Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

SPL-SZZ - Uma implementação do algoritmo SZZ extensível

Adelson Jhonata Silva de Sousa
José Fortes Neto

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2022

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

SPL-SZZ - Uma implementação do algoritmo SZZ extensível

Adelson Jhonata Silva de Sousa
José Fortes Neto

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Walter Mendonça
Universidade de Brasília

Luís Amaral
Universidade de Brasília

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 3 de outubro de 2022

Dedicatória

Eu, José Fortes, dedico este trabalho à minha mãe Maria Dulcineide, que sempre me apoiou e fez tudo que estava a seu alcance para que eu pudesse chegar até aqui, a minha namorada Mariana Maciel, que acreditou no meu potencial e esteve presente na minha vida ao longo dos últimos 4 anos, sempre me ajudando a vencer os obstáculos presentes dentro e fora do curso. Dedico também a minha tia, Maria Dilma, que sempre quis o melhor para mim e a Fernando e Ana, os quais sempre acreditaram no meu potencial. Por fim, dedico este trabalho ao meu orientador Dr. Rodrigo Bonifácio, o qual me proporcionou oportunidades incríveis e que me permitiram avançar em minha carreira.

Eu, Adelson Jhonata, dedico este trabalho primeiramente a Deus, por nunca ter me desamparado em toda minha vida. Dedico também à toda a minha família, em especial à minha mãe Aldenir, ao meu irmão Marcos Felipe e ao meu avô Adelson que sempre me apoiaram nos momentos mais difíceis e por sempre acreditarem em mim. Por fim, dedico à minha melhor amiga, Maria Luísa por todo incentivo, risadas e companheirismo.

Também dedicamos este trabalho aos nossos amigos: Saulo, Daniel, Diego, Gabriel, Lucas, Matheus, Breno, João Victor, João Pedro e Maria que conhecemos ao longo desta jornada e nos ajudaram a chegar até o fim.

Agradecimentos

Agradecemos ao nosso orientador Dr. Rodrigo Bonifácio de Almeida por nos guiar ao longo deste trabalho, compreendendo e ajudando nas adversidades encontradas durante a elaboração desta pesquisa.

Resumo

Um problema comum que incide sobre todas as equipes de desenvolvimento desde o início do processo de codificação de um software é conseguir lidar com *bugs* e identificar os motivos comuns que levam a esse tipo de problema para que, assim, estes possam ser evitados. Esta é uma tarefa complexa, pois envolve a identificação dos trechos de códigos alterados responsáveis pela introdução do *bug*. A partir deste contexto, foi proposto o algoritmo SZZ, que tem como objetivo encontrar a revisão que acabou por introduzir o *bug* no código.

Existem diversas implementações do algoritmo SZZ, porém essas implementações são focadas em pesquisas acadêmicas e, por isso, possuem uma elevada complexidade de execução. Este trabalho apresenta uma nova implementação do algoritmo SZZ que tem por objetivo facilitar o uso deste algoritmo pela comunidade, através de um software que possa ser facilmente executado e que tenha uma boa confiabilidade. Para isso, ele conta com a utilização da linguagem Rust e uma arquitetura pensada em extensividade.

Palavras-chave: SZZ, RA-SZZ, SPL-ZZ, bug, software, Rust, git

Abstract

A common problem that affects all development teams from the beginning of the software coding process is dealing with *bugs* and identifying the common reasons that lead to this type of problem so that they can be resolved. avoided. This is a complex task, as it involves identifying the changed code snippets responsible for introducing the *bug*. From this context, the SZZ algorithm was proposed, which aims to find the revision that ended up introducing the *bug* in the code.

There are several implementations of the SZZ algorithm, but these implementations are focused on academic research and, therefore, have a high execution complexity. This work presents a new implementation of the SZZ algorithm that aims to facilitate the use of this algorithm by the community, through a software that can be easily executed and that has good reliability. For this, it relies on the use of the Rust language and an architecture designed for extensibility.

Keywords: SZZ, RA-SZZ, SPL-ZZ, bug, software, Rust, git

Sumário

1 Introdução **1**

1.1 Objetivos 2

1.2 Metodologia 3

1.3 Organização 3

2 Revisão de Literatura e Trabalhos Relacionados **5**

2.1 Algoritmo SZZ 5

2.2 Implementações do SZZ 7

2.2.1 B-SZZ 7

2.2.2 AG-SZZ 8

2.2.3 MA-SZZ 8

2.2.4 RA-SZZ 8

2.2.5 L-SZZ 9

2.2.6 R-SZZ 9

2.3 Limitações do SZZ 10

2.4 Linguagem Rust 11

3 Arquitetura e Implementação **12**

3.1 Estrutura de módulos 12

3.2 Estrutura de entrada e saída 13

3.3 Grafo de anotação 15

3.3.1 Customização do Grafo de Anotações 16

3.4 Algoritmo Implementado 17

3.5 Execução 18

4 Resultados **20**

5 Conclusão **23**

Referências **25**

Lista de Figuras

2.1 Passos de execução SZZ Unleashed [1]. 10

3.1 Diagrama de módulos 12
3.2 Grafo de Anotações - Fonte: *Zimmermann et al*. [2] 15
3.3 Grafo de Anotações - Fonte: Fan *et al.* [3] 16
3.4 Diagrama de execução 18

Lista de Tabelas

4.1 Precisão 21
4.2 Resultados 21

Capítulo 1

Introdução

Uma característica comum que todas as equipes de desenvolvimento vão enfrentar e gastar muito tempo e esforço no processo de desenvolvimento de um *software* é tentar identificar, localizar e corrigir *bugs*. Corrigir *bugs* envolve isolar a parte do código que faz com que o programa se comporte de forma inesperada e fazer alterações nela para corrigir o *bug*. Essa é uma tarefa desafiadora, e os desenvolvedores geralmente gastam mais tempo corrigindo *bugs* e tornando o código mais sustentável do que desenvolvendo novos recursos [4, 5, 6]. Para diminuir o tempo gasto corrigindo *bugs*, é crucial entender melhor as práticas de desenvolvimento e as propriedades dos sistemas que são mais propensos a introduzir *bugs*.

Para estudar esta área, em 2005 os autores Jacek Śliwerski, Thomas Zimmermann e Andreas Zeller apresentaram um estudo [7] sobre o hoje conhecido como algoritmo SZZ, que visa detectar inserção de *bugs* em projetos e códigos, por meio:

1. Da análise das mudanças feitas ao longo do tempo;
2. Da utilização de programas de versionamento para percorrer os arquivos e linhas que foram alterados;
3. Da análise de todas as modificações feitas, sejam elas cosméticas, estruturais e/ou quaisquer outras mudanças executadas;

O algoritmo SZZ tem sido amplamente utilizado a fim de identificar mudanças introdutoras de *bugs* com fins científicos e acadêmicos, para que assim possam ser feitas análises empíricas de quando, como e por que(m) os *bugs* são introduzidos [2, 3, 8, 9, 10, 11]. Ao longo dos anos, os pesquisadores propuseram várias heurísticas para melhorar a precisão do algorítimo SZZ, proporcionando várias implementações diferentes do algoritmo [7, 12, 13, 14]. No entanto, essas implementações acabam por serem utilizadas apenas pela comunidade científica, pois possuem

alta complexidade para serem executadas e necessitam, muitas vezes, da instalação de várias ferramentas complementares no ambiente, obrigando o usuário ou a executar várias etapas ou a implementar alterações no código para que alcance resultados que possam vir a ser utilizados fora do meio científico.

