experiment when the authors of AVL Algorithm manually validated DOA results produced for a sample of 120 source code files from six open-source systems [Avelino et al., 2016].

AVL algorithm is presented in Algorithm 2. It receives as input the list of top-authors of a system S, i.e., the list of authors in S ordered by the number of files they have authorship on. This list should be previously computed using the DOA measure, as described in the previous paragraph. AVL algorithm continually simulates the removal of the top-author from the system. After each top-author removal (line 7), the current TF estimate is incremented by one (line 8), and the removed author is added to the set of TF developers (line 9). The algorithm terminates when more than 50% of the files become abandoned (line 10-12). A file is considered *abandoned* when all its authors have been removed from the system.

Algorithm 2: AVL Algorithm

```
   Input: TA (list of top-authors of a system S)
   Output: tf (Truck Factor), TFSet (TF developers)
 1 begin
 2 |   tf ← 0;
 3 |   TFSet ← ∅;
 4 |   TA ← list of top-authors;
 5 |   while TA ≠ ∅ do
 6 |   |   dev ← head(TA);
 7 |   |   remove-author(dev);
 8 |   |   tf++;
 9 |   |   TFSet ← TFSet + dev;
10 |   |   if rate-abandoned-files() ≥ 0.5 then
11 |   |   |   break;
12 |   |   end
13 |   |   TA ← TA − dev;
14 |   end
15 |   return tf, TFSet;
16 end
```

2.1.1.3 RIG Algorithm

[Rigby et al., 2016] proposed the RIG algorithm, as part of piece of work in which the authors assessed and quantified the susceptibility of software projects to developer turnover. The algorithm uses a blame-based approach to computing code authorship instead of a commit-based approach, as in the AVL algorithm. As implemented in git-based version control systems, the *blame* feature indicates the developer who last changed each line in a file. RIG algorithm defines that a line of code is *abandoned* when the `git-blame` command attributes the line to a developer who has left the project. The algorithm also defines that a file is *abandoned* when at least 90% of its lines are abandoned. RIG authors claim they use this high threshold to exclude developers with trivial contributions to a file.

RIG algorithm is presented in Algorithm 3. The algorithm works by simulating several Truck Factor scenarios. First, it varies the group size (g), of developers who leave from 1 to 200 developers (line 2). Then, the algorithm randomly selects groups of developers to leave, each one with size g (line 4). This selection procedure is performed 1,000 times, for each group size (line 3). The algorithm also computes the likelihood of each disaster scenario, i.e., the departure of the selected group of developers (line 5), using a statistical technique initially proposed to manage financial risks. We omit details on this specific likelihood computation (line 5) because our interest is the TF determination, despite the probability of its occurrence. We refer the interested reader

to the original paper [Rigby et al., 2016]. To make fair the comparison of RIG with AVL, we consider the TF value returned by RIG as being the lowest g that implies in more than 50% of the files being abandoned in the system (lines 7-9), using the blame-based definition of abandoned files, explained in the first paragraph of this section. Indeed, RIG authors discuss the use of this solution to compare their results with other TF algorithms, like the one originally defined by Zazworka et al. [2010].

We emphasize two important characteristics of RIG algorithm. First, it is a non-deterministic algorithm, i.e., due to the use of a random sample of developers (line 4), the results produced by the algorithm vary from one execution to another. Second, RIG can finish without computing a valid TF result (line 12). This happens when all groups of developers (line 4) do not meet the conditions of a Truck Factor scenario (line 7).

Algorithm 3: RIG Algorithm

```
   Input: git-blame results for each file in the system
   Output: g (Truck Factor), TFSet (TF developers)
 1 begin
 2   for g ← 1 to 200 do
 3     for i ← 1 to 1,000 do
 4       TFSet ← random sample of g developers;
 5       likelihood ← compute-likelihood (TFSet);
 6       remove-authors(TFSet);
 7       if rate-abandoned-files() ≥ 0.5 then
 8         return g, TFSet;
 9       end
10     end
11   end
12   return null, null;
13 end
```

2.1.1.4 CST Algorithm

Cosentino et al. [2015] proposed, in a tool paper, the CST algorithm that computes the Truck Factor using two sets of developers: primary and secondary developers. *Primary developers (P)* are those that have a minimum knowledge K_p on a given software artifact (e.g., a file). *Secondary developers (S)* are the ones that have at least a knowledge K_s, where $K_s < K_p$. The authors suggest that K_p should be set to $1/D$, where D is the number of developers that have ever changed the artifact and that K_s should be set to $K_p/2$. The Truck Factor of an artifact is defined as $|P \cup S|$, i.e., the number of developers classified as primary or secondary developers. To calculate the

developer's knowledge of a file, Cosentino et al. [2015] propose a set of metrics, including *last change takes it all* and *multiple changes equally considered*. In *last change takes it all*, all knowledge on a file is assigned to the last developer who modified it and in *multiple changes equally considered*, the knowledge of a developer d on a file is defined as C_d/C, where C_d is the number of commits on the artifact performed by d and C is the total number of commits on the artifact. They also propose a strategy to aggregate knowledge data to the level of directory, branch, or project. This aggregation is computed by summing up the knowledge on individual files and scaling the results considering the total number of files in a directory, branch or project. In their tool paper, Cosentino et al. [2015] do not either provide more details nor the pseudo-code of CST algorithm.

2.2 Core Developers

Core Developers are those developers who play a leading role in a project [Yamashita et al., 2015], making important decisions such as defining the system's architecture and implementing critical features [Joblin et al., 2017]. Core developer is related to Truck Factor because both concepts are somehow tightly coupled to the knowledge distribution in software project teams. Nevertheless, we still lack studies to investigate the actual relation between these two concepts. In the present work, we carried out an empirical study to identify the level of relation between Core Developer and Truck Factor (Chapter 6).

2.2.1 Heuristics for Identifying Core Developers

In this section, we present the two heuristics investigated in this work and proposed by Yamashita et al. [2015] to identify Core Developers: *Commit-Based Heuristic* and *LOC-Based Heuristic*. The authors also suggested a third heuristic, called *Access-Based Heuristic*, in which Core Developers are those who have write access to the project repository. However, we were not able to retrieve this kind of data, since only the repository contributors have access to this kind of information. Consequently, this heuristic is not used in this master dissertation.

2.2.1.1 Commit-Based Heuristic

The Commit-based Heuristic identifies Core Developers according to the developer contribution to the software project. In this case, a contribution is defined as a

commit. Yamashita et al. [2015] define that the Core Developers set is constituted by the developers responsible for 80% of the commits of a project.

Algorithm 4 presents the Commit-Based Heuristic. The algorithm receives as input a list of system developers (LD) and the total of commits made in the system (size). First, the set *coreDev* is initialized with an empty set and variable *cumulativeRatio* is initialized with zero (line 2-3). Then, the list of developers is ordered according to the number of commits that each developer performed in the project (line 4). After that, the process of identifying Core Developers starts (5-10). In this process, the first developer on the *LD* list is selected (*dev*) (line 6). Then, the *dev* commit rate is calculated by dividing the *dev*'s number of commits by the total commits of the system (*size*) (*devRatio(dev, size)*). The Core Developers set (line 11) is composed of developers who together are responsible for at least 80% of commits in the project.

Algorithm 4: Commit-Based Heuristic

```
   Input: LD (list of developers of a system S),
   size (total of commits of a system S)
   Output: coreDev (set of Core Developers)
 1 begin
 2     coreDev ← ∅;
 3     cumulativeRatio ← 0;
 4     sort(LD);
 5     while cumulativeRatio ≥ 0.8 do
 6         dev ← head(LD);
 7         cumulativeRatio ← cumulativeRatio
 8         + devRatio(dev, size);
 9         coreDev ← coreDev + dev;
10         remove-developers(dev);
11     end
12     return coreDev;
13 end
```

2.2.1.2 LOC-Based Heuristic

The LOC-Based Heuristic identifies Core Developers according to the number of lines of code added and removed by the developer in the project. More precisely, Yamashita et al. [2015] identify a Core Developer by the sum of the added and removed number lines, which they called *churn*.

The algorithm followed by LOC-Based Heuristic is essentially the same of Commit-Based Heuristic. The only difference is that the input *size* receives the total LOC of the analyzed project. The Core Developers set will be composed of the

developers who together are responsible for at least 80% of the churn of a software project.

2.3 Model Software Architecture

Software architecture is a widely relevant concept in Software Engineering. It is related to the structure and the organization of a software system and aims to make externally visible the software system components and to show how these elements are related to each other [Garlan and Shaw, 1993; Pressman, 2011; Sommerville, 2011; Bass et al., 2012]. In this work, we investigate the relation between Truck Factor and software architecture. For this purpose, we represent software architecture by mean of a generic model previously proposed in the literature. The model, named *Little House*, is described in this section.

2.3.1 Little House

Little House is a model that aims to provide a macroscopic and generic visualization of a software architecture [Ferreira, 2011]. Based on the Bow-tie model (Broder et al. [2000] apud Ferreira [2011]), the Little House model illustrates the componentization of the dependency graph of object-oriented systems. The vertices in the dependency graph represent the classes and the edges represent the dependencies among classes. The dependency relation, in this context, is established by inheritance, by calling a method, or by using a field from another class. Therefore, class `A` uses a class `B` if in `A` there is a reference to `B`. Similarly, we say that `B` is used by `A`. In this case, there will be a directed edge from `A` to `B`.

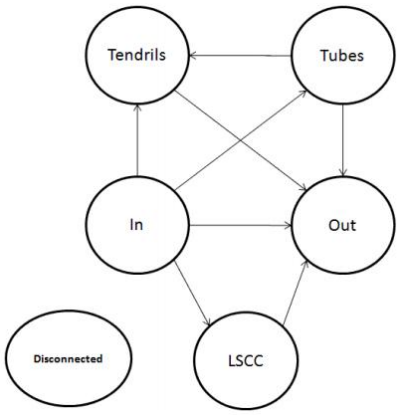
The Little House model is a macro-graph of the relationship among the classes of a software system. This macro-graph is constituted by six vertices, called components. Inside each component, the classes may be connected in an undisciplined way. However, the communication between classes of different components is disciplined. Architectures modeled using Little House model have the following components:

- **Component Tubes**: this component contains classes that use classes of Tubes, In, or Out and that are only used by classes of In.

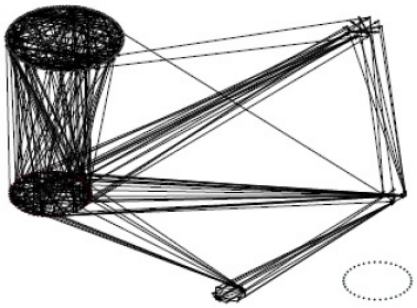
- **Component Tendrils**: this component groups classes that use classes of Out and that can be used by classes of In, Tubes, or Out.

- **Component In**: this component aggregates classes that use classes of any other components, except Disconnected, but which are not used by classes of other components.
- **Component Out**: this component groups classes that do not use classes from other components but are used by classes of the other components, except Disconnected.
- **Component LSCC**: this is the largest strongly connected component of the model. LSCC classes can be reached directly or indirectly by any other class of this component.
- **Component Disconnected**: this component groups classes that are not used by and do not use any class of the other components.

Figure 2.1a shows the Little House model. Figure 2.1b shows the dependency graph of JUnit 4.8 modeled according to Little House.



(a) Little House Model ([Ferreira, 2011]).



(b) JUnit 4.8 as modeled by Little House ([Ferreira, 2011]).

2.3.2 Previous Evaluation of the Little House Model

Studies carried out to assess Little House indicate that *Out*, *In*, and *LSCC* are the components that concentrate classes with the higher degradation as the system evolves [Ferreira et al., 2012]; they also have the worst metric values when compared to the other components [Ferreira et al., 2012]; finally, they are the components that group classes with the highest change impact [Ferreira et al., 2015]. In the present

work, we investigate whether the developers of the Truck Factor set have knowledge in the more complex components of the software: In, Out and LSCC. Such analysis would provide the interpretation of Truck Factor result regarding the importance of the classes in the software architecture.

