Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Ciência da Computação

Análise de extensão do CSDiff para uso em linguagens com poucos separadores sintáticos

José Gabriel Silva Pereira

Trabalho de Graduação

Recife
27 de Abril de 2023

Universidade Federal de Pernambuco
Centro de Informática

José Gabriel Silva Pereira

Análise de extensão do CSDiff para uso em linguagens com poucos separadores sintáticos

Trabalho apresentado ao Programa de Bacharelado em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Paulo Henrique Monteiro Borba*
Co-orientadora: *Paola Rodrigues de Godoy Accioly*

Recife
27 de Abril de 2023

Resumo

A prática de desenvolvimento de software, há muito tempo deixou de ser uma tarefa que era realizada por somente uma pessoa, pois, com o avanço da tecnologia, sistemas cada vez mais complexos foram sendo criados fazendo com que muitas pessoas trabalhassem no mesmo projeto. Por conta disso, ferramentas de controle de versionamento de código foram criadas, permitindo que múltiplos desenvolvedores trabalhassem modificando o mesmo trecho de código simultaneamente. Porém, essas modificações simultâneas podem gerar conflitos quando feitas em um mesmo pedaço de código, o que impacta negativamente na produtividade de um time. Ao decorrer do tempo, diversas formas de como detectar conflitos na junção de versão de códigos foram criadas, dentre elas: linha a linha, estruturada e semiestruturada. Neste trabalho, é proposta uma extensão para uma ferramenta semiestruturada de detecção de conflitos ja existente, o *CSDiff* [1], de forma que ela utilize indentação como separador da linguagem, permitindo assim que, durante a detecção de conflitos em linguagens com poucos separadores sintáticos, ainda seja possivel permitir uma redução de falsos conflitos, consequentemente melhorando a produtividade de um time.

Palavras-chave: Processo de Merge, Desenvolvimento Colaborativo, Merge Textual, Merge Estruturado, Separadores sintáticos

Abstract

The practice of software development has long ceased to be a task performed by only one person, as with the advancement of technology, increasingly complex systems have been created, causing many people to work on the same project. Therefore, code version control tools were created, allowing multiple developers to work on modifying the same piece of code simultaneously. However, these simultaneous modifications can generate conflicts when made on the same piece of code, negatively impacting a team's productivity. Over time, several ways of detecting conflicts in the merging of code versions have been created, including line-by-line, structured, and semi-structured. In this work, an extension is proposed for an existing semi-structured conflict detection tool, the *CSDiff* [1], so that it uses indentation as a language separator, thus allowing a reduction in false conflicts during conflict detection in languages with few syntactic separators, consequently improving a team's productivity.

Keywords: Merge Process, Collaborative Development, Textual Merge, Structured Merge, Syntactic Separators.

CAPÍTULO 1

Introdução

Com o crescimento da complexidade dos sistemas de software, surge a necessidade de que múltiplas pessoas trabalhem num mesmo projeto. Essas modificações, com o objetivo de trazer mais produtividade, costumam ser executadas de forma paralela e podem acontecer em trechos de código em comum. Tendo como objetivo auxiliar os desenvolvedores a controlar e versionar suas modificações no código, ferramentas de controle de versão de código foram criadas. Essas ferramentas auxiliam a reduzir o trabalho extra quando se trata de modificações paralelas que precisam ser unidas. O processo de unir duas modificações paralelas de código é chamado de *merge* [2].

No processo de *merge*, quando dois desenvolvedores modificam o mesmo trecho de código e essas mudanças interferem uma na outra, é gerado um conflito. Esses conflitos, quando detectados, algumas vezes precisam ser resolvidos por um ou ambos os desenvolvedores, o que acaba impactando na produtividade, dado que resolvê-los geralmente é uma tarefa que geralmente demanda tempo [3]. Além do impacto na produtividade do time, quando esses conflitos não são detectados pela ferramenta de *merge*, ou quando são detectados e mal resolvidos, eles podem levar à introdução de bugs dentro do código, o que influencia na qualidade do produto final [4].