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver uma versão especifica do SZZ que seja funcional, performática e principalmente de fácil instalação e utilização por parte da comunidade, de forma a contribuir para a distribuição e difusão das implementações do algoritmo SZZ. Para tanto, as seguintes atividades foram realizadas:

- Estudo teórico sobre o SZZ e as possíveis linguagens de programação a ser utilizadas no desenvolvimento do software.
- Desenvolvimento do software SZZ na linguagem escolhida na atividade anterior.
- Avaliar a implementação desenvolvida utilizando o *benchmark* disponível [15] e comparar os resultados obtidos pelo nosso algoritmo com os resultados de outras implementações disponíveis na comunidade.

Para atingir o objetivo geral deste trabalho de graduação, tivemos que avançar em uma série de objetivos específicos, organizados como etapas ou grandes marcos do trabalho. A primeira etapa da fase de desenvolvimento compreendeu o estudo teórico sobre o SZZ e suas peculiaridades, tais como o grafo de anotação e a análise das diferenças entre os arquivos utilizando o *diff*. Além disso, foram feitos um estudo e a seleção da linguagem a ser utilizada no desenvolvimento, devendo a linguagem escolhida ser uma que atenda aos objetivos do software listados nos objetivos gerais.

A segunda etapa consistiu no efetivo desenvolvimento e implementação do software proposto nesta pesquisa, onde tal implementação foi feita atentando-se a cada uma das regras e convenções observadas durante os estudos realizados na primeira etapa, afim de que o software aqui desenvolvido fosse equiparável a outros disponíveis na comunidade com o mesmo objetivo.

A terceira etapa do desenvolvimento teve como objetivo avaliar de forma qualitativa e quantitativa o software desenvolvido, utilizando como base o *benchmark* disponível [15] e observando os resultados obtidos aos executá-lo em nossa ferramenta. Estes resultados foram então comparados com os obtidos por outras ferra-

mentas ao executar o mesmo *benchmark*, afim de comparar o desempenho obtido por cada uma.

Por fim, o resultado da implementação e da avaliação devem ser entregues à comunidade, de forma pública para que todos tenham acesso e possam usar da maneira que quiserem.

1.2 Metodologia

Para atingir os objetivos descritos no tópico anterior, foi feito primeiramente um estudo focado no SZZ e em suas implementações, de modo a possibilitar a escolha consciente de qual implementação e quais características dessa implementação utilizar. A implementação escolhida foi a R-SZZ [16] utilizando como base o trabalho *Evaluating SZZ Implementations Through a Developer-informed Oracle* [17] que compara, entre as diversas implementações existentes, a performance e a taxa de acurácia. O R-SZZ usa o grafo de anotação como base, ignora alterações cosméticas e meta-alterações, considerando apenas como revisões introdutoras de *bugs* a última alteração que impactou uma linha alterada ao corrigir o erro.

Foi feito, também, um estudo focado nas diversas linguagens existentes no mercado a fim de selecionar a mais adequada para a implementação, focando sempre em performance, confiabilidade e facilidade de utilização para atingir o objetivo de ser um software performático e de fácil acesso pela comunidade. Dentre as diversas opções disponíveis no mercado, focamos apenas em linguagens compiladas e, ao mesmo tempo, modernas como *Rust* e *Go*. Por fim, a linguagem escolhida foi a *Rust*, por se destacar com seu foco em sistemas de baixo nível, performance e em segurança de memória [18].

Para avaliar a implementação do SPL-SZZ [19], foi utilizada uma base de dezenas de projetos e revisões disponíveis [15]. Essa base, que utiliza arquivos no formato *.json*, foi utilizada como arquivo de entrada do software, e foram comparadas as saídas geradas pela implementação com as saídas da base do *benchmark* de modo a avaliar o desempenho do software desenvolvido.

1.3 Organização

Este estudo está estruturado da seguinte forma:

- No Capítulo 2, está descrita a revisão de literatura, trabalhos relacionados e fundamentação teórica dos assuntos tratados neste trabalho.

- No Capítulo 3, está descrita a arquitetura geral e implementação do SPL-SZZ e o detalhamento de como tudo foi feito.
- No Capítulo 4, tem-se a descrição da avaliação e dos resultados da implementação desenvolvida.
- No Capítulo 5, tem-se a conclusão acerca deste trabalho.

Capítulo 2

Revisão de Literatura e Trabalhos Relacionados

2.1 Algoritmo SZZ

A primeira menção do assunto de detecção automática de inserção de *bugs* deu-se no ano de 2005 no trabalho intitulado "When Do Changes Induce Fixes?” [7] de autoria de Śliwerski, Zimmermann e Zeller, no qual é proposto um mecanismo/algoritmo que analisa e verifica os comentários das revisões, tentando encontrar padrões que remetem à correção de bugs. Com o auxilio de expressões regulares para essa verificação e com o uso do comando *git annotate*, o algoritmo percorre todas as revisões analisando os arquivos e linhas com o intuito de identificar quais foram modificadas e/ou excluídas na correção do bug e assim classificando a revisão como possível introdutora de bug. Esse algoritmo ficou conhecido como SZZ, refletindo as iniciais de seus criadores.

A primeira versão do algoritmo SZZ proposto por Śliwerski *et al.* [7] considera toda e qualquer mudança feita no código como possível introdução de *bugs*, isto é, comentários alterados, linhas e espaços em branco retiradas ou acrescentadas, indentação de código e quaisquer coisas do tipo que não alteram de fato o código.

Um ano após o primeiro trabalho notório sobre o assunto, em 2006, foi lançado o estudo "Automatic Identification of Bug-Introducing Changes” de autoria de Kim *et al.* [12] que em seu conteúdo trás melhorias para o algoritmo SZZ original proposto com o intuito de aumentar a performance e a acurácia. O algoritmo SZZ proposto por Kim *et al.* [12] desconsidera qualquer alteração não semântica como introdução de *bugs*, isto é, ignora quaisquer alterações nos comentários de código e linhas e/ou espaços em branco como candidatos a *commits* indutores de *bugs*, além de utilizar

grafos de anotações para acompanhar as evoluções das mudanças linha a linha ao longo do desenvolvimento.