Chapter 3

Related Work

The interest of Software Engineering community in studying knowledge of code is due to many reasons. As pointed by Fritz et al. [2014], it is hard to a developer to have knowledge about the software system, and the developers usually need to know who they should report their doubts about the code. Therefore, evaluating the distribution of the knowledge of a software system among its contributors is relevant both to developers and to managers.

Previous research on knowledge of code has been carried out with three main purposes: (i) to define models and metrics to assess the level of the developer's knowledge of the code, (ii) to define algorithms to calculate such metrics, and (iii) to evaluate the usefulness of the proposed models and metrics. Truck Factor is one of the main metrics proposed in the literature in this context. However, we still lack studies that analyze and evaluate the behavior of the algorithms proposed to calculate Truck Factor. The present work aims to contribute to overcoming this problem. As this work focuses on evaluating Truck Factor algorithms, we dedicated Section 2.1.1 to describe them. Therefore, in this chapter, we discuss other important related work.

3.1 Authorship and Ownership

Besides Truck Factor, two other concepts are commonly associated with source code knowledge: *ownership* and *authorship*. Authorship is the result of the contribution of a developer (author) to a piece of software, e.g., a line of code, a method, or a class. Ownership refers to the level that a contributor is responsible (or owns) a piece of software [Bird et al., 2011; Foucault et al., 2014]. Rahman and Devanbu [2011] propose metrics for authorship and ownership. For them, authorship is the rate of contribution of a developer to a code fragment, calculated as a function of the number

of lines contributed by him. They define ownership as the author who has the highest authorship on a code element. Truck Factor is related to both concepts since it aims to identify the key contributors of a software project. Torchiano et al. [2011] refer to Truck Factor as the "collective code ownership" of a project. Therefore, Truck Factor is a system-level metric, whereas authorship and ownership are fine-grained metrics.

3.2 Developer Categorization

Assessing how the knowledge of code is distributed among the developers is also important to identify the *major* and the *minor* contributors. The minor contributor has small contributions in the code, whereas the major contributor centralizes the knowledge of the code [Foucault et al., 2014]. A correlated concept is *core and peripheral developers* in software projects. Core developers are those who make the strategic and long-term decisions, including defining the system's architecture and implementing the most critical features. In contrast, peripheral developers are responsible for bug fixes and simple enhancements. A commonly used approach to distinguish core and peripheral developers relies on the number of commits made by each developer. The 80th percentile threshold is then used to separate the developers. Developers with a number of commits above this threshold are core developers; the others are considered peripheral [Joblin et al., 2017].

In this work, we investigate the relation between core developers and truck factor by means of an empirical study. For this reason, the main heuristics for core developers have already been detailed in Section 2.2.

3.3 Experimental Studies on Source Code Knowledge

Rahman and Devanbu [2011] report the results of an experimental study about code ownership and defects. They investigated whether the quantity of developers that work on a file leads to more defects in software systems. Their study was carried out at a fine-grained level, considering lines of implicated code, i.e., lines that are modified to fix a bug. In the experimental study, they considered four open-source projects hosted on GitHub. Among their findings, they concluded that, in general, implicated code is contributed by a single developer, contrasting with the idea that more developers working in a code lead to more defects. They also found that specialized knowledge in the code is more important than the general knowledge of the system to avoid defects.

Bird et al. [2011] also investigated the role of ownership in faults. They proposed a metric for ownership that is based on the concepts of minor and major contributors. A minor contributor of a module has ownership on it less than 5%; a major contributor has ownership of a module equals to or above 5%. They, then, define the ownership of a module as the "proportion of ownership for the contributor with the highest proportion of ownership". They analyzed two large commercial projects, Windows Vista and Windows 7, that were developed by thousands of developers. In their analysis, they considered the Windows compiled binaries as modules. The main finding of their work is that the number of minor contributors of a module is positively correlated with the number of faults in the module. Foucault et al. [2014] replicated the study of Bird et al. [2011] in seven open-source software systems developed in Java. The classes of the systems were considered the modules in their study. In contrast to the results of the study of Bird et al. [2011], Foucault et al. [2014] have not found a strong correlation between ownership and faults, and they concluded that the total number of developers is better than ownership to predict failures.

3.4 Experimental Studies on Truck Factor

Some works have applied Truck Factor in experimental studies. Torchiano et al. [2011] proposed a theoretical model to infer the maximum possible Truck Factor value of a system. The model is based on variables such as the number of source files, the number of developers, and the number of missing developers, (i.e., the number of developers "hit with a truck"). They evaluated the model in 20 open-source software systems, with team size varying from 2 to 38 developers. Truck Factor was calculated in their work by the algorithm proposed by Zazworka et al. [2010]. They compared the results of their proposed model with another proposal referenced in their paper as Govindaray threshold. Although the problem of identifying a threshold for Truck Factor is important, the results of their study are not conclusive and, as the authors themselves recognize, further investigation is necessary. Moreover, their work is based on the algorithm of Zazworka et al. [2010] that has been proved to be applicable only to small projects.

After investigating 2,496 projects hosted on GitHub, Yamashita et al. [2015] report that several projects are susceptible to Truck Factor scenarios. They report that 26%-58% of the projects have core teams that are too small ($\leq$ 10% of active contributors) to be considered compliant with the Pareto principle. Ye and Kishida [2003] mention that the development of GIMP (GNU Image Manipulation Program) was once halted for about 20 months because two developers have left the project.

3.5 Final Remarks

In this chapter, we presented the main works related to this master dissertation. The results of those studies indicate that the problem of assessing developer's code knowledge is in discussion in the literature, with low consensus. Moreover, those studies rely on metrics of developer's code knowledge that have not been solid evaluated. For instance, it is important to investigate whether ownership indeed represents the distribution of knowledge among the contributors. The same occurs with Truck Factor, which is the focus of the present work. The results of our study bring a deeper comprehension on the Truck Factor metric and on how it can be better measured.

Chapter 4

Study Design

In this chapter, we describe the design of the studies conducted in this master dissertation. We carried out three studies that are presented in Chapter 5, 6, and 7. Specific details of the design followed in these studies are described in their respective chapters. This chapter is organized as follows: Section 4.1 presents the research questions investigated in the conducted studies. The used dataset and its cleaning steps are presented in Sections 4.2 and 4.3, respectively. Finally, Section 4.4 describes the configuration, implementation, and execution of the algorithms compared in this work.

4.1 Research Questions

The research questions investigated in this work are grouped in three studies, as described next.

Comparison of Algorithms for Computing Truck Factor

RQ1. How accurate are the results provided by each algorithm?

To answer this question, we define accuracy as a measurement of how close an estimated Truck Factor (by an algorithm – AVL, CST, or RIG) is to the true value (reported in the constructed oracle).

RQ2. How accurate is the identification of TF developers by each algorithm?

This research question targets the cases where an algorithm correctly estimates the Truck Factor, but not the list of key authors. For example, suppose a system with TF=2 and that $\{Mary, John\}$ are the developers responsible for this result. An

algorithm can correctly estimate the system's TF, but by considering $\{Lisa, Mark\}$ as the key developers. Therefore, this second research question investigates how often this situation happens with the studied algorithms.

RQ3. What is the impact of different thresholds and configurations in the results of each algorithm?

The studied algorithms depend on specific thresholds and configurations to produce their results. In the answers of the previous questions, we consider the default thresholds and configurations, as suggested by the algorithms' authors. Therefore, in this third research question, we explore the impact of different thresholds on the algorithms' results.

Truck Factor vs. Core Developers

RQ4. How accurate are the results provided by each heuristic?

In this research question, we aim to identify how close is the number of Core Developers, estimated by the heuristics, to the Truck Factor reported in the oracle.

RQ5. How accurate is the identification of Truck Factor developers by each heuristic?

This research question aims to identify how often Core Developers indicated by the studied heuristics are the same authors reported in the oracle of Truck Factor.

RQ6. What is the correlation between Truck Factor and the Core Developers sets?

In this research question, we investigate the correlation between the Truck Factor and the Core Developers sets, by analyzing their intersection. Our hypothesis is that Truck Factor is made up of a subset of the Core Developers.

Knowledge Distribution in Software Architectures

RQ7. Do Truck Factor developers have knowledge on the most complex components of a system?

In this research question, we investigate whether the Truck Factor developers are those who actually have knowledge of the most complex architectural components of the system.

4.2 Dataset

4.2.1 AVL Dataset

In this work, we reuse the dataset proposed by Avelino et al. [Avelino et al., 2016] to validate the results obtained by their algorithm with GitHub developers. The selection of the software systems included in this dataset was carried out in three steps, as follows.

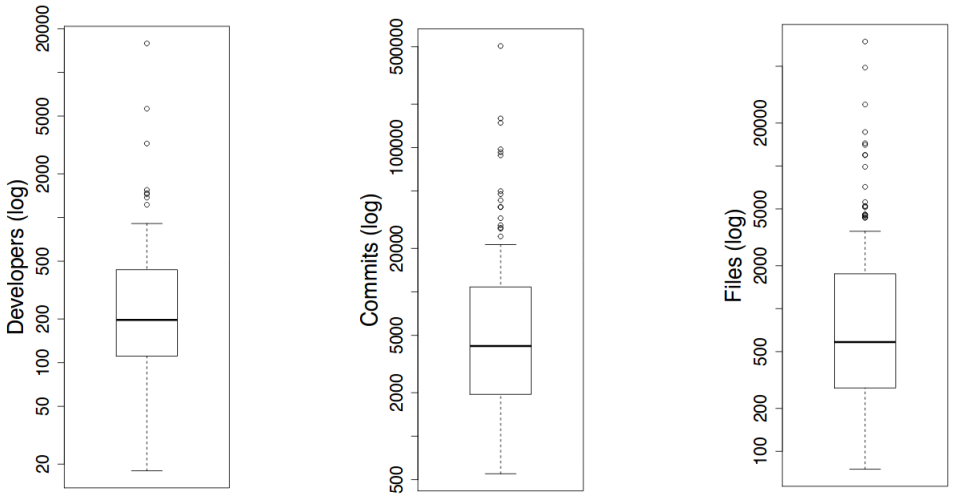
- *Selecting the Programming Language:* Avelino et al. [2016] chose systems from the six programming languages with the largest number of repositories in GitHub: JavaScript, Python, Ruby, C/C++, Java, and PHP.
- *Selecting Software Systems*: Avelino et al. [2016] selected the top-100 most popular systems of each of the languages identified in the previous step. The adopted popularity criterion adopted was the number of system's stars.
- *Filtering the Software Systems:* From the systems obtained in the previous step, the most important ones in each programming language were selected, according to their number of developers, commits, and files. Basically, the systems that were in the first quartile of the distribution of the number of developers, commits, and files were discarded by Avelino et al. [2016]. Finally, systems with evidence of incorrect migration (i.e., with more than 50% of system files added to the repository in less than 20 commits) were discarded.

Table 4.1 characterizes the AVL dataset used in this study. It includes 133 systems, implemented in six different programming languages; Ruby is the language with more systems (33), and PHP is the language with fewer systems (17). Considering all systems, the dataset includes more than 63,000 developers, 373,000 files, and 2 million commits.