A abordagem de *merge* mais utilizada na indústria atualmente é o *merge* não estruturado [5], que se utiliza de uma análise puramente textual, equiparando linha a linha trechos do código para detectar e resolver conflitos. Porém, por não utilizar a estrutura do código que está sendo integrado, muitas vezes essa abordagem gera falsos conflitos. Ao observar isso, pesquisadores propuseram ferramentas que se baseiam na estrutura dos arquivos que estão sendo integrados, criando uma árvore sintática a partir do texto dos arquivos e de sua linguagem de programação [6]. Essas abordagens são chamadas estruturadas e semiestruturadas.

Em estudos anteriores [6]–[8], as duas abordagens (estruturada e semiestruturada) foram comparadas em relação à não estruturada e mostrou-se que, para a maioria das situações de *merge* dos projetos, houve uma redução de conflitos em favor da semi ou da estruturada. Essa redução se dá por conta de falsos conflitos que possuem resolução óbvia, como por exemplo, quando os desenvolvedores adicionam dois métodos diferentes e independentes numa mesma região do código [9].

Esse benefício advém da exploração da estrutura gerada pela análise sintática, também chamada de análise gramatical. Ela envolve o agrupamento dos tokens (palavras) do programa fonte em frases. Cada linguagem possui conjuntos de tokens, onde alguns servem como divisores de elementos sintáticos e escopo semântico, como por exemplo as chaves (’{’, ’}’) numa linguagem como Java. Estes tokens são definidos aqui como **separadores** sintáticos.

A solução não estruturada mais utilizada, o *Diff3*, se baseia somente na quebra de linha

como o divisor de contexto para detecção de conflitos. Assim, o algoritmo de *merge* compara as linhas mantidas, adicionadas, e removidas por cada desenvolvedor e, com base nisso, reporta conflito quando as mudanças ocorrem em uma mesma área do texto, isto é, uma área onde não há uma mesma linha (ou linhas consecutivas) mantida que separa as mudanças feitas pelos dois desenvolvedores.

Como forma de melhorar os resultados do *Diff3*, o *CSDiff*, proposto em trabalhos anteriores [1], utiliza-se dos separadores mencionados acima para dividir o contexto de cada linha de código. Assim, o algoritmo de *merge* consegue, por exemplo, resolver conflitos em uma mesma linha, contanto que esses conflitos estejam separados por pelo menos um dos separadores definidos.

Apesar de funcionar bem para linguagens como Java, o *CSDiff* possui limitações, pois linguagens como Python, possuem poucos separadores (seu principal separador é a própria indentação do código, que não é considerado pelo *CSDiff* atual). Este trabalho propõe uma modificação para o *CSDiff*, que utiliza a indentação como um separador sintático, de forma a tentar resolver esse problema, e analisa os resultados em comparação ao *merge* não estruturado puramente textual. Em particular, investiga-se as seguintes perguntas de pesquisa:

1) PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?
2) PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?
3) PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?
4) PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?
5) PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

Os resultados obtidos mostram que além de aumentar a quantidade de conflitos reportados (como esperado considerando os resultados dos trabalhos anteriores), o *merge* não estruturado utilzando separadores e indentação, demonstra, para a amostra utilizada, um aumento de cenários *aFN* (Falsos Negativos Adicionados, ou seja, cenários onde a ferramenta resolveu conflitos de forma errada, enquanto a ferramenta *Diff3* resolveu corretamente) proporcional a quantidade de cenários *aFP* (Falsos Positivos Adicionados, ou seja, cenários onde a ferramenta reportou conflitos enquanto a ferramenta Diff3 não reportou nenhum e seu resultado era igual ao do merge do repositório) reduzido quando comparado ao *Diff3*.