Em 2008, os autores Chadd Williams e Jaime Spacco lançaram o estudo "SZZ Revisited: Verifying When Changes Induce Fixes" [13], em que abordaram melhorias e críticas sobre as ideias anteriores do algoritmo SZZ. Entre as abordagens feitas, os autores criticaram o grafo de anotação proposto por Kim *et al.* [12], pois em seus estudos observou-se que, para blocos de mudanças muito grandes, o grafo de anotações não conseguia lidar muito bem. Para solucionar tal problema e se baseando no trabalho de Canfora *et al.* [20], propuseram uma nova estrutura para acompanhar a evolução das mudanças feitas ao longo do desenvolvimento. A estrutura proposta para substituir o grafo de anotações foi o mapeamento do número de linhas que tem como objetivo rastrear linhas únicas que evoluem a cada revisão ao longo do ciclo de vida do desenvolvimento.

Davies *et al.* [14], no ano de 2014, em seu estudo "Comparing text-based and dependence-based approaches for determining the origins of *bugs*."[14] analisaram a implementação original do SZZ proposta por Śliwerski *et al.* [7] e uma abordagem baseada em dependência para identificar mudanças introdutoras de *bugs*. Para avaliar a precisão das abordagens descritas, os autores registraram manualmente a origem dos *bugs* em três grandes projetos. De acordo com a avaliação que foi realizada, os autores, propuseram as possíveis seguintes melhorias para o SZZ:

- Considerar o impacto quando adicionado código em alteração de correção, sempre observando ao redor do bloco de código em que foi feita essa adição.
- Considerar apenas a revisão mais recente introdutora de correção como a revisão que realmente introduziu o *bug*.
- Ou considerar apenas a revisão com maior número de linhas modificadas como a revisão que realmente introduziu o *bug*.

Em 2016, Da Costa *et al.* [9] lançaram o estudo "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes" [9] em que os autores avaliaram cinco implementações distintas do SZZ e identificaram que essas implementações marcavam revisões que continham meta-alterações, ou seja, alterações de *branches e/ou merges* como possíveis introdução de *bugs*. No entanto, essas chamadas meta-alterações não podem ser consideradas alterações de introdução de *bug*, pois não alteram o comportamento do software desenvolvido e por isso devem ser descartadas. Os autores então propuseram uma nova implementação com o intuito de aprimorar o algoritmo SZZ, a qual ficou conhecida como MA-SZZ (*Meta-Changes Aware SZZ*) e teve como melhorias o mapeamento

de alterações no código e o descarte de meta-alterações. A próxima seção resume algumas implementações do algoritmo SZZ.

2.2 Implementações do SZZ

Ao longo dos anos, o algoritmo SZZ ganhou diversas implementações [17, 9, 8, 10, 3] e cada versão contava com características próprias com o intuito de melhorar a acurácia do algoritmo em si. Entre as implementações e soluções de melhorias desenvolvidas destacam-se:

2.2.1 B-SZZ

O B-SZZ vem de *Basics SZZ* sendo a primeira implementação do algoritmo SZZ proposto por Sliwerski *et al.* [7]. O algoritmo B-SZZ é divido em duas etapas essenciais para seu funcionamento: Identificação de correções de *bugs* e Identificação das alterações indutoras de *bugs*.

Identificação de correções de *bugs*

Na primeira etapa, o algoritmo, com auxilio de um sistema de controle de versão e uma base de registro de *bugs*, percorre todos os registros, verificando um por um, identificando e armazenando quais estão marcados como corrigidos. Desenvolvedores têm a convenção de nas revisões colocar mensagens que identifiquem o que aquela revisão em especifico está fazendo. Com a lista de registros de *bugs* corrigidos e com auxílio de regras de expressões regulares, o próximo passo é identificar em quais revisões o *bug* foi resolvido, a partir das mensagens em cada revisão. A dificuldade dessa primeira etapa se dava em razão das ferramentas utilizadas no processo de desenvolvimento e no registro de *bugs*. Atualmente, porém, esta etapa é mais fácil e confiável, uma vez que a maioria dos projetos usam sistemas de rastreamento de *bugs*, os quais mantêm um link entre cada *bug* e as revisões que os resolvem.

Identificação das alterações indutoras de *bugs*

Na segunda etapa do algoritmo, o objetivo é determinar quais revisões introduziram um *bug* que foi corrigido pela revisão de correção apontada. Nessa etapa, o algoritmo B-SZZ recebe como entrada uma revisão de correção de *bug* e a lista de revisões identificadas na etapa anterior. Com o auxílio do comando *annotate*

do sistema de controle de versão utilizado, para cada linha na revisão de correção de *bug*, o algoritmo encontra a revisão mais recente que modificou a linha analisada e que seja cronologicamente anterior a correção do *bugs*. Como uma revisão pode conter mais de uma linha e cada linha pode ter sido modificada em diferentes revisões anteriores, várias revisões podem ser selecionadas como introdutores de *bug*.

O algoritmo B-SZZ por ser o mais básico não desconsidera alterações não-semânticas na sua avaliação, isto é, todas as alterações cosméticas como: identação, comentários e espaços em brancos e qualquer outra coisa que não interfira de fato no código alterado é contada na avaliação de possível introdutor de *bugs*, fazendo assim com que o B-SZZ tenha um alto número de revisões apontadas como introdutores de *bugs*.

2.2.2 AG-SZZ

O algoritmo AG-SZZ vem de *Annotation Graph SZZ* e foi implementado seguindo a base do B-SZZ, descrito na seção 2.2.1, com o acréscimo das melhorias propostas por Kim *et al.* [12]. O AG-SZZ utiliza o grafo de anotação, que será explicado em 3.3, para armazenar e comparar a evolução das linhas da revisão. Assim, é feita uma busca em profundidade no grafo de anotação para encontrar as possíveis revisões de introdução de *bug*. Com as mudanças implementadas, o AG-SZZ reduz os falsos positivos, ignorando as alterações de formatação, as alterações nos comentários, linhas em branco e indentação.

2.2.3 MA-SZZ

O algoritmo MA-SZZ foi construído a partir do AG-SZZ, cuja sigla deriva de *Meta-Changes Aware SZZ* e foi proposto por Da Costa *et al.* [9]. Essa implementação tem como característica principal a remoção de meta-alterações que são ocasionadas quando se tem alterações de *branchs* e/ou *merges*. Por ser construído a partir do algoritmo AG-SZZ, o MA-SZZ também utiliza o grafo de anotações para representar a evolução das linhas modificadas e encontrar as possíveis revisões introdutoras de *bugs*.

2.2.4 RA-SZZ

O algoritmo RA-SZZ, cuja sigla vem de *Refactoring Aware SZZ*, foi proposto por Neto *et al.* [11] e foi construído a partir do MA-SZZ. Essa implementação tem como

objetivo principal lidar com o impacto das linhas de refatoração no código e sua principal característica é a incorporação do *RefDiff* no fluxo de execução do MA-SZZ. O RefDiff é uma ferramenta de detecção de refatoração para código escrito especificamente na linguagem Java.