Language	**#Systems**	**#Devs**	**#Commits**	**#Files**
JavaScript	22	5,740	108,080	24,688
Python	22	8,627	276,174	35,315
Ruby	33	19,960	307,603	33,556
C/C++	18	21,039	847,867	107,464
Java	21	4,499	418,003	140,871
PHP	17	3,329	125,626	31,221
Total	**133**	**63,194**	**2,083,353**	**373,115**

Table 4.1: AVL dataset (Avelino et al. [2016])

Figure 4.1 shows the distribution of the number of developers (Figure 4.1a), *commits* (Figure 4.1b) and files (Figure 4.1c) of the dataset. For number of developers, the first, second and third quartiles are 111, 197, and 437, respectively. The systems with the highest number of developers are torvalds/linux (15.8K), Homebrew/homebrew (5.6K), and rails/rails (3.2K). Regarding the number of commits, the first, second and third quartiles are 1,950, 4,210 and 10,800, respectively. The three systems with the highest number of commits areandroid/platform_frameworks_base (159.3K) and JetBrains/intellij-community (148.5k). Finally, for system source code files, the first, second, and third quartiles are 269, 574, and 1,750, respectively. The three systems with the largest number of files are JetBrains/intellij-community (74K), torvalds/linux (49K), and android/platform_frameworks_base (27K).



(a) Developers (b) Commits (c) Files

Figure 4.1: AVL Dataset: distribution of number of developers, commits, and files (Avelino et al. [2016])

4.2.2 Oracle of Truck Factors

To answer the research questions investigated in this master dissertation, we created an oracle with the Truck Factor of open-source systems hosted on GitHub. First, we populated this oracle with a subset of the systems in the AVL dataset [Avelino et al., 2016]. In their work, Avelino et al. [2016] asked the developers of 133 open-source systems, by means of issues that are publicly available on GitHub, if they agree with the TF results estimated by the AVL algorithm. Developers of 67 systems answered to these issues. To create our oracle, we carefully read these answers, aiming to retrieve TF results and developers. We were able to retrieve this information in two situations:

1) When the GitHub developers agreed with the result produced by AVL. For example, for system netty/netty, AVL estimates TF = 2 and the project's developers promptly agreed with this value (*"Yes. Let's just say we all hope here that neither of them get hit by a truck."*)[1].

2) When the developers did not agree with the result produced by AVL, but mentioned the correct value in their answer, together with the name of the developers responsible for this value. For example, developers of ipython/ipython did not agree with the result suggested by AVL (TF = 4) and argued that the correct value is 5 (*"... though I would add [developer name] who is the expert on many pieces of the code."*).[2]

Out of 67 answers, 20 answers (29.9%) fit in the first described situation and seven answers (10.4%) fit in the second one. Therefore, 40 answers (59.7%) were discarded because they do not fit any of the situations, i.e., in these cases it is not possible to figure out the TF number and the TF developers by inspecting the GitHub issues. It is also important to clarify that the aforementioned procedure does not imply in a bias towards AVL. All TF results included in the oracle were validated by the systems' principal developers. Indeed, as mentioned, seven results are different from the ones generated by AVL.

We also extended this initial oracle with new systems. For this purpose, we started by considering systems from the top-6 most popular programming languages on GitHub, i.e., JavaScript, Ruby, Python, PHP, Java, and C/C++. For each language, we selected 100 systems with the highest number of stars (a common measure of popularity of GitHub software [Borges et al., 2016]) and attending the following criteria: at least 100 contributors and three years of development history. The goal was to select mature systems with a large community of developers. We only considered systems that use GitHub to handle issues and that are not already in the oracle. This initial selection produced 27 systems. We then opened issues for each system on GitHub, asking two questions: (a) What is the TF of [project-name]? (b) Who are the developers responsible for this TF result? We received answers from 8 systems, which corresponds to a response ratio of 29.6%.

The final oracle provides the Truck Factor of 35 systems, including 27 systems reused from AVL's work and eight new systems. Our resulting oracle is presented in Section 5.2.

[1] https://github.com/netty/netty/issues/4069
[2] https://github.com/ipython/ipython/issues/8710

4.3 Handling Third-Party Libraries and Aliases

Before running the Truck Factor algorithms over the commit history of the 35 systems evaluated in this work, we performed two cleaning steps. First, we used the open-source Linguist tool[3] to remove third-party libraries and other external source code from the cloned repositories. Linguist relies on an extensive set of pattern matching rules to detect external code copied to GitHub repositories. The tool is implemented and evolved by GitHub with the main purpose to provide statistics on the amount of source code in a repository implemented in different programming language (which also requires discarding external code, as in our case). Second, we also handled developer's names alias in the evaluated commits. In this step, we listed all developers of each project and automatically grouped those who had the same name linked with more than one e-mail and those who had the same e-mail linked with different names.

4.4 Implementing, Configuring, and Executing the Algorithms

To compute the TF results estimated by AVL, we used the tool provided by the algorithm's authors, which is publicly available on GitHub.[4] To compute the results estimated by RIG, we implemented the algorithm, since the authors do not provide a public tool. For CST, we use the tool provided by the authors, also available on GitHub.[5] When answering RQ1 (*How accurate are the results provided by each algorithm?*) and RQ2 (*How accurate is the identification of TF developers by each algorithm?*), we used the default configuration suggested by the algorithms' authors (including the thresholds reported in Section 2.1.1). In the case of CST, we used *multiple changes equally considered* as the metric to infer developer's knowledge. To compute the Core Developers estimated by Commit-Based Heuristic and LOC-Based Heuristic, we implemented the algorithms based on the description of the heuristics presented by Yamashita et al. [2015].

When executing the algorithms over AVL's dataset, we used the commits available on GitHub on August, 2015 (when the survey with the developers was concluded). For the remaining systems, we used the commits available on September, 2016 (date of the second survey).

[3]https://github.com/github/linguist
[4]https://github.com/aserg-ufmg/Truck-Factor
[5]https://github.com/SOM-Research/busfactor

Due to its non-deterministic behavior, RIG results reported when discussing RQ1 and RQ2 refer to the median measure of 30 executions. Specifically, in RQ2 we report the median precision, recall, and F-measure results for these executions. Even after 30 runs, RIG failed to produce valid results for two systems: symfony/symfony and saltstack/salt. This happened because the 200,000 groups of developers tested by the algorithm for each of these two systems do not meet the conditions of a Truck Factor scenario (i.e., at least 50% of files abandoned, according to the git-blame criterion defined by RIG authors). Therefore, when discussing the first two research questions, RIG results do not include these two systems.

Chapter 5

Comparison of Algorithms for Computing Truck Factor

In this chapter, we present a comparative study of three recent algorithms for estimating Truck Factor metric. Section 5.1 describes the context and resumes the research questions of the study. Section 5.2 presents the oracle for Truck Factor, proposed in this work to assist the investigation of these research questions. Section 5.3 presents the results of the research questions. Section 5.4 discusses broader questions about algorithms. Finally, Section 5.6 discusses the final considerations of this chapter.

5.1 Motivation

Due to the relevance of Truck Factor data to project managers and users of open-source software, algorithms have been proposed to automatically estimate Truck Factors by using code-ownership data retrieved from version control repositories [Avelino et al., 2016; Rigby et al., 2016; Cosentino et al., 2015]. However, to the best of our knowledge, we still lack studies that (i) validate the results produced by such algorithms, and (ii) compare these results and discuss the main benefits and limitations of each algorithm. Therefore, in this chapter, we assess three recent algorithms proposed to estimate Truck Factors. The main characteristics of these algorithms are described next, since they have been detailed in Section 2.1.1.

- *AVL Algorithm*

  The algorithm proposed by Avelino et al. [2016], which is based on a degree-of-authorship (DOA) metric and on a greedy heuristic to infer the key developers in a project.

- *RIG Algorithm*

  The algorithm proposed by Rigby et al. [2016], as part of a study to assess the impact of turnover in software projects, which uses a git-blame-based heuristic to infer code ownership.

- *CST Algorithm*

  The algorithm proposed by Cosentino et al. [2015], which computes Truck Factors for files and then scales the results to the level of branches and projects.

This study aims to answer the following research questions:

- RQ1 – How accurate are the results provided by each algorithm?
- RQ2 – How accurate is the identification of TF developers by each algorithm?
- RQ3 – What is the impact of different thresholds and configurations in the results of each algorithm?

5.2 Oracle of Truck Factors

We built an oracle with the Truck Factor of 35 open-source systems, as indicated by the main developers of these systems to validate and compare the algorithms. We use open-source software in the study due to the importance of Truck Factor information in this context of software development. Moreover, we surveyed the main developers of these projects to construct the oracle because they are probably the only authority capable of providing this information.

The main characteristics of the systems of our oracle, including the TF results according to the systems' main developers, are summarized in Table 5.1. The oracle includes well-known systems, such as junit-team/junit4, d3/d3, and ReactiveX/RxJava. The number of stars ranges from 2,863 (nicolasgramlich/AndEngine) to 59,184 (d3/d3). The number of contributors range from 21 (nicolasgramlich/AndEngine) to 7,265 (saltstack/salt) and the system's age ranges from 2 (facebook/osquery) to 16 years (junit-team/junit4).

Figure 5.1 shows a histogram of the distribution of the system's Truck Factor. As we can see, 20 systems (57.1%) have TF=1; and five systems (14.3%) have TF=2. Only two systems have TF $>$ 10, saltstack/salt (TF = 11) and symfony/symfony (TF = 15). The primacy of systems with a small Truck Factor (e.g., TF $\leq$ 2) is common in open-source systems, which are usually maintained and evolved by a small team of core developers Mockus et al. [2002].

System	Language	Stars	Contributors	Age	TF
alexreisner/geocoder	Ruby	4,309	233	8	1
./androidannotations	Java	8,781	58	6	2
atom/atom-shell	C++	40,401	526	4	1
bjorn/tiled	C++	4,359	137	9	1
celluloid/celluloid	Ruby	3,457	90	6	1
chef/chef	Ruby	4,621	502	9	4
d3/d3	JavaScript	59,184	117	6	1
dropwizard/metrics	Java	4,562	131	7	1
facebook/osquery	C++	7,719	126	2	2
gruntjs/grunt	JavaScript	11,271	64	5	1
ipython/ipython	Python	11,005	473	9	5
Leaflet/Leaflet	JavaScript	17,295	446	5	1
less/less.js	JavaScript	14,371	208	7	1
mailpile/Mailpile	Python	6,756	119	5	1
netty/netty	Java	8,837	238	8	2
nicolas.../AndEngine	Java	2,863	21	7	1
pallets/flask	Python	24,670	370	7	1
powerline/powerline	Python	6,892	84	4	1
puphpet/puphpet	PHP	3,729	147	4	1
ReactiveX/RxJava	Java	20,457	132	5	1
requirejs/requirejs	JavaScript	10,121	98	7	1
Respect/Validation	PHP	3,671	92	6	3
saltstack/salt	Python	7,265	1,701	6	11
sandstormio/capnproto	C++	4,295	70	4	1
sass/sass	Ruby	9,176	179	10	1
SFTtech/openage	C++	5,472	94	3	2
thoughtbot/paperclip	Ruby	8,300	348	9	1
capistrano/capistrano	Ruby	9,141	207	4	2
deis/deis	Python	5,936	163	4	3
ruby-grape/grape	Ruby	7,754	230	7	4
cantino/huginn	Ruby	15,473	135	3	3
junit-team/junit4	Java	5,602	133	16	4
kennethreitz/requests	Python	22,874	440	6	3
symfony/symfony	PHP	13,658	1,339	7	15
tornadoweb/tornado	Python	12,788	246	7	1

Table 5.1: Oracle of TF results. Systems in the first table's section are reused from Avelino et al. [2016]. Age is measured in years.

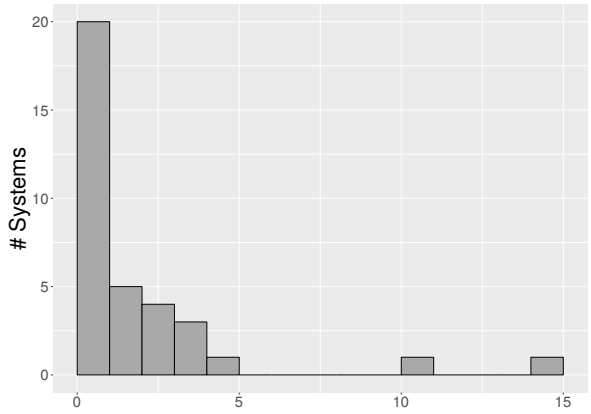


Figure 5.1: Distribution of the system's Truck Factor

5.3 Results

This section presents the results of the investigation to answer RQ1, RQ2, and RQ3.