Por outro lado, ao se fazer a análise de aumento de produtividade considerando o ato de resolver e reduzir conflitos, além de considerar geração de falsos conflitos reais e resolução errada de conflitos, notou-se um bom aumento de produtividade com a utilização do *CSDiff* original, enquanto que o *CSDiff* modificado demonstrou um resultado levemente pior. Os conceitos utilizados para esta análise são explicados na seção 4.2.5.

CAPÍTULO 2

Motivação

2.1 Merge Não Estruturado

Apesar da evolução dos sistemas de controle de versão, as ferramentas utilizadas para realizar *merge* não evoluiram tanto. A ferramenta mais utilizada atualmente é o *Diff3* [2], que consiste em uma abordagem puramente textual baseada em linhas. Essa ferramenta, ao receber os três arquivos que configuram o cenário de merge (chamemos de *base*, *left* e *right*), compara, linha a linha, as duas versões modificadas (*left* e *right*) com seu ancestral comum (*base*), que foi o arquivo que deu origem às modificações que vão ser integradas. Após isso, a ferramenta agrupa as maiores áreas em comum e checa se existem interseções entre as áreas modificadas por *left* e *right*. Essas interseções são definidas como conflitos de merge [5], elas serão sinalizadas pelo *Diff3* através de marcadores (``>>>>>>>``, ``=======`` e ``<<<<<<<``), como demonstrado na Figura 2.4.

Por considerar as mudanças linha a linha para detectar os conflitos, muitas vezes essa ferramenta relata falsos conflitos de modificações que alteram a mesma linha ou linhas consecutivas, mas que não interferem entre si semanticamente. Esse problema poderia ser resolvido utilizando ferramentas que explorem a estrutura sintática do código em questão [8]

Para ilustrar esse problema causado pelo *merge* não estruturado, utilizamos a implementação de um método to_string que recebe uma lista de strings e retorna uma string juntando todos os elementos da lista separados por dois *underlines* (“__“). Observe que as mudanças feitas pelo *left* foram a adição de uma nova condição no *if*, e a mudança da string separadora do *return* final (modificado de “..“ para “__“). Por outro lado, as mudanças feitas pelo *right* foram o valor padrão de retorno de uma string vazia para uma constante (por simplicidade consideramos que a constante foi definida na classe onde o método está sendo implementado), e a mesma modificação que o *left* fez no último return do método. Perceba que todas essas mudanças são independentes entre si, mas por elas acontecerem em linhas consecutivas, o *Diff3* relata todas elas em um mesmo conflito (Figura 2.4).

```
1 def to_string(l: List[str]) -> str:
2     if len(l) == 0:
3         return ""
4     return "..".join(l)
```

Figura 2.1 Arquivo *base* que contém o método to_string

```
def to_string(l: List[str]) -> str:
    if l is null or len(l) == 0:
        return ""
    return "__".join(l)
```

Figura 2.2 Arquivo *left* que contém o método to_string

```
def to_string(l: List[str]) -> str:
    if len(l) == 0:
        return self.D
    return "__".join(l)
```

Figura 2.3 Arquivo *right* que contém o método to_string

2.2 Merge Semiestruturado e Estruturado

Como alternativa ao uso de merge não estruturado, existem as abordagens semiestruturadas ou completamente estruturadas [8]. Ao contrário da abordagem não estruturada, essas abordagens levam em consideração a estrutura sintática da linguagem de programação para identificar conflitos com maior precisão e resolvê-los de forma mais correta. Essas abordagens criam árvores sintáticas para cada versão dos arquivos a serem integrados (*base*, *left* e *right*) e comparam essas árvores para identificar nós comuns e adições ou remoções em cada árvore. Dessa forma, cada elemento sintático é representado em nós distintos, e conflitos são sinalizados quando as mudanças a serem integradas estão relacionadas ao mesmo nó da árvore. Isso significa que, em vez de usar linhas como a unidade básica para comparação, essas ferramentas usam nós sintáticos como unidade.