A ferramenta RefDiff encontra relacionamentos entre elementos de código de duas revisões do projeto. Os relacionamentos indicam que ambos os elementos são iguais ou que foi aplicada uma operação de refatoração os envolvendo.

O RA-SZZ utiliza o RefDiff para detectar linhas de refatoração. As linhas de refatoração não são consideradas como linhas com erros e por isso não devem entrar nas possíveis revisões introdutoras de *bug*. O RA-SZZ não para nas alterações envolvendo linhas de refatoração e rastreia ainda mais o histórico do código para identificar as alterações reais de introdução de *bugs*.

2.2.5 L-SZZ

O algoritmo L-SSZ tem como base o B-SZZ e implementou as melhorias propostas por Davies *et al.* [14]. Foi construído a partir do algoritmo MA-SZZ e também utiliza o grafo de anotações para representar a evolução das modificações das linhas ao longo das revisões e desenvolvimento do *software*.

Como o algoritmo MA-SZZ pode retornar várias revisões introdutores de *bugs*, o algoritmo L-SZZ funciona como um filtro dos resultados retornados e com suas heurísticas aplicadas retorna a revisão com o maior número de linhas adicionadas, excluídas e/ou alteradas.

2.2.6 R-SZZ

O algoritmo R-SZZ segue a linha de melhorias propostas por Davies *et al.* [14]. A única diferença para o algoritmo L-SZZ é que, enquanto o L-SZZ retorna a revisão com o maior número de linhas adicionadas, excluídas e/ou alteradas, o R-SZZ retorna a revisão mais recente entre as possíveis revisões introdutoras de *bugs*.

Dentre as diversas implementações do algoritmo SZZ, escolhemos uma em específico para fazer a nossa implementação a partir dela, a qual é tema do presente trabalho. A implementação escolhida foi a R-SZZ, proposta por Davies *et al.* [14], a qual tem como característica principal a utilização de um grafo de anotação para representar a evolução das linhas modificadas. A escolha de implementar o algoritmo a partir do R-SZZ se deu através do estudo teórico dos algoritmos SZZ com base em vários artigos disponíveis na comunidade científica, em especial, com base no artigo *Evaluating SZZ Implementations Through a Developer-informed Oracle*

[17], o qual apresenta um estudo que compara diversas implementações do algoritmo SZZ e a implementação que melhor se apresentou nos resultados foi a R-SZZ, pois demonstrou ser a mais precisa.

2.3 Limitações do SZZ

O principal problema das implementações citadas na seção 2.2 é que são protótipos cuja finalidade é apenas o estudo científico do algorítimo SZZ e de suas várias implementações, impedindo que este seja amplamente adotado pela comunidade. Problemas como *bugs*, necessidade de inúmeras configurações e também códigos que necessitam de modificações tornam o processo de execução difícil e oneroso. São alguns exemplos de código de difícil execução:

- RA-SZZ, MA-SZZ, LA-SZZ e R-SZZ - Estas implementações dependem da instalação e configuração de banco de dados PostgreSQL. É necessária a instalação do Java SDK e do programa *GIT*, além de não possuírem ferramentas que permitam inserir os dados no banco de forma a possibilitar que o programa fosse executado, sendo necessária a criação de *scripts* para esse processo ser feito de forma automatizada.
- SZZ Unleashed: Esta implementação está dividida em uma série de programas menores, cada um responsável por uma parte da execução do algoritmo. Possui uma configuração de ambiente complexa e exige que o usuário instale as ferramentas Java SDK e Python, além do software programa *GIT*. Seu fluxo de execução está ilustrado na Figura 2.1.

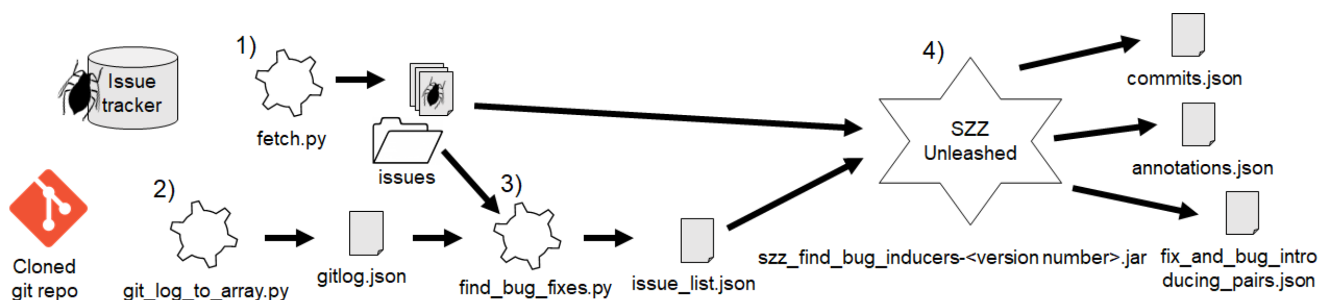

Figura 2.1: Passos de execução SZZ Unleashed [1].

2.4 Linguagem Rust

Rust [18] é uma linguagem de programação compilada e feita com foco em diferentes tipos de aplicações, como ferramentas de linhas de comando, ferramentas pra devops, aplicativos de IoT - Internet of Things, sistemas embarcados, web services e muito mais. Tem como principais características o desempenho e a confiabilidade, sendo uma linguagem extremamente rápida e com um eficiente sistema de gerenciamento de memória, sem um coletor de lixo, além de contar com um rico sistema de tipagem estática. Por esse motivo foi decidido pelo uso de Rust para a implementação do SPL-SZZ.

Capítulo 3

Arquitetura e Implementação

Com um dos principais focos deste trabalho sendo usabilidade, o SPL-SZZ foi criado com o objetivo de ter fácil execução, uma boa performance, aliado à uma alta flexibilidade, para que fosse possível a alteração de seu código fonte de maneira simples, seja em seu algoritmo principal, seja na sua saída e entrada de dados.

3.1 Estrutura de módulos

Para que pudesse ser um código modular, com alta flexibilidade, o SPL-SZZ utilizou o recurso de *módulos* da linguagem Rust. Assim, cada módulo recebeu uma responsabilidade única e crucial para o funcionamento do algoritmo. O diagrama da Figura 3.1 ilustra a divisão de pastas e arquivos do projeto.