RQ1. How accurate are the results provided by each algorithm?

To answer this first question, we compute the *error* of the Truck Factors estimated by each algorithm, compared to the oracle values, as follows:

$$error = \mathrm{TF}_{algorithm} - \mathrm{TF}_{oracle} \tag{5.1}$$

Figure 5.2 shows violin plots with the absolute error measures. AVL and CST have very similar results. In both algorithms, the first quartile and the median of the error measures are 0, and the third quartile is 1. By contrast, RIG presents worst results, with a median error of 2.

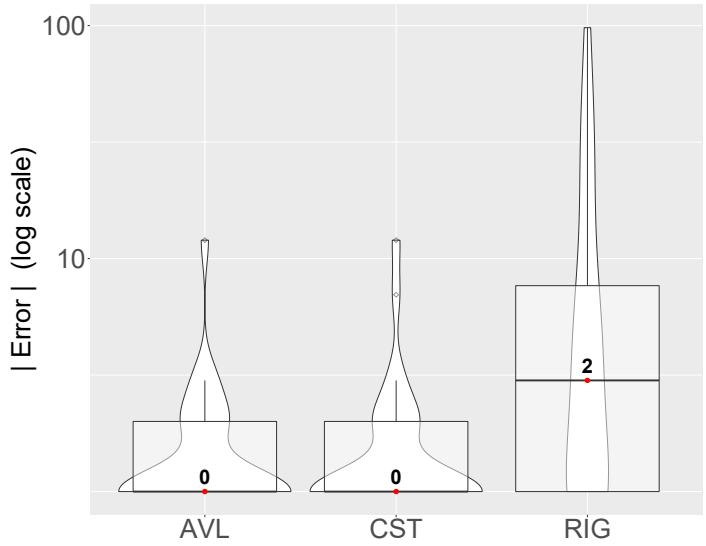


Figure 5.2: Absolute error results

Six systems with an outlier behavior regarding their error measures are listed in Table 5.2. The table shows the TF of each system (as indicated by their developers) and the error of the results computed by AVL, RIG, and CST. As we can see, RIG presents the highest error measures. For the first five systems in this table, the algorithm's error ranges from 21 (junit-team/junit4) to 58 (ruby-grape/grape). Furthermore, RIG was not able to compute a TF for symfony/symfony (as mentioned in Section 4.4). Indeed,

symfony/symfony is the system with the highest error produced by AVL and CST. It has TF=15, but both AVL and CST estimated a TF of 11 for the system.

System	TF	AVL	RIG	CST
ruby-grape/grape	4	1	58	2
ipython/ipython	5	1	34	1
alexreisner/geocoder	1	0	31	0
Leaflet/Leaflet	1	0	23	0
junit-team/junit4	4	2	21	1
symfony/symfony	15	11	–	11

Table 5.2: Error measures for outlier systems

Table 5.3 shows how the algorithms perform for groups of systems with different TF results. The table divides the 35 systems in three groups: systems with TF = 1 (20 systems), systems with $2 \leq \text{TF} \leq 5$ (13 systems), and systems with $\text{TF} \geq 6$ (two systems). For each group, the table presents the percentage of systems each algorithm was able to estimate their TF precisely, i.e., *error* = 0. AVL is the most accurate algorithm, with 100%, 30%, and 50% of perfect accuracy for the groups TF1, TF2-5, and TF6+, respectively. However, CST has a very close performance, with results of 85%, 46%, and 50% for the same groups. RIG results are worse; for example, the algorithm correctly infers the results of 60% of the systems in TF1. In Table 5.3, we also observe that the accuracy declines for the systems with high TF results. For example, although AVL is able to precisely infer the TF of all systems in TF1, the algorithm infers correctly the TF of four system (out of 13 systems) in TF2-5. A similar behavior happens with CST, although this algorithm outperformed AVL for TF2-5.

Group	Range	Size	Alg	%
TF1	TF = 1	20	AVL	100
			RIG	60
			CST	85
TF2-5	$2 \leq \text{TF} \leq 5$	13	AVL	30
			RIG	40
			CST	46
TF6+	$\text{TF} \geq 6$	2	AVL	50
			RIG	0
			CST	50

Table 5.3: Percentage of results with *error* = 0 per group of systems. Size is the number of systems in each group.

Summary: AVL and CST are the most accurate algorithms. They correctly estimated ($error = 0$) the Truck Factor of 71.4% (AVL) and 68.6% (CST) of the systems in the oracle. However, their accuracy decreases for systems with high TF values. For both algorithms, the highest error is 11, produced for a system with TF=15. RIG has the worst accuracy; it estimates correctly the TF of only 34.3% of the systems.

RQ2. How accurate is the identification of TF developers by each algorithm?

Besides the Truck Factor, the algorithms report the TF sets, i.e., the developers responsible for the estimated Truck Factor. In this second research question, we compare the TF sets reported by each algorithm with the TF sets in the oracle. To this purpose, we use three metrics: precision, recall, and F-measure. These metrics are based on the analysis of true positive, false positive, and true negative cases. A true positive occurs when a developer in the TF set computed by the algorithm is also in the oracle. A false positive occurs when a developer indicated in the TF set by the algorithm is not in the oracle. A false negative occurs when a developer indicated by the oracle is not in the algorithm's TF set.

Figure 5.3 shows violin plots with the results of precision for the three algorithms. Precision is defined as described in Equation 5.2, where TP and FP are the set of true and false positives returned by the algorithm, respectively. For AVL, the first quartile, the median, and the second quartile are 1. For CST, precision is slightly lower than AVL; the first quartile is 0.54, the median and the third quartile are 1. For RIG, the first quartile is 0.09, the median is 0.33, and the third quartile is 1. Therefore, AVL has the highest precision, followed by CST. Furthermore, RIG presented the worst precision results.

$$Precision = \frac{TP}{TP \cup FP} \tag{5.2}$$

Recall measures the rate of true TF developers that the algorithms are able to find. Recall is defined in Equation 5.3, where FN is the set of false negatives returned by the algorithm.

$$Recall = \frac{TP}{TP \cup FN} \tag{5.3}$$

Figure 5.4 shows violin plots with the results of recall. The three algorithms have median recall of 1; the third quartiles are also 1. For RIG and AVL, the first quartile is 1, and for CST, the first quartile is 0.70. Therefore, RIG and AVL have the best recall, followed by CST.

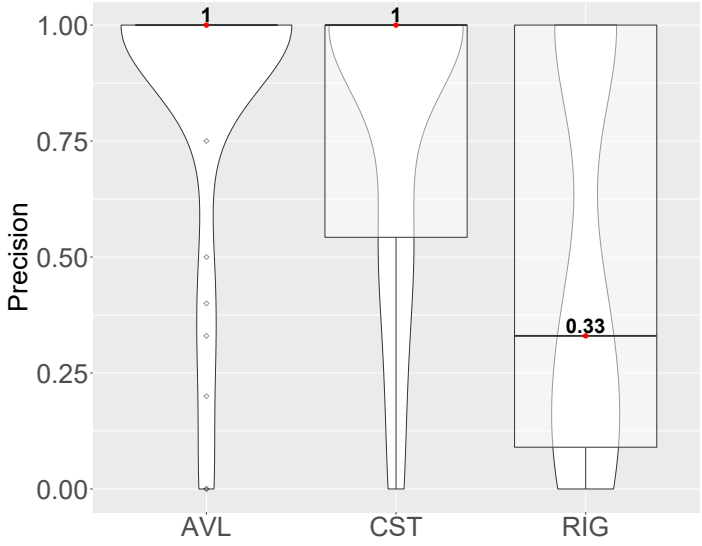


Figure 5.3: Precision

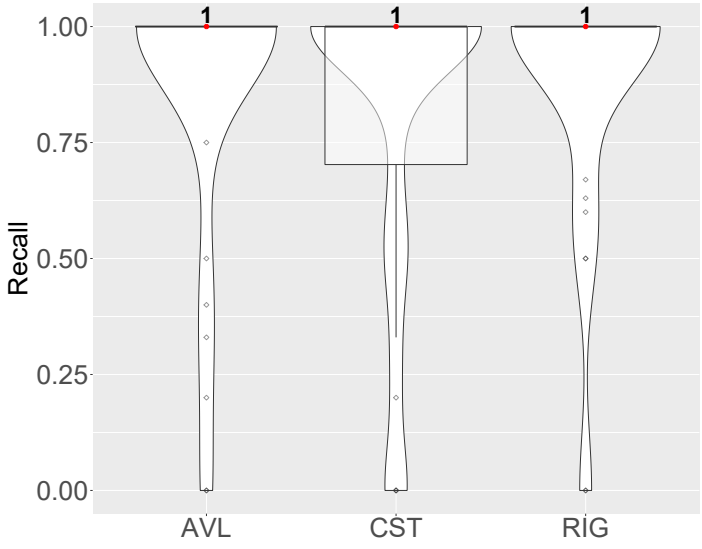


Figure 5.4: Recall

F-measure is the harmonic mean of precision (P) and recall (R), as described in Equation 5.4. As shown in Figure 5.5, for AVL, the first, median, and third quartiles are 0.70, 1, and 1, respectively. For CST, the same measures are 0.60, 1, and 1. For

RIG, they are 0.17, 0.5, and 1. Therefore, AVL has the highest results for F-measure, closely followed by CST and then by RIG.

$$F - measure = \frac{2 * P * R}{P + R} \tag{5.4}$$

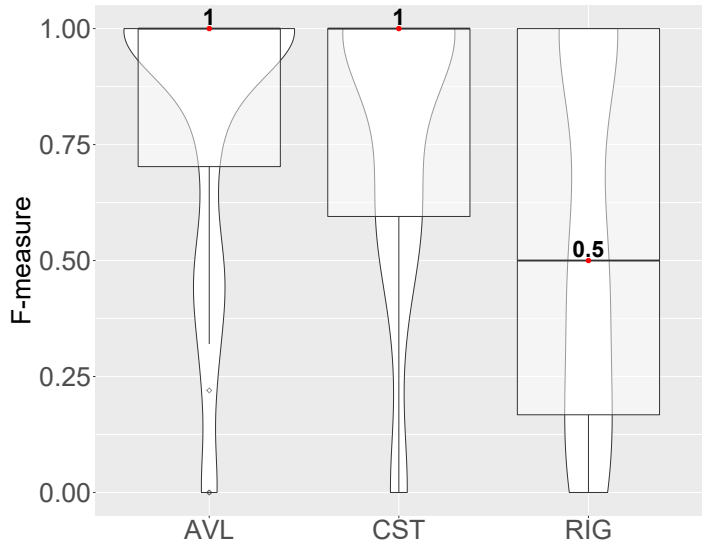


Figure 5.5: F-measure

Summary: Regarding the first quartile measures, AVL has the highest precision (1.00); CST has a lower precision (0.54), and RIG has the worst one (0.09). RIG and AVL are the algorithms with the highest recall (1). CST has the second best recall (0.70). Regarding F-measure, AVL is the best algorithm (0.70), followed by CST (0.60), and RIG is the less accurate algorithm (0.17).

RQ3. What is the impact of different thresholds and configurations in the results of each algorithm?

AVL Algorithm

The algorithm depends on a threshold that defines the rate of abandoned files that configures a Truck Factor disaster (see the algorithm in Section 2.1.1.2). AVL's authors suggest to set this threshold to 0.5, but they do not provide detailed evidence that this is indeed the best option. They only mention that "our estimation relies on a coverage assumption: a system will face serious delays or will be likely discontinued

if its current set of authors covers less than 50% of the current set of files in the system" [Avelino et al., 2016].