Essas ferramentas estruturadas e semiestruturadas conseguem evitar falsos conflitos encontrados na abordagem não estruturada. Por exemplo, duas situações em que dois desenvolvedores adicionam separadamente dois novos métodos com diferentes assinaturas em uma mesma área do texto podem ser conciliadas com sucesso. As mudanças ocorrem na mesma linha, mas cada declaração é representada por um nó diferente, pois o identificador do método é parte do nó, e os dois nós são mantidos na árvore resultante da integração.

Apesar da melhora em relação ao relato de falsos positivos, Cavalcanti [9] argumenta que ferramentas semiestruturadas tendem a gerar falsos negativos que são mais difíceis de detectar e resolver, além de não encontrar evidências de que essas ferramentas reduzem a quantidade de falsos negativos relatados em comparação a ferramentas não estruturadas.

Dessa forma, é fácil observar que uma ferramenta estruturada para Python evitaria o conflito apresentado na Figura 2.4. A ferramenta, utilizando a estrutura da linguagem, identificaria que, apesar das mudanças representadas na Figura 2.2 e na Figura 2.3 ocorrerem em linhas consecutivas (o que faz com que o *Diff3* agrupe as mudanças em um único bloco de conflito), elas estariam associadas a nós diferentes na árvore sintática. A ferramenta então juntaria as mudanças em uma versão resultante que contém a nova condição proposta por *left* e o novo

```
def to_string(l: List[str]) -> str:
<<<<<<< ./left.py
    if l is null or len(l) == 0:
        return ""
    return "__".join(l)
=======
    if len(l) == 0:
        return self.D
    return "__".join(l)
>>>>>>> ./right.py
```

Figura 2.4 Resultado de executar o *Diff3*

valor de retorno proposto por *right*, evitando o falso conflito.

2.3 Merge não Estruturado com separadores

As abordagens estruturadas e semiestruturadas discutidas na seção anterior, apesar benéficas em algumas situações, possuem um custo adicional, dado que elas se baseiam em manipulação de árvores sintáticas, que por sua vez são dependentes da linguagem em questão e demandam um esforço significativo de implementação por linguagem, além de que a manipulação das árvores sintáticas possui um custo computacional maior em relação a abordagens puramente textuais.

Para reduzir essas desvantagens, Clementino [1] propõe uma nova ferramenta chamada **Custom Separators Diff**[1] (*CSDiff*), cujo funcionamento baseia-se na abordagem puramente textual, mas que também considera a estrutura sintática do programa por meio de um conjunto de separadores da linguagem, escolhidos pelo usuário e passados como parâmetro.

(1) Transforma os arquivos *base*, *left* e *right*, fazendo com que os separadores sintáticos dados como entrada fiquem em linhas separadas, adicionando uma linha antes e uma depois de cada separador; essas novas linhas são marcadas com a sequência de caracteres: '$$$$$$$$';
(2) Chama o *merge* textual do *Diff3*, passando como entrada os arquivos gerados por 1;
(3) No arquivo resultante do Passo 2, remove as linhas extras e marcadores adicionados no Passo 1.

Listagem 2.1: Processo do *CSDiff*

Para ilustrar o funcionamento do *CSDiff* e seu potencial para resolver falsos conflitos, observamos as Figuras 2.5, 2.6 e 2.7 onde executamos o processo descrito na Listagem 2.1 utilizando os separadores sintáticos de Python "(", ")" e ",".

Nota-se na Figura 2.8 que ao executar o *CSDiff* nos arquivos *base*, *left* e *right*, as linhas conflitantes ficam separadas pelos marcadores (as sequencias "$$$$$$$$"), o que faz com que o

[1] https://github.com/JonatasDeOliveira/custom-separators-merge-tool/blob/main/csdiff.sh

conflito relatado pelo *diff3* seja reduzido (indicando que parte dele foi parcialmente resolvido automaticamente).