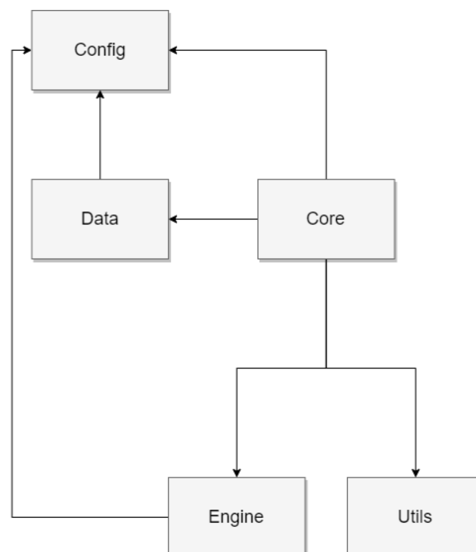

Figura 3.1: Diagrama de módulos

config: Responsável por ler todas as configurações presentes em um arquivo de configurações. Este arquivo contêm o endereço base da nuvem em que o código está disponível e deve ser clonado, o local onde os projetos clonados serão armazenados e, por fim, o caminho do diretório que será usado para adicionar arquivos temporários durante a procura pela revisão que introduziu o *bug*.

core: Este módulo contém toda a parte central do algoritmo. Aqui foi implementado o grafo de anotação responsável pela busca de revisões. É também responsável pelo início da execução do programa.

data: Trata-se do módulo responsável por toda a entrada e saída de dados do aplicativo. Aqui, foram implementadas as funções responsáveis por ler o arquivo que contém os revisões analisadas, ler o arquivo de configuração para o modulo *config* e criar o arquivo com o resultado da execução. Este módulo está pronto para que as entradas e saídas de dados possam ser adaptadas de forma simples e, dessa maneira, possam se adequar a qualquer estrutura que se faça necessária.

engine: É o responsável por extrair os dados necessários da ferramenta de versionamento de código, tendo sido o GIT, o engine utilizado neste trabalho. Este módulo fornece para o *core* as informações necessárias para que o grafo de anotação seja formado através de comandos executados via linha de comando, no diretório do projeto analisado.

utils: Contêm funções para edição de texto e alguns arquivos auxiliares. É utilizado pelo módulo *core*.

3.2 Estrutura de entrada e saída

```
[
 {
    "id": 85,
    "repo_name": "computationalcore/smartcoins-wallet",
    "fix_commit_hash": "038034fcb6daee5355c58bde8b870b5b87c28b3d",
    "bug_commit_hash": [
      "65efaa932ee6d0bcda6f97bf03616f1b7024aeaa"
    ],
    "best_scenario_issue_date": "2017-01-04T18:15:34",
```

```
10      "language": [
11        "java"
12      ]
13    },
14  ]
```

Código 3.1: Arquivo "bugfix_commits.json"

A entrada de dados, ilustrada em 3.1, corresponde a um arquivo com formato *.json* que tem como objetivo armazenar todas as informações sobre os projetos analisados. A estrutura de dados foi adaptada para que tivesse o mesmo formato do arquivo de *benchmark* da linguagem Java descrito por Rosa *et al.* [17] para os experimentos conduzidos neste trabalho.

```
1  [
2      {
3        "id":85,
4        "repo_name":"computationalcore/smartcoins-wallet",
5        "fix_commit_hash":"038034fcb6daee5355c58bde8b870b5b87c28b3d",
6        "bug_commit_hash":"65efaa932ee6d0bcda6f97bf03616f1b7024aeaa",
7        "founded_bug_commit_hash":"65efaa932ee6d0bcda6f97bf03616f1b7024
            aeaa",
8        "best_scenario_issue_date":"2017-01-04T18:15:34",
9        "language":[
10          "java"
11       ],
12       "correct":true
13     }
14   ]
```

Código 3.2: Arquivo "result.json"

A saída de dados, ilustrada em 3.2, corresponde a estrutura de entrada com acréscimo de dois novos valores: *"founded_bug_commit_hash"* e *"correct"* que correspondem ao *commit*/revisão indicado pelo algoritmo como o responsável pela introdução do *bug* e um valor booleano, sendo *true* (verdadeiro), se o resultado estiver correto, e *false* (falso), se estiver errado.

3.3 Grafo de anotação

O grafo de anotação, figura 3.2, é utilizado para mapear as mudanças no código ao longo do tempo de desenvolvimento do software. O grafo de anotação é um grafo multipartido, em que cada parte corresponde a uma versão de um arquivo modificado. Em cada parte/versão/revisão, cada linha de código alterada é representada por um único nó, as arestas entre os nós indicam que uma linha se origina de outra, seja por modificação, adição ou exclusão de outra linha. As linhas em negrito representam as linhas modificadas na revisão.

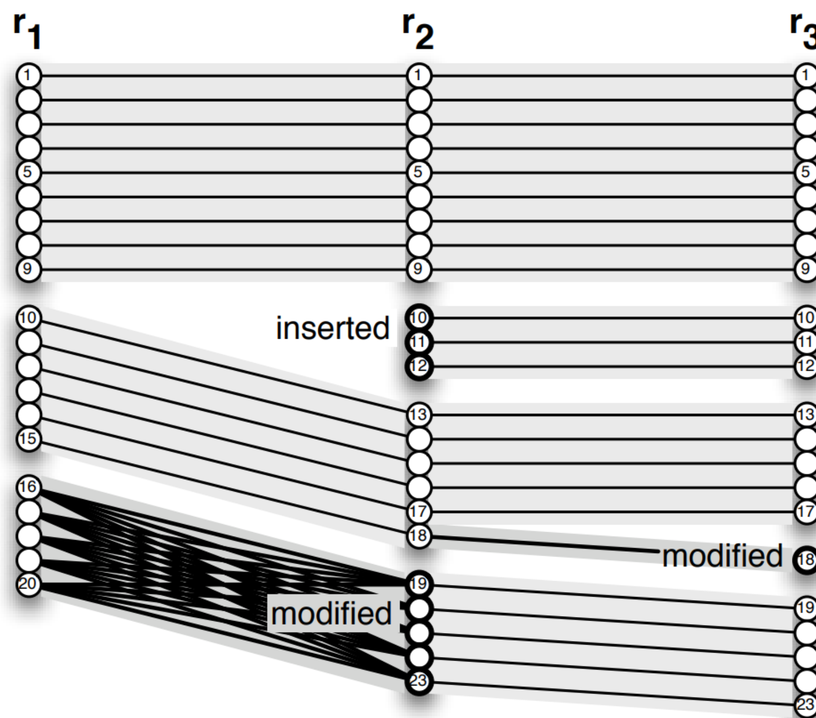

Figura 3.2: Grafo de Anotações - Fonte: *Zimmermann et al.* [2]

Na figura 3.3, temos a evolução das linhas ao longo de três mudanças consecutivas: Ck, Ck+1 e Ck+2. Percebe-se que de Ck a Ck+1 uma linha de código é excluída e uma linha de código é adicionada. Caso várias linhas sejam modificadas, o sistema de controle de versionamento identifica o caso como excluído e adicionado, e o grafo de anotação conectará as linhas adicionadas de Ck+1 à cada linha excluída de Ck através de arestas. Depois de rastrear todo o histórico do código, é executada uma busca em profundidade para encontrar as revisões introdutoras de *bugs*.