Figure 5.6 shows the results of varying this threshold in the systems of our oracle. In the x-axis, the threshold ranges from 0.1 to 1.0. The curves represent the number of systems where $error = 0$ and the number of systems where $|error| \leq 1$, for each threshold. Therefore, the second curve tolerates a small error in the Truck Factor results. Both curves achieve their maximal value for the threshold of 0.5. Therefore, this result provides additional confidence to the default threshold recommended by AVL's authors.

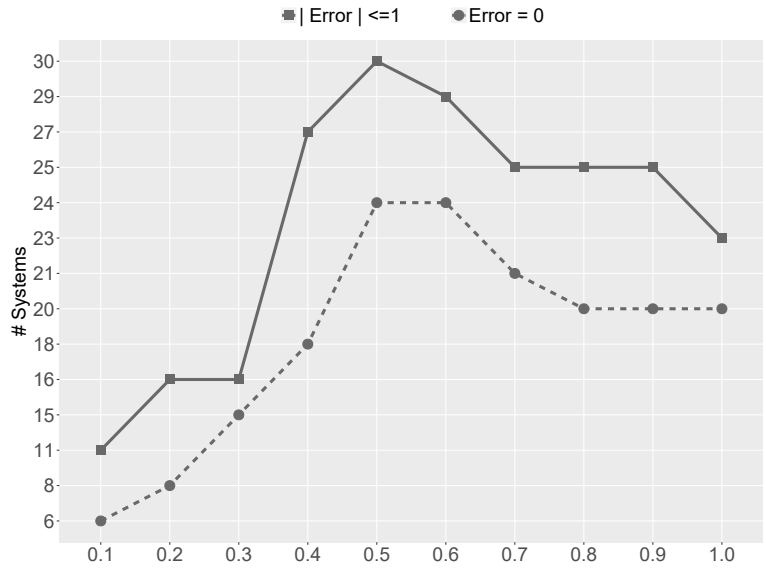


Figure 5.6: Number of systems with $error = 0$ and $|error| \leq 1$, after varying AVL's threshold on abandoned files.

Summary: AVL assumes a project will face serious maintenance problems if its current set of contributors cover less than 50% of the files in the system. In our study, this threshold indeed has the best results, producing an accurate Truck Factor result in 24 out of 35 systems.

RIG Algorithm

We rerun RIG varying the number of random samples of developers tested by the algorithm. In its original configuration, RIG generates 1,000 samples of developers (see Section 2.1.1.3, Algorithm 2, line 3). First, we increase this threshold to 2,000 samples, aiming to double the likelihood of selecting the correct group of developers. We kept

the maximal TF tested by the algorithm ($g = 200$) since in the oracle the highest TF is 15. Moreover, we also kept the threshold of 50% of abandoned files, which is used to define a Truck Factor scenario. After setting this configuration, we executed the algorithm 30 times, as performed when answering RQ1 and RQ2. Figure 5.7 reports the absolute error of each run, for the oracle systems that are not reused from AVL's work. We restrict the figure to these systems for the sake of legibility.

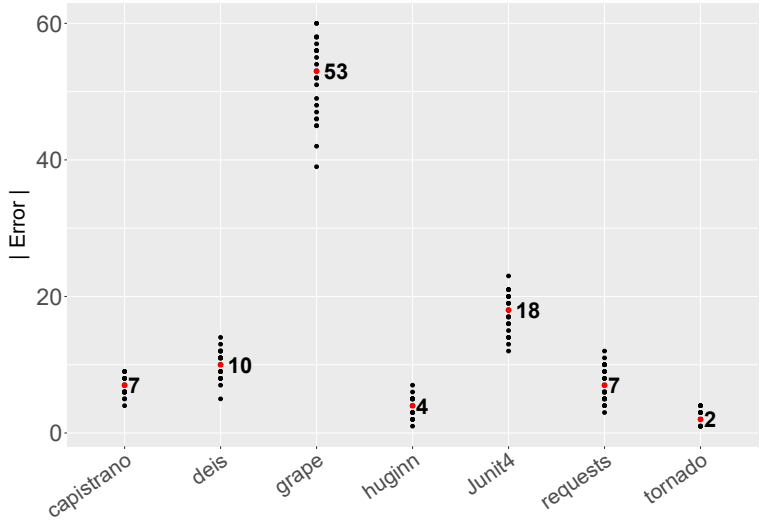


Figure 5.7: Dispersion of RIG errors, considering 30 runs. The highlighted value (in red) is the median error.

RIG does not produce results for symfony/symfony (as in RQ1 and RQ2). In Section 5.4, we elaborate on the reasons why RIG does not finish with a valid TF in some systems. For now, we mention that the results in Figure 5.7 confirm that RIG is a non-deterministic algorithm, producing different outputs, from one execution to another. Considering 30 runs, the dispersion of the error results (maximal error minus minimal error) ranges from 3 developers (tornadoweb/tornado) to 21 developers (ruby-grape/grape).

Figure 5.8 shows the results of varying the number of samples tested by RIG from 1,000 (default value proposed by the algorithm) to 10,000 samples. The curves represent the number of systems where $error = 0$ and the number of systems where $|error| \leq 1$, for each threshold and considering the median results of 30 runs. For the first curve, varying the number of samples has no impact on the algorithm results. In all tested thresholds, RIG matches the Truck Factor of 12 systems (out of 35 systems). For the second curve, there is an increment in the number of matched systems, from

14 systems (1,000 samples) to 17 systems (5,000 samples). After this threshold, the curve remains constant.[1]

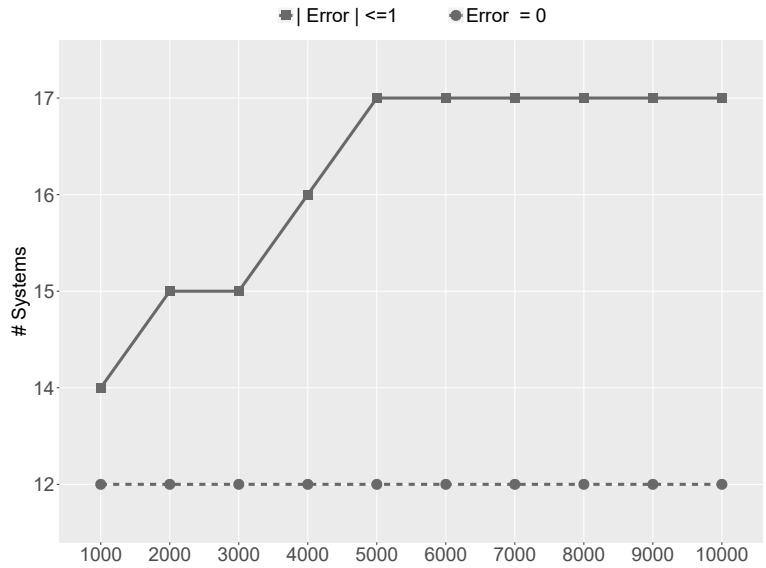


Figure 5.8: Number of systems with $error = 0$ and $|error| \leq 1$, after varying RIG's threshold on the number of tested samples.

Summary: RIG has a non-deterministic behavior, which can cause, for example, a difference of 21 developers in the Truck Factor estimated by the algorithm, from one execution to another. Increasing the number of tested samples—from 1,000 to 10,000 samples—does not have a major positive impact on RIG results

CST Algorithm

As described in Chapter 4, the algorithm depends on two thresholds: primary developers knowledge (K_p) and secondary developers knowledge (K_s). It is also possible to configure the metrics used to compute knowledge on the source code. In this case, the options are *last change takes it all (LCTA)* and *multiple changes equally considered (MCEC)*. However, CST is implemented as a GUI-based tool, and the algorithm is tightly coupled to the interface layer. For this reason, it is not simple to experiment with different thresholds, by means of scripts that call the core algorithm's implementation. Given this context, we investigate only the impact of changing the knowledge metrics on CST results. Table 5.4 shows the number of systems with $error = 0$ and $error \leq 1$.

[1]We also ran experiments for 100,000 samples. After increasing the number of analyzed samples, the results presented by the RIG algorithm did not show significant improvements: 13 systems presented error = 0, and 21 systems have |error| $\leq$ 1.

In both scenarios, MCEC leads to more accurate result than LCTA. For example, 24 systems have their Truck Factor correctly estimated when CST is configured to use MCEC. When LCTA is used, this number decreases to 18 systems.

Metric	$error = 0$	$error \leq 1$
LCTA	18	25
MCEC	24	30

Table 5.4: Number of systems with $error = 0$ and $error \leq 1$, when CST is configured to use LCTA and MCEC metrics.

Summary: When using CST, MCEC is the knowledge metric that leads to the best results. When MCEC is used (as in RQ1 and RQ2), there is an increment of six systems with $error = 0$, when compared to the alternative metric LCTA.

5.4 Discussion

5.4.1 What is the runtime performance of each algorithm?

We did not investigate runtime performance because the three algorithms execute very fast, for all systems in the oracle. For each combination of algorithm and system, the execution requires less than one minute (in an HP Xeon six-Core server, with 64 GB RAM, and Ubuntu 12.04). However, this time also considers the checkout of the systems from GitHub and the preprocessing steps to remove non-source code files, third-party APIs, and to handle alias. Specifically, the tool that implements the CST algorithm uses a database to store data about the source code and the developers of the target systems. Creating and populating this dataset is the most time-consuming task when using CST. It required for example around eight hours in the case of symfony/symfony and saltstack/salt, which are the two system with more contributors in the oracle.

5.4.2 Why and when the algorithms fail?

RIG Algorithm

The algorithm is inspired on precursor algorithm to estimate Truck Factors. Zazworka's has a serious scalability problem, since it tests all combinations $\binom{N}{g}$, where N is the number of contributors and g ranges from 1 to N. To tackles this problem, RIG selects – for each g – exactly 1,000 random samples of developers. Therefore, the

algorithm can easily miss the group of TF developers. For example, rub-grape/grape has TF=4 and 230 contributors (see Table 4.2.2). An exhaustive algorithm must test $\binom{230}{4} = 113,525,855$ combinations to guarantee the selection of the correct group of four developers. However, RIG only tests 1,000 combinations. As a result, RIG suggests a TF = 56 for this system, i.e., it only evaluates the TF developers when testing for combinations of 56 developers.

The random selection of developers also leads RIG to does not present results for some systems. In such cases, no tested samples meet the conditions of a Truck Factor disaster. Indeed, this happens for two systems, symfony/symfony and saltstack/salt, when answering RQ1 and RQ2. In RQ3, after increasing the number of samples to 2,000, RIG started to estimate the Truck Factor of saltstack/salt.

AVL and CST Algorithms:

We acknowledge that explaining the reasons for AVL and CST failures would require a second round of surveys with the developers of the systems in our oracle, asking then why certain developers with relevant contributions to their systems – counting the number of commits – are not ranked as TF developers, while other developers with less relevant contributions are. For example, in symfony/symfony (TF=15) one of the TF developers is the 22nd developer with more commits in the system. However, he seems to play an important social role in the system's community. For example, the announcement of his engagement in the group of symfony/symfony's core developers mentions that "*he is well known for being the one able to get hundreds of comments on his pull requests; he knows how to engage the community!*"[2]. In other words, developers can play specific roles in a project, which are important enough to raise them to the level of TF developers, but that are not reflected in their commit history. In such cases, AVL, CST, and RIG will fail to identify them.

5.4.3 Are Truck Factor scenarios realistic?

Rigby et al. [2016] claim that Truck Factor scenarios (i.e., the loss of all TF developers) computed using loss percentages are unrealistic. They mention that, for large projects, these algorithms produce results with hundreds of developers, for instance. Therefore, only outsourcing an entire project or Google removing funding for Chrome could trigger such massive loss of developers. For this reason, they suggest that Truck Factor should be renamed to "Airplane Factor", to refer to an airplane disaster with all developers on board.