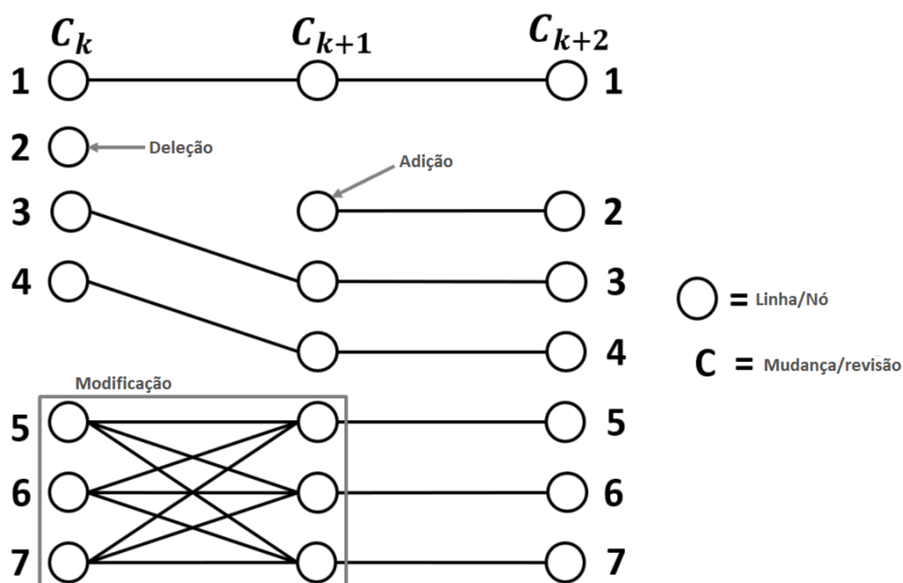

Figura 3.3: Grafo de Anotações - Fonte: Fan *et al.* [3]

3.3.1 Customização do Grafo de Anotações

Um dos problemas do grafo de anotação são as mudanças que modificam grande parte do código, pois essas mudanças resultam em um grande número de arestas. A fim de solucionar esse problema, foi proposto por *Zimmermann et al.* [2] que todas as grandes mudanças devem ser tradas não como alteração ou modificação, mas sim como um combinado de adição e exclusão de linhas. Assim, quando houverem grandes modificações, não serão criadas quaisquer arestas no grafo.

Foi proposta uma heurística que visa a auxiliar na identificação de grandes modificações no código. Considerando L e R como o tamanho da esquerda e da direita da região de um *Hunk* e FL e FR como o tamanho dos arquivos de cada revisão, e utilizando dessa heurística, um pedaço de código vai ser considerado uma grande modificação quando satisfazer as seguintes condições:

- O tamanho da região excede o limite:
  $L > máximo(\alpha * FL, \beta)$ ou $R > máximo(\alpha * FR, \beta)$
- O raio do tamanho da região excede o limite:
  $\frac{L}{R} < \frac{1}{\gamma}$ ou $\gamma < \frac{L}{R}$

A primeira condição reconhece alterações que afetam grandes partes de um arquivo, e a segunda reconhece alterações que inserem ou excluem grandes partes de um arquivo ou região. Nos experimentos do estudo de *Zimmermann et al.* [2] foram usados os seguintes valores: $\alpha = 0,10$ e $\beta = \gamma = 4$.

3.4 Algoritmo Implementado

Como mencionado na seção 2.2.6, o algoritmo implementado pelo presente estudo foi baseado na versão R-SZZ. Com o propósito de tornar o SPL-SZZ uma versão base, foram feitas algumas pequenas mudanças, sendo elas:

- Foi removida a verificação de comentários.
- Não verifica qual o formato do arquivo.
- Adaptação no grafo de anotações (ver a seção anterior) proposta por *Zimmermann et al.* [2].

Com o objetivo de focar em flexibilidade, definiu-se que os filtros deveriam ser executados externamente ou adicionados ao algoritmo em versões alternativas, que pudessem compor, dessa forma, uma linha de produtos com base no SPL-SZZ. Com uma arquitetura extremamente extensiva, adicionar filtros e alterar entradas mostraram ser alterações triviais e de fácil implementação.

Assim como no R-SZZ, no SPL-SZZ utilizou-se um grafo de anotação, descrito em *Zimmermann et al.* [2], como o *core* da aplicação. Este modulo *core* é responsável por fazer a busca na árvore de revisões do *GIT* com o objetivo de encontrar possíveis introdutores de *bug*.

Após ler o arquivo de entrada, o algoritmo inicia uma estrutura de repetição que segue os seguintes passos:

1. Clona o projeto do repositório remoto.
2. Busca todos os arquivos alterados na revisão que corrigiu o *bug*.
3. Constrói um grafo de anotação para cada arquivo modificado.
4. Identifica todas as linhas modificadas.
5. Busca a última revisão em que a linha foi alterada.
6. Insere a revisão mais recente na lista de revisões introdutoras de *bugs* que ao final será salva em um arquivo.

A Figura 3.4 ilustra o fluxo implementado:

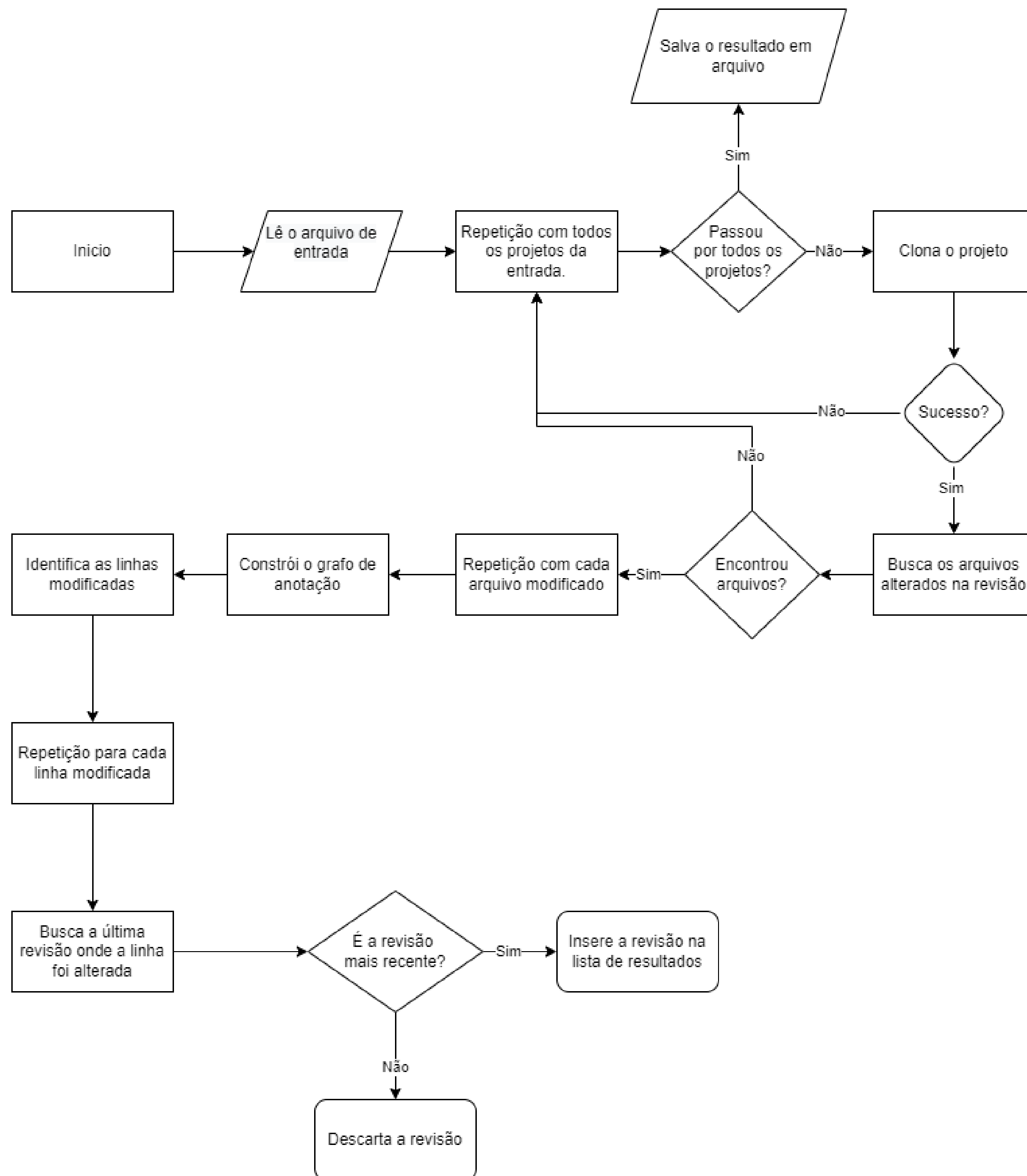

Figura 3.4: Diagrama de execução

3.5 Execução

A execução do programa é feita de forma extremamente simples. Como mencionado na seção 2.4, Rust é uma linguagem compilada, sendo assim ela gera arquivos binários que podem ser executados de forma muito simplificada pelo terminal. É necessário que o programa GIT esteja instalado na máquina que irá executar o algoritmo para que assim seja possível fazer buscas por modificações nos códigos analisados.

Com o programa compilado, o qual pode ser facilmente baixado do repositório [19], basta abrir um *Prompt de comando* e executá-lo através da linha de comando. Ele irá ler o arquivo de entrada e, após o processamento do algoritmo, irá salvar os resultados em outro arquivo chamado *"result.json"*.

Capítulo 4

Resultados

Após a conclusão da primeira versão, foi possível testar a implementação desenvolvida neste trabalho. Para estes testes, utilizando um benchmark disponível no estudo *Evaluating SZZ Implementations Through a Developer-informed Oracle* [17], que contempla uma lista de revisões recuperadas tanto de forma manual quanto automática e organizadas em arquivos através de repositórios públicos na plataforma *Github*. Deste benchmark, foi extraído um conjunto de dados que contêm informações sobre as correções de setenta e nove programas e as respectivas revisões que introduziram o *bug* no código. Com estes dados, foi possível executar o algoritmo implementado neste trabalho e imediatamente obter resultados de forma confiável acerca de sua precisão na busca por revisões introdutoras de *bugs*

Os dados utilizados foram os arquivos de entradas com as informações como o nome do projeto, data, revisões que corrigiram e as responsáveis por introduzir o *bug*, e ao final da execução foram comparadas com os resultados do software desenvolvido. A comparação e avaliação entre os dados obtidos do *benchmark* e os resultados do software implementado neste trabalho foram feitas de forma automatizada utilizando *scripts*.

Para comparação, utilizou-se outra implementação do SZZ, o RA-SZZ descrito por Costa *et al.* [11]. O RA-SZZ e o SPL-SZZ foram executados no mesmo ambiente, em que cada um foi responsável por clonar os projetos do repositório remoto. A execução do SPL-SZZ foi extremamente simples. Enquanto o RA-SZZ necessitou de configurações complexas, tais como: i) instalação de banco de dados, ii) geração de códigos para que os dados pudessem ser extraídos do banco de dados e iii) instalação da máquina Java e o GIT; o SPL-SZZ necessitou somente da instalação do programa GIT e da geração do arquivo *.json*, o qual funciona como entrada de dados.

Após a execução de ambos, criaram-se *scripts* (pequenos códigos executáveis)

com o objetivo de extrair os resultados do banco de dados utilizado pelo RA-SZZ. O SPL-SZZ salvou seus resultados em formato propício para que esses testes fossem analisados. Para que os dados pudessem ser comparados de forma automática, foi implementado um pequeno código, utilizando a linguagem *JavaScript*, o qual comparou os dados de ambas as implementações. As métricas utilizadas foram a precisão, *recall* e *F1-score*, calculadas de acordo com as seguintes fórmulas:

$$precisão = \frac{acertos}{acertos + erros}$$

$$\textit{recall} = \frac{acertos}{acertos + não\ encontrados}$$

$$\textit{F1-score} = 2 * \frac{precisão * recall}{precisão + recall}$$

Os resultados podem ser observados nas tabelas abaixo:

Algoritmo	Precisão
RA-SZZ	0.3596
SPL-SZZ	0.7142

Tabela 4.1: Precisão

A Tabela 4.1 apresenta uma taxa de acertos consideravelmente alta para o SPL-SZZ e baixa para o RA-SZZ. Embora tenha demonstrado uma alta taxa de precisão, a disparidade de resultados corretos é considerável. Em razão disso, foram levados em consideração, para essa análise, outros parâmetros. A Tabela 4.2 tem como objetivo demonstrar o motivo dessa discrepância em relação à precisão de ambas as implementações.

Algoritmo	Obteve resultado	Corretos	Errados	Recall	F1-Score
RA-SZZ	57	41	73	0.6508	0.4632
SPL-SZZ	14	10	4	0.1333	0.2246

Tabela 4.2: Resultados

Embora tenha obtido uma ótima precisão, o SPL-SZZ conseguiu encontrar menos revisões introdutoras de *bugs* do que a outra implementação. Essa diferença acontece pois, enquanto o SPL-SZZ busca apenas uma, o RA-SZZ procura em várias possíveis revisões introdutoras de *bugs* para uma única revisão de correção. De um total de quatorze resultados obtidos, o SPL-SZZ errou apenas em quatro,

porém, deixou as demais sessenta e cinco revisões sem um resultado. Desta forma, ele obteve uma boa precisão em detrimento de ter um menor número de acertos. Pode-se perceber esta diferença nas métricas *Recall* e *F1-Score*.