[2]http://symfony.com/blog/new-symfony-core-team-member-grgoire-pineau

In fact, the likelihood of a disaster decreases as the Truck Factor increases. For example, it is more likely a Truck Factor scenario in a single-man project than in projects with 200 key developers, as seems to be the case of Google Chrome. However, the Truck Factor computation also reveals the key developers in a project (and the project's area where they have expertise on). This is useful both in small and in large projects. For example, it is indeed unlikely that 200 developers will leave a project at once. However, even in this case, managers can benefit from information on project's key developers (among possibly thousands of other developers) and about the code locations where they are making their contributions.

5.5 Threats to Validity

Construct Validity

The data gathering of AVL results was performed by a tool developed by the algorithm's authors. We could access the source code of the tool so that we were able to inspect it and verify that it implements AVL as described in this work. The data of CST was also gathered by a tool developed by the authors of the algorithm; however, we did not inspect the source code of this tool. Nevertheless, as the tool is implemented by the authors who proposed CST, we consider that it follows the algorithm presented by Cosentino et al. [2015]. We did not find any public tool that implements RIG. Then, we implemented the algorithm, trying to follow the algorithm description presented by its authors carefully. Since the algorithm's core is not complex, we claim the risks of misunderstandings are small. In fact, the most complex part of RIG is the one that computes the likelihood of each disaster scenario, which is not used in our comparison. Our central interest is the determination of TF values, despite the likelihood of their occurrence.

Internal Validity

We use an oracle of Truck Factors that comprises 35 open-source software systems hosted on GitHub. The oracle has two types of data: the Truck Factor number and the name of the developers that are part of the Truck Factor. The data were gathered in two ways: (i) an initial oracle was constructed by presenting AVL results to the contributors of the software systems and asking them to indicate whether the results are correct or not; (2) by asking the contributors of eight open-source systems to indicate the data, i.e., the Truck Factor value as well as the developers they consider to be part

of the Truck Factor. One may consider that using two approaches to constructing the oracle might be misleading. However, these approaches are complementary and, in both cases, the data are based on the opinion of key team members. To mitigate a threat related to inaccurate answers we only considered consensual responses coming from the top-10 contributors to the projects. Finally, the contributors are identified by their names on GitHub. However, there are cases of contributors with more than one name. To solve this issue, we pre-process the data in order to merge names referring to the same contributor.

External Validity

The data analyzed in this work are from 35 open-source software systems hosted on GitHub. Aiming representativeness, we considered systems implemented in six popular programming languages. The systems have different popularity, varying from 2,863 to 59,184 stars on GitHub. The same occurs with the team size and the maturity of the projects: the teams vary from 21 to 1,701 contributors, and the systems have from 2 to 16 years. Even though, it is not possible to claim that the study results generalize to any open-source software. In the same vein, they may not generalize to proprietary software.

5.6 Final Remarks

In this chapter, we provided a comprehensive study comparing three Truck Factor algorithms: AVL (proposed by Avelino et al. [2016]), RIG (proposed by Rigby et al. [2016]), and CST (proposed by Cosentino et al. [2015]). For this purpose, we relied on an oracle with the Truck Factor of 35 open-source projects hosted on GitHub. The oracle data are based on information provided by the projects' teams. We investigated three aspects in an experimental study: (i) accuracy of the computed Truck Factors, i.e., the ability of the algorithms to predict a Truck Factor close to the one in the oracle data; (ii) accuracy of the Truck Factor sets, i.e., the ability of the algorithms to predict the developers responsible for the Truck Factors; and (iii) the impact of different thresholds and configurations on the algorithms results.

The main findings of this study are as follows.

1. AVL and CST are the most accurate algorithms for predicting the Truck Factor of open-source projects.

2. AVL is the most accurate algorithm to predict the Truck Factor sets, i.e., the developers responsible for a given Truck Factor, closely followed by CST.

3. The best threshold for AVL is 50%, i.e., in order to get the most accurate results, we should configure AVL to consider that a project will be at risk if its current developers cover less than 50% of the files in the system.

4. RIG has a non-deterministic behavior and changing the number of samples tested by the algorithm has a minor impact on its results.

5. The *multiple changes equally considered* metric used by CST to infer code knowledge leads to the best results.

Chapter 6

Truck Factor vs. Core Developers

This chapter aims to conduct a comparative study between the concepts of Truck Factor and Core Developers. Section 6.1 describes the context and the research questions of the study. The results and discussion are presented in Sections 6.2 and 6.3, respectively. The main conclusions of this study are presented in Section 6.4.

6.1 Motivation

Developers that contribute to open-source projects are usually categorized by the level of participation in these projects [Foucault et al., 2014; Joblin et al., 2017]. In this sense, project contributors can be categorized in *core* and *peripheral* developers. Core developers are those "*who take a leading role in the development and maintenance of a software project*" [Yamashita et al., 2015]. Core developer is, then, a concept related to Truck Factor, since both aim to identify developers who play a central role in software development projects.

In this chapter, we present a study that aims to answer the following research questions:

- RQ4. How accurate are the results provided by each heuristic?
- RQ5. How accurate is the identification of Truck Factor developers by each heuristic?
- RQ6. What is the correlation between Truck Factor sets and Core Developers sets?

For purposes of comparison, the results of the AVL Algorithm were included in RQ4 and RQ5 because this algorithm presented the best results in estimating Truck Factors among the algorithms evaluated in the previous study (Chapter 5).

6.2 Results

In this section, we describe the results of the study performed to compare Truck Factor and Core Developers.

RQ4. How accurate are the results provided by each heuristic?

To answer research questions RQ4 and RQ5, we used the oracle of Truck Factors and developers of 35 open-source systems (Section 5.2). First, we generate, for each of these systems, the Core Developers set according to Commit-Based Heuristic and LOC-Based Heuristic. After obtaining the developers' sets, we calculate the error, precision, recall and F-measure of the heuristics in relation to the oracle of Truck Factors. The results of precision, recall, and F-measure were calculated as described in Section 5.3. The error metric is the absolute value of the difference between the number of developers indicated by the Core Developers (CD) heuristic and the number of Truck Factor developers present in the oracle, as shown in Equation 6.1.

$$error = CD_{heurisitc} - TF_{oracle} \tag{6.1}$$

Figure 6.1 shows violin plots with the absolute error measures. For the Commit-Based Heuristic, the first quartile is 1, the median is 2, and the third quartile is 9.5. The largest error presented by this heuristic is 56. For the LOC-Based Heuristic, the first quartile, the median, and the third quartile are 0, 1 and 2, respectively. The maximum error presented by this heuristic is 14. For the AVL algorithm, the first quartile is 0, and the median and the third quartile are 1. The maximum error presented by this algorithm is 11.

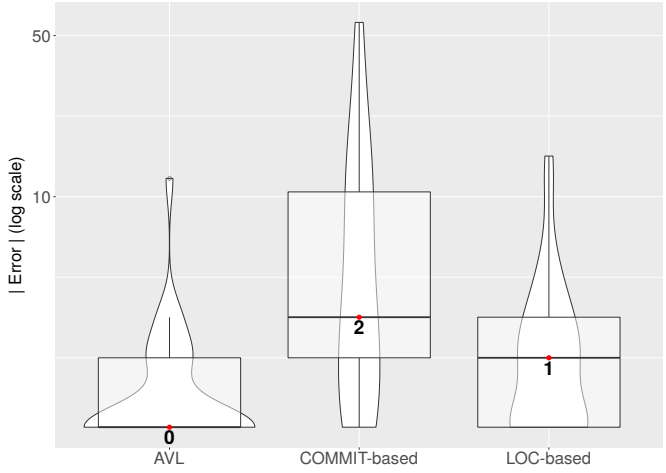


Figure 6.1: Error

From this result, we observe that LOC-Based is the heuristic whose error distribution most closely resembles the AVL algorithm, presenting better results than Commit-Based Heuristic. However, the AVL algorithm continues to be the most appropriate for the detection of Truck Factors, since it presents the best accuracy among the algorithms proposed and analyzed in this work.

Table 6.1 shows how the heuristics and the AVL algorithm perform for groups of systems with different TF results. As shown, AVL is the most accurate algorithm, identifying the TF correctly for 100% of systems in TF1 group, 30% of systems in TF2-5, and 50% of systems in the TF2-5 group. The LOC-Based Heuristic has the second best result, being able to identify 40% of the systems in TF1 group, 3% (i.e., one system) in TF2-5. Commit-Based Heuristic has the worst results. For TF1, the heuristic identified the results correctly for only 17% of the systems evaluated in TF1, and 6% (i.e., two systems) in TF2-5. Both heuristics were not able to correctly identify the TF results for systems with Truck Factor equal or greater than 6 (TF6+).

Group	**Range**	**Size**	**Alg**	%
TF1	$TF = 1$	20	AVL	100
			COMMIT	17
			LOC	40
TF2-5	$2 \leq TF \leq 5$	13	AVL	30
			COMMIT	6
			LOC	3
TF6+	$TF \geq 6$	2	AVL	50
			COMMIT	0
			LOC	0

Table 6.1: Percentage of results with *error* = 0 per group of systems. Size is the number of systems in each group.

RQ5. How accurate is the identification of TF developers by each heuristic?

Figure 6.2 shows violin plots with the precision the results. The Commit-Based Heuristic presents the worst results: the first quartile, the median and the third quartile of this heuristic are 0.15, 0.49, and 0.9, respectively. For LOC-Based Heuristic the first quartile is 0.33, the median is 0.67, and the third quartile is 1. Therefore, this heuristic presents slightly better results than Commit-Based. For AVL algorithm the first quartile, the median, and the third quartile are 1.

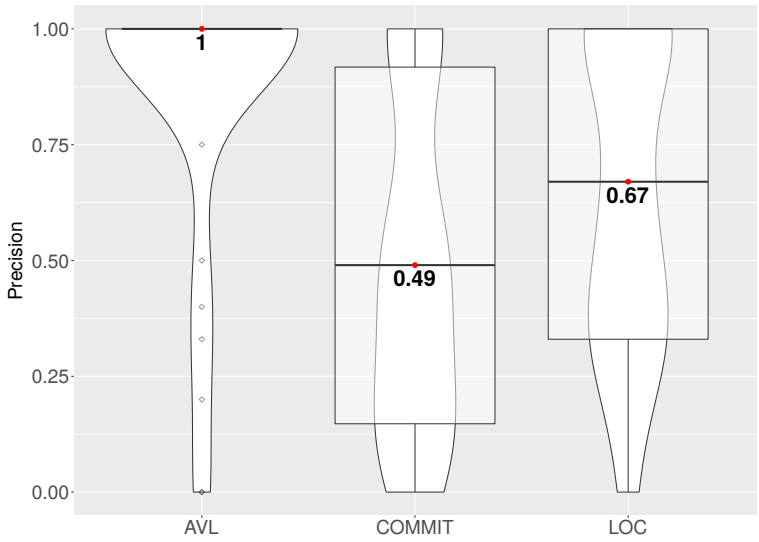


Figure 6.2: Precision

Recall results are shown in Figure 6.3. The first quartile of AVL algorithm, Commit-Based Heuristic, and LOC-Based Heuristic are, respectively, 0.54, 1, and 0.76. The median and the third quartile are 1 for AVL, Commit-Based and LOC-Based Heuristic.

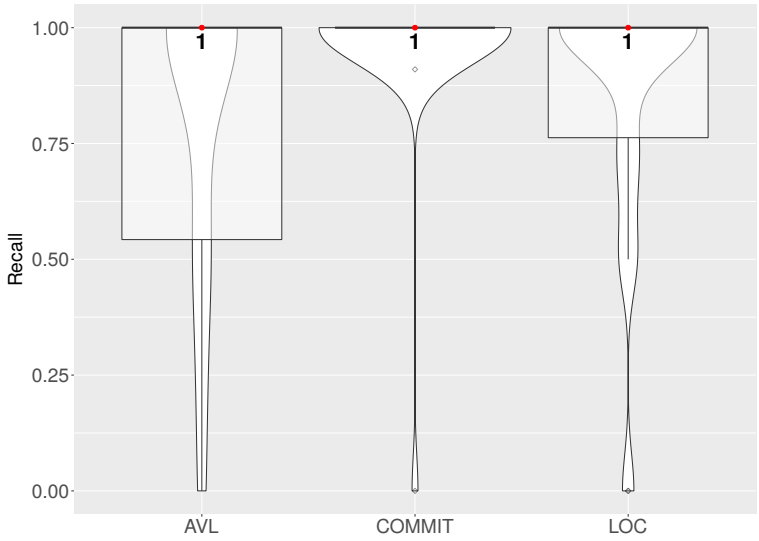


Figure 6.3: Recall

Figure 6.4 shows the F-measure results. For the AVL algorithm, the first quartile is 0.70, the median and the third quartile are 1. For the Commit-Based Heuristic, the first quartile is 0.54, the median is 0.68, and the third quartile is 0.9. The LOC-Based

Heuristic presents better results than the previous one; its first quartile is 0.5, the median is 0.76, and the third quartile is 1.

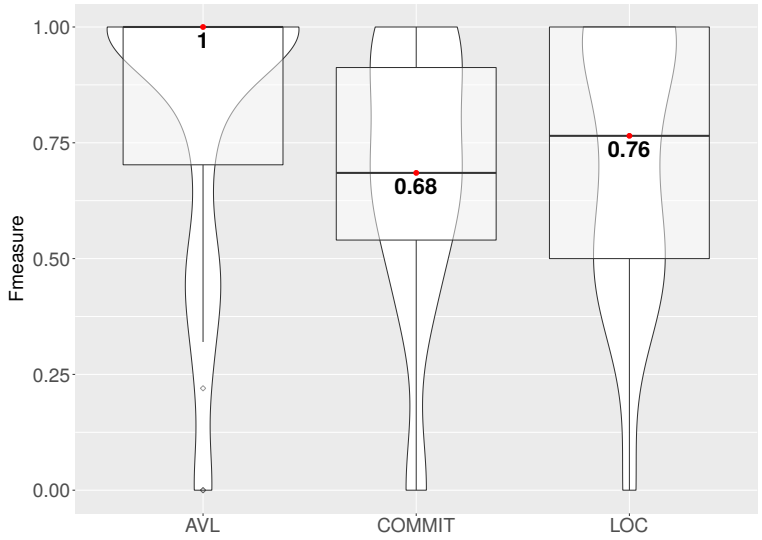


Figure 6.4: F-measure

Summary: Among the heuristics evaluated, LOC-Based is the one that presents the best results. Considering the obtained results, we can observe that although they can be considered as related concepts, the algorithms for estimating Truck Factors and the heuristics for identifying Core Developers present different results.

RQ6. What is the correlation between Truck Factor and the Core Developers sets?

The results of the investigation of RQ5 indicated that, although similar, Truck Factor and Core Developers are different concepts. This research question aims to better identify the relation between these concepts. To answer this research question we identify the *True Positive* (TP), *False Positives* (FP) and *False Negatives* (FN) regarding the sets of Truck Factor and Core Developers. A true positive occurs when a developer is both in the Truck Factor set and in the core developer set. A false positive occurs when the core developer algorithm indicates a developer that is not part of the Truck Factor set in the oracle. In contrast, a false negative indicates that the algorithm did not find a developer that belongs to the Truck Factor oracle. Thus, the relationship between the two sets can take the following forms:

- **True Positive > 0 (TP1+)**
  The heuristic algorithm in this case reports developers in common with the Truck Factor set.

  * **FP = 0 and FN = 0**
    The absence of False Positives (FP0) and False Negatives (FN0) indicates that the heuristic algorithm did not provide incorrect results, i.e., the core developer set and the Truck Factor set are equal.

  * **FP = 0 and FN ≥ 1**
    The presence of False Negatives (FN1+) indicates that the heuristic has failed to detect developers that are in the Truck Factor set. The absence of False Positives (FP0) indicates that there is no overestimation of developers by the heuristic. These two situations indicate that the set of developers reported by the heuristic is smaller and it is contained within the Truck Factor set. That is, the set of Core Developers is a subset of the oracle of Truck Factor.

  * **FP ≥ 1 and FN = 0**
    The absence of False Negatives (FN0) shows that the heuristic indicated all developers that are in the oracle. The presence of False Positives (FP1+) indicates that heuristics reported more developers than those in the Truck Factor set. That is, the core developer set contains all the developers that are in the Truck Factor set, but it is larger than the Truck Factor set. In this case, the Truck Factor is a subset of Core Developers.

  * **FP ≥ 1 and FN ≥ 1**
    The presence of false positives (FP1+) and false negatives (FN1+) indicates that heuristics fail to identify Truck Factor developers and also introduced developers who are not part of this set. In this case, there is an intersection between the two sets.

- **True Positive = 0 (TP0)**

  In this case, the algorithm does not report developers in common with the Truck Factor set. The absence of true positives (TP0) indicates that the sets are distinct.

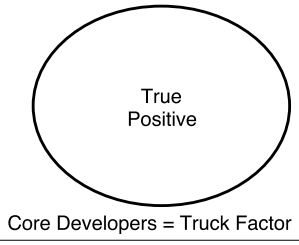
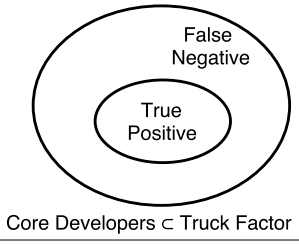
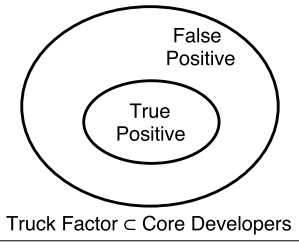
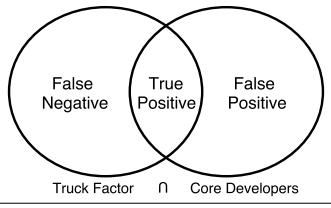
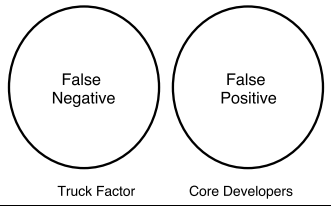
			Venn Diagram	Heuristic Results	
				Commit-Based	LOC-Based
	FP0	**FN0**		9	13
TP1+	**FP0**	**FN1+**		0	0
	FP1+	**FN0**		23	11
	FP1+	**FN1+**		1	8
TP0	———	———		2	3

Table 6.2: Summary of intersection results.

Table 6.2 presents the results for each of aforementioned cases. Among the 35 systems evaluated, there were no cases, for both heuristics, in which it was possible to observe the existence of Core Developers as a subset of Truck Factor (row TP1+, FP0, FN1+). The LOC-Based Heuristic presents the following results for the other cases: 13 systems are FP0 and FN0, 11 systems are FP1+ and FN0, and eight systems are FP1+ and FN1+. These results indicate that in 37% of the systems the heuristic found exactly the same sets of the Truck Factor oracle; 31% of the Truck Factor sets are subsets of Core Developers; and in 23% of the evaluated systems there is an intersection between the two sets.

The results obtained for the Commit-Based Heuristic are significantly different from the results of LOC-Based Heuristic. In 9 systems (26%), the Commit-Based Heuristic showed that Core Developers and Truck Factor are the same set (FP0 and FN0). In only one system (3%) it was found a small intersection between the sets (FP1+ and FN1+). In 23 systems (66%), it was found that Truck Factor is a subset of Core Developers. The LOC-Based and Commit-Based Heuristics present completely different results from Truck Factor set (TP0) in only 2 (5%) and 3 (9%) cases, respectively.

Summary: The results of this study indicate that, in general, the Truck Factor is a subset of Core Developers. In 94% and 64% of the systems evaluated by the Commit-Based and LOC-Based Heuristics, respectively, the Truck Factor set is the same, or it is contained in the Core Developers set.

6.3 Discussion

Core developers is a concept used to designate the developers who play a critical role in software development. Therefore, this concept is directly related to the Truck Factor metric. In this study, we compared an algorithm for estimating Truck Factor and two heuristics developed for the detection of Core Developers. In particular, we investigated: (i) whether heuristics for the detection of Core Developers are also capable of correctly identifying the Truck Factor developers of a system; and (ii) whether Truck Factor developers are a subset of the Core Developers.

Among the heuristics evaluated, LOC-Based Heuristic is the one that presents the best performance in identifying the Truck Factor developers. When answering RQ4, it was possible to observe that the error of LOC-Based Heuristic is lower than the error of the Commit-Based one. In RQ5, we observe that LOC-Based Heuristic features better accuracy, lower recall, and better F-measure than Commit-Based Heuristic. However, the heuristic for Core Developers detection that obtained the best performance in the detection of Truck Factor was not able to present results as good as those presented by the algorithm AVL. This fact indicates that although similar, Truck Factor and Core Developers are different concepts. Understanding how these two concepts are related was the goal of RQ6. The results presented indicate that, in all studied systems, there is an intersection between Truck Factor developers and Core Developers. In addition, we could observe that in most cases the Truck Factor set is a subset of Core Developers.

6.4 Final Remarks

The results reported in this chapter indicate that, although related, Truck Factor and Core Developers are different concepts. This is evidenced by the better results presented by the AVL algorithm when contrasted with the ones produced by the Commit-Based and LOC-Based heuristics. In addition, there are indications that Truck Factor developers are a subset of the Core Developers of a system. The cases in which both sets are the same occur only in systems that have Truck Factor equal to 1. In these systems, the concentration of knowledge is so high that the concepts converge.

Chapter 7

Knowledge Distribution in Software Architectures

This chapter presents an empirical study that investigates whether the set of developers indicated by a Truck Factor algorithm are those who have knowledge of the most complex components of a system. Software architecture is represented by means of the Little House model, described in Section 2.3. The study design is described in Section 7.1. Section 7.2 presents the results. Threats to validity are presented in Section 7.4. Finally, discussion and final remarks are presented in Sections 7.3 and Section 7.5.

7.1 Study Design

The dataset considered in this chapter was the one proposed by Avelino et al. [2016]. It consists of 133 most popular GitHub systems, distributed in six programming languages: Java Script, Python, Ruby, C/C++, Java, and PHP. The systems present in this dataset have significant numbers of files and developers, and also have significant development history.

The systems analyzed in this work meet the following criteria: (i) they are written in Java; (ii) they have their bytecodes available or there are mechanisms that facilitate the compilation of the code to obtain it, i.e., the *pom.xml* file, since the compilation of systems with many external dependencies is not a trivial task [Terra et al., 2013]. These rules serve the operating premises of the Connecta tool [Ferreira, 2011], which is responsible for generating the Little House model data. The model represents the macroscopic and generic visualization of the software in an automatic way, not requiring intervention of the architects of the analyzed systems. After applying this criteria,

we selected four out of 22 Java systems available in the original dataset: *Android Annotations*[1] (an Android framework); *Dropwizard*[2] (a web framework); *Titan*[3] (a graph database); and *Glide*[4] (a dependency manager).

The Truck Factor (TF) results were obtained by the Truck-Factor tool [5] [Avelino et al., 2016]. With this tool, the following data were collected: TF value of the system, set of developers responsible to the collected TF; and the list of files in which the TF developers have authorship on. The Connecta tool[6] was used to obtain data concerning the Little House model [Ferreira, 2011]. From the system's bytecodes, Connecta provides a list containing the system classes and components to which they belong. As the last step, a Java procedure was implemented, which joins the data provided by the Truck Factor and Connecta tools. The script also computes the percentage of classes that a developer has in a component.

7.2 Results

In this section, we present and analyze the results obtained for the four investigated systems. The data are presented in tables that report the total classes per component (Classes / Comp.); the total of classes that each author has in each component of the Little House (#Classes); and the percentage that this number means regarding the whole classes of the component. The acronym *TFDev#* will identify the authors responsible for the TF results.