Capítulo 5

Conclusão

O projeto desenvolvido ao longo deste trabalho de graduação resultou na implementação de um software correspondente do algoritmo SZZ denominado SPL-SZZ. Mostrou-se necessário um grande estudo e discussão acerca de implementações já desenvolvidas do algoritmo SZZ, visto que cada uma destas tinha suas particularidades que as diferenciava das demais. Mostrou-se necessário também um grande estudo acerca de qual linguagem poderia vir a ser a ideal para este trabalho, onde no fim, após analisadas diversas opções, optou-se pela linguagem Rust.

A partir dos resultados observados na busca de revisões introdutoras de *bugs*, pode-se concluir que o SPL-SZZ foi uma implementação do SZZ com uma boa precisão, mas que não conseguiu encontrar um grande número de revisões introdutoras de *bugs*. Houve uma evolução considerável em usabilidade, e o SPL-SZZ resultou em um projeto extremamente simples de ser executado, não demandando muitos passos para que ele conseguisse executar a análise e salvar os resultados.

Ao comparar os dados obtidos pelo SPL-SZZ com os resultados da avaliação feita no estudo *Evaluating SZZ Implementations Through a Developer-informed Oracle* [17] observou-se que os resultados obtidos pelo projeto desenvolvido ao longo deste trabalho se equiparam em taxas de precisão com o algoritmo R-SZZ, escolhido como modelo de implementação a ser seguido. O algoritmo desenvolvido alcançou a taxa de aproximadamente 71% de precisão ao executar o *benchmark*, enquanto o modelo descrito em *Evaluating SZZ Implementations Through a Developer-informed Oracle* [17] alcançou 73% de precisão.

Dessa forma, observou-se que o objetivo principal deste trabalho, que consistiu em desenvolver uma implementação do algoritmo SZZ totalmente focada em usabilidade e extensibilidade, foi alcançado, tendo em vista que a taxa de precisão obtida foi satisfatória e que o software desenvolvido facilita a execução do algoritmo e o armazenamento dos resultados obtidos pelo mesmo, o que o torna atrativo para

cenários reais de uso e não só acadêmicos.

A avaliação dos resultados obtidos, nos permite concluir que um caminho para melhorias futuras deste trabalho estaria relacionado com o aperfeiçoamento do algoritmo SZZ presente no modulo *core* do código aqui implementado. Isso porque o baixo número de revisões introdutoras de *bugs* encontrados pelo SPL-SZZ em relação a outra implementação aqui apresentada (RA-SZZ), demonstra que embora o SPL-SZZ esteja no caminho correto, ele deve ser melhorado para que consiga manter sua taxa de assertividade e, ainda assim, obter resultados para um maior número de revisões. Também tem-se como possível trabalho futuro a adição de filtros ao SPL-SZZ, para que seja possível entender qual impacto direto eles teriam nos resultados obtidos, levando em conta métricas como o tempo de execução e precisão.

Referências

[1] Borg, Markus, Oscar Svensson, Kristian Berg e Daniel Hansson: *Szz unleashed: an open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project*. MaLTeSQuE 2019: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, páginas 7–12, 2019. viii, 10

[2] Zimmermann, Thomas, Sunghun Kim, Andreas Zeller e E. James Whitehead Jr.: *Mining version archives for co-changed lines*. MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, páginas 72–76, 2006. viii, 1, 15, 16, 17

[3] Fan, Yuanrui, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan e Shanping Li: *The impact of mislabeled changes by szz on just-in-time defect prediction*. IEEE Transactions on Software Engineering, 47(8), 2019. viii, 1, 7, 16

[4] Pan, Kai, Sunghun Kim e Emmet James Whitehead: *Toward an understanding of bug fix patterns*. Empirical Software Engineering, 14(3):286–315, 2009. 1

[5] Murphy-Hill, Emerson, Thomas Zimmermann, Christian Bird e Nachiappan Nagappan: *The design space of bug fixes and how developers navigate it*. IEEE Transactions on Software Engineering, 41(1):65–81, 2014. 1

[6] LaToza, Thomas D., Gina Venolia e Robert DeLine: *Maintaining mental models: a study of developer work habits*. Proceedings of the 28th international conference on Software engineering, páginas 492–501, 2006. 1

[7] Śliwerski, Jacek, Thomas Zimmermann e Andreas Zeller: *When do changes induce fixes?* Software Engineering Notes, 30(4):1–5, 2005. 1, 5, 6, 7

[8] Neto, Edmilson Campos, Daniel Alencar da Costa e Uirá Kulesza: *Revisiting and improving szz implementations*. IEEE Transactions on Software Engineering, 2019. 1, 7

[9] Costa, Daniel Alencar da, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho e Ahmed E. Hassan: *A framework for evaluating the results of the szz approach for identifying bug-introducing changes*. IEEE Transactions on Software Engineering, 43(7), 2016. 1, 6, 7, 8

[10] Petrulio, Fernando, David Ackermann, Enrico Fregnan, Gül Calikli, Marco Castelluccio, Sylvestre Ledru, Calixte Denizet, Emma Humphries e Alberto Bacchelli: *Szz in the time of pull requests*. IEEE Transactions on Software Engineering, 2022. 1, 7

[11] Costa; Uirá Kulesza, Edmilson Campos Neto; Daniel Alencar da: *The impact of refactoring changes on the szz algorithm: An empirical study*. IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018. 1, 8, 20

[12] Kim, Sunghun, Thomas Zimmermann, Kai Pan e E. James Jr. Whitehead: *Automatic identification of bug-introducing changes*. 21st IEEE/ACM International Conference on Automated Software Engineering, 2006. 1, 5, 6, 8

[13] Williams, Chadd e Jaime Spacco: *Szz revisited: verifying when changes induce fixes*. DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems, páginas 32–36, 2008. 1, 6

[14] Davies, Steven, Marc Roper e Murray Wood: *Comparing text-based and dependence-based approaches for determining the origins of bugs*. JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS, páginas 01–37, 2014. 1, 6, 9

[15] Rosa, Giovanni, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza e Rocco Oliveto: *Replication package*. https://github.com/SZZ-Research/icse2021-szz-replication-package. 2, 3

[16] Correia, João Lucas Marques: *Reproduzindo algoritmos para detecção de pontos de inserção de bugs*. páginas 30–32, 2019. 3

[17] Rosa, Giovanni, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza e Rocco Oliveto: *Evaluating szz implementations through a developer-informed oracle*. International Conference on Software Engineering, 2021. 3, 7, 10, 14, 20, 23

[18] Lin, Yi, Stephen M. Blackburn, Antony L. Hosking e Michael Norrish: *Rust as a language for high performance gc implementation*. ACM SIGPLAN Notices, 51(11):89–98, 2016. 3, 11

[19] Fortes, José e Adelson Jhonata: *Spl-szz*. https://github.com/zfortes/spl-szz. 3, 19

[20] Canfora, Gerardo, Luigi Cerulo e Massimiliano Di Penta: *Identifying changed source code lines from version repositories*. Fourth International Workshop on Mining Software Repositories, 2007. 6