7.2.1 Android Annotations

The *Android Annotations* system is a framework for Android development, with a Truck Factor of 2. TFDev1 and TFDev2 are the developers rated as critical in this system, and they own, respectively, 24.81% and 15.76% of the system's Java files. Table 7.1 presents the results for *Android Annotations*.

Although TFDev1 is the developer whose leave may have the greatest impact on the system, he does not have the higher percentage of authorship in the most critical components of the system. As we can see, TFDev2 has a higher proportion of authorship in two of the three most critical components of the system (LSCC and In).

[1] https://github.com/androidannotations/androidannotations
[2] https://github.com/dropwizard/dropwizard
[3] https://github.com/thinkaurelius/titan
[4] https://github.com/bumptech/glide
[5] https://github.com/aserg-ufmg/Truck-Factor
[6] http://homepages.dcc.ufmg.br/~kecia/connecta.htm

		TFDev1		TFDev2	
Component	**Classes/Comp.**	**#Classes**	**%Authorship**	**#Classes**	**%Authorship**
LSCC	10	1	10%	9	90%
In	161	3	2%	89	55%
Out	10	2	20%	2	20%
Tendrils	71	10	14%	1	1%
Tubes	32	8	25%	1	3%
Disconnected	655	209	32%	46	7%

Table 7.1: *Android Annotations* results

Therefore, when considering the importance of the classes, the developer whose leave would cause the greatest impact is TFDev2.

7.2.2 Titan

Titan is a graph database, optimized for storing and querying graphs with billions of vertices and edges. This system has a Truck Factor of 2, with TFDev3 having 10% of the Java files and TFDev4 having 39%. As reported in Table 7.2, this system has characteristics similar to *Android Annotations*. Although TFDev3 is indicated as the developer with the greatest impact on the system, the data suggests that TFDev4 has a higher percentage of authorship in the most critical components of *Titan*.

		TFDev3		TFDev4	
Component	**Classes/Comp.**	**#Classes**	**% Authorship**	**#Classes**	**%Authorship**
LSCC	74	1	1%	50	68%
In	214	66	31%	108	50%
Out	402	20	5%	163	41%
Tendrils	277	25	9%	48	17%
Tubes	57	4	7%	19	33%
Disconnected	371	27	7%	160	43%

Table 7.2: *Titan* results

7.2.3 Dropwizard and Glide

Both *Dropwizard* and *Glide* systems have a Truck Factor of 1. Tables 7.3 and 7.4 present the results of the systems. Different from the results obtained in the systems described in Section 7.2.1 and 7.2.2, in systems with a minimal value of TF, the developers indicated as part of the Truck Factor set have the highest percentage of authorship in the most complex components of the system. In *Dropwizard*, developer TFDev5 has the highest percentage of authorship in the LSCC and In components and 25% of the Out component classes. In the *Glide* system, the Truck Factor developer, TFDev6,

has a significant percentage of authorship on the three most critical components of the system.

		TFDev5	
Component	**Classes/Comp.**	**#Classes**	**%Authorship**
LSCC	6	5	83%
In	46	24	52%
Out	106	40	38%
Tendrils	406	112	28%
Tubes	47	13	28%
Disconnected	293	89	30%

Table 7.3: *Dropwizard* results

		TFDev6	
Component	**Classes/Comp.**	**#Classes**	**% Authorship**
LSCC	32	23	72%
In	5	2	40%
Out	220	126	57%
Tendrils	40	19	48%
Tubes	1	0	0%
Disconnected	93	44	47%

Table 7.4: *Glide* results

7.3 Discussion

In this chapter, we investigated how the knowledge of the developers that compose the system Truck Factor set is distributed in the system architecture. Specifically, we verified if these developers are in fact the ones that have knowledge of the most critical components of the system. For this purpose, the architecture of four Java systems was represented by the Little House model.

According to the study results, there are indications that the importance of the classes in a system is a factor to be considered in the computation of Truck Factors. As observed in systems with Truck Factor of 2, the developers having the second largest impact were those that have the highest percentage of authorship in the most critical system components. However, the same assertion is not valid for projects whose Truck Factor is 1. In these systems, the single Truck Factor developer is the one with the highest authorship on the most critical architectural components.

The main conclusion to be drawn from this study is that the developers responsible for the most critical architectural components are indicated among the developers listed by the Truck Factor. Therefore, the analysis performed by the Truck Factor needs to be improved so that it can more accurately capture the influence of the developers on the project, especially in architectural issues.

7.4 Threats to Validity

Four systems developed in Java were used to carry out the study reported in this chapter. Therefore, it is not possible to state that the presented results will be the same for other projects. To get around this issue, we used important systems of high complexity and different domains.

In this work, we consider only the Java files of the evaluated systems. However, it is common to have files with other extensions in a system. Thus, there may be other as important files as the considered Java files, but they were not included in the analysis.

7.5 Final Remarks

The existing algorithms for calculating Truck Factor do not consider the relative importance of the classes of a system. This chapter aimed to identify how the knowledge of the developers, as expressed by Truck Factor, is distributed in the system architecture. To this purpose, four case studies were carried out. It investigated if the developers pointed as Truck Factor are those that have the greatest percentage of authorship on the most critical components of the system architecture. The system architecture was represented by means of the *Little House* model. Our results show that the analysis of the Truck Factor associated with the importance of the class from the architectural point of view may reveal more accurate information about the influence of the developers on the system.

Chapter 8

Conclusion

In modern software projects, it is crucial to have reliable data on how source code knowledge is distributed among the team members. This information can help to avoid "islands of knowledge" and to prevent the risks associated to the loss of key developers. Truck Factor is a key measure proposed to estimate such risks. In this context, this master dissertation assessed the distribution of knowledge in software projects through an in-depth analysis of the Truck Factor metric. To accomplish that, we carried out three empirical studies, whose summary and results are presented next.

Study #1: Comparison of Algorithms for Computing Truck Factor

The first study was a comparative study of three algorithms to estimate Truck Factor. We built an oracle of Truck Factor, gathered via a survey with 35 open-source project teams. By comparing the accuracy, precision, recall, and F-measure results of the algorithms in relation to the oracle of Truck Factors, it was possible to establish that AVL is the algorithm that presents the best results for Truck Factor developers estimation. Besides that, we also analyzed the algorithms when their thresholds are modified and we concluded that: (i) the original threshold (50% of abandoned files) proposed by Avelino et al. [2016] has the best results in identifying Truck Factor developers, producing an accurate result in 68% of the evaluated systems; (ii) the CST Algorithm showed its best results when using the MCEC (*multiple changes equally considered*), presenting $error = 0$ in 24 out of 35 systems; and (iii) RIG Algorithm has a non-deterministic behavior, and increasing the number of tested samples does not have a positive impact in correctly estimating of Truck Factor developers.

Study #2: Truck Factor vs. Core Developers

The second study, investigated the relationship between the concepts of Truck Factor and core developers. We analyzed two heuristics proposed by Yamashita et al. [2015] to infer core developers were analyzed their accuracy regarding the oracle of Truck Factor. Considering the results, we found evidences that, although Truck Factor and core developers are related concepts, algorithms for estimating Truck Factor and the heuristics for identifying core developers present different results. Through the analysis of the presence of true positives, false positives and false negatives in the heuristics results, we provide evidence that Truck Factor developers are in most cases a subset of core developers.

Study #3: Knowledge Distribution in Software Architectures

In the last study, we aimed to check whether the key developers of a systems are those who have knowledge in the most critical architectural components. The software architecture was represented by means of the Little House model. To accomplish our goal, we carried out four case studies. For systems with Truck Factor of 2, we observed that the developers considered by the Truck Factor algorithms as the most critical one, in fact has the second highest percentage of authorship in classes from the most critical system components. For systems with Truck Factor equal to one, the developer indicated as the system Truck Factor, has the dominant knowledge in the most critical system components. These results indicated that the relevance of classes should be considered when estimating Truck Factor.

In conclusion, we find that AVL is the best algorithm proposed so far to estimate Truck Factor. Nevertheless, it is possible that such estimation might be improved, for instance, by considering architectural characteristics of the software system. Therefore, we suggest the following future work:

- **Replication of our studies with proprietary software.** The studies presented in this dissertation were conducted using open-source projects. For this reason, it would be interesting to check out how the behavior of the Truck Factor metric performs with proprietary software environments.
- **Consider a time window for the artifacts used in the algorithms for estimating Truck Factors.** The algorithms evaluated in this master dissertation do not discard source files whose last commit happened a long time ago, i.e., commit in very stable files. In addition, they may not include developers who no longer belong to the project, i.e., developers that have not contributing to the system for a long time. Both aspects may have negatively impacted the algorithms results.

- **Consider social interaction as a source of developer knowledge.** Modifying the source code may be not the only way to gain knowledge about a project. The social relationship among developers may also be related to the source code knowledge. For example, observing how developers interact by contributing to email lists and issues can be an added factor to calculating Truck Factor.
- **Consider architectural characteristics of the software system.** The estimation of Truck Factor might be improved by considering the internal characteristics of the software system, such as the complexity of the classes of the system.

Bibliography

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10.

Bass, L., Clements, P., and Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley Professional.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code!: Examining the effects of ownership on software quality. In *19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 4–14.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Bowler, M. (2005). Truck factor. Online. Date Accessed: December 03, 2016.

Cosentino, V., Izquierdo, J. L. C., and Cabot, J. (2015). Assessing the bus factor of git repositories. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503.

Ferreira, K. A. M. (2011). *Um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos*. PhD thesis, Universidade Federal de Minas Gerais.

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012). A generic macroscopic topology of software networks - a quantitative evaluation. In *26th Brazilian Symposium on Software Engineering (SBES)*, pages 161–170.

Ferreira, M., Ferreira, K. A., and Neto, H. T. (2015). Mapping the potential change impact in object-oriented software. In *30th Annual ACM Symposium on Applied Computing (SAC)*, pages 1654–1656.

Foucault, M., Falleri, J., and Blanc, X. (2014). Code ownership in open-source software. In *18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–9.

Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology*, 23(2):1–42.

Fritz, T., Ou, J., Murphy, G. C., and Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity. In *32nd International Conference on Software Engineering (ICSE)*, pages 385–394.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3):1–42.

Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. (2017). Classifying developers into core and peripheral: An empirical study on count and network metrics. In *39th International Conference on Software Engineering (ICSE)*, pages 1–12.

Mens, T. (2016). An ecosystemic and socio-technical view on software maintenance and evolution. In *32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–8.

Mockus, A. (2010). Organizational volatility and its effects on software defects. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 117–126.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346.

Pressman, R. S. (2011). *Engenharia de Software - Uma Abordagem Profissional*. McGraw-Hill.

Rahman, F. and Devanbu, P. (2011). Ownership, experience and defects: A fine-grained study of authorship. In *33rd International Conference on Software Engineering (ICSE)*, pages 491–500.

Ricca, F. and Marchetto, A. (2010). Are heroes common in FLOSS projects? In *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–4.

Ricca, F., Marchetto, A., and Torchiano, M. (2011). On the difficulty of computing the truck factor. In *12th Product-Focused Software Process Improvement (PROFES)*, pages 337–351.

Rigby, P. C., Zhu, Y. C., Donadelli, S. M., and Mockus, A. (2016). Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. In *38th International Conference on Software Engineering (ICSE)*, pages 1006–1016.

Sommerville, I. (2011). *Engenharia de Software*. Pearson.

Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4.

Torchiano, M., Ricca, F., and Marchetto, A. (2011). Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold. In *2nd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 12–18.

Williams, L. and Kessler, R. (2003). *Pair Programming Illuminated*. Addison Wesley.

Yamashita, K., McIntosh, S., Kamei, Y., Hassan, A. E., and Ubayashi, N. (2015). Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *14th International Workshop on Principles of Software Evolution (IWPSE)*, pages 46–55.

Ye, Y. and Kishida, K. (2003). Toward an understanding of the motivation open source software developers. In *25th International Conference on Software Engineering (ICSE)*, pages 419–429.

Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V. R., and Schneider, K. (2010). Are developers complying with the process: an XP study. In *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10.