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return ""
    return "..".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.5 Arquivo *base* após a o Passo 1 da Listagem 2.1

Entretanto, nota-se uma limitação para este caso na linguagem Python, visto que as duas expressões de *return* acabam não se separando simplesmente pelo fato de que não há nenhum separador sintático entre eles. Essa limitação não ocorre em linguagens como Java, por exemplo, pois geralmente essas duas expressões *return* seriam separadas por um "}" ou "**;**" (que em Java delimita final de escopo). Isso acontece para a linguagem Python pelo fato de que a mesma não possui separador para delimitar escopo, pois utiliza da indentação para tal.

Dada esta limitação, propomos neste trabalho uma modificação no algoritmo do *CSDiff*, visando resolver esse problema de separação de escopo. Para fazer isso, modificamos o algoritmo para que também adicione marcadores ao encontrar mudanças na indentação do programa. Ilustraremos isso no próximo capítulo.

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if l is null or len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return ""
    return "__".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.6 Arquivo *left* após a o Passo 1 da Listagem 2.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
        return self.D
    return "__".join
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.7 Arquivo *right* após a o Passo 1 da Listagem 2.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
    if l is null or len
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
<<<<<<< left.py_temp.py
        return ""
    return "__".join
=======
        return self.D
    return "__".join
>>>>>>> right.py_temp.py
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 2.8 Arquivo resultante após a execução do Passo 2 da Listagem 2.1 nos arquivos *base*, *left* e *right*

```
def to_string(l: List[str]) -> str:
    if l is null or len(l) == 0:
<<<<<<< ./left.py
        return ""
    return "__".join
=======
        return self.D
    return "__".join
>>>>>>> ./right.py
(l)
```

Figura 2.9 Arquivo resultante após a execução do Passo 3 da Listagem 2.1 nos arquivos *base*, *left* e *right*

CAPÍTULO 3

Solução

3.1 Visão Geral da Implementação

3.1.1 Simplificação com uso de *awk*

Primeiramente, para que conseguíssemos fazer o *CSDiff* detectar mudanças de indentação, era necessário que o programa iterasse linha por linha, no Passo 1 da Figura 2.3. Então, ao pesquisar por ferramentas alternativas ao *sed* (que, ao contrário do *awk*, faz a substituição no arquivo inteiro de uma vez), encontramos e escolhemos utilizar a ferramenta *awk* [10], utilizada primeiramente para simplificar alguns dos scripts feitos com a ferramenta *sed* que foram utilizados em [1], [11]. Esta simplificação foi feita através do conceito chamado de *charclass*[1] que permite criar um comando de substituição mais simples do que o utilizado anteriormente, que consistia em uma sequencia de substituições *sed* individuais. Além disso, essa modificação foi o que permitiu que o programa iterasse linha por linha de uma maneira eficiente e programática.

Essa simplificação foi posteriormente testada pelo autor utilizando o *miningframework* (utilizado nessa pesquisa, bem como nas anteriores já mencionadas), para comprovar sua equivalência em relação ao script inicial (sequência de comandos *sed*).

3.1.2 Inserindo Marcadores ao Detectar Mudanças de Indentação

Após a simplificação acima, foi criada uma nova versão do *CSDiff* cujo processo é descrito na Listagem 3.1 e seu resultado ilustrada nas Figuras 3.1, 3.2, 3.3, enquanto que o resultado da execução do *diff3* nos arquivos modificados pode ser visto na figura 3.4

(1) Iterando linha a linha nos arquivos *base*, *left* e *right*:
   (a) Transforma os arquivos, fazendo com que os separadores sintáticos dados como entrada fiquem em linhas separadas, adicionando uma linha antes e uma depois de cada separador; essas novas linhas são marcadas com a sequência de caracteres: '$$$$$$$$', e adiciona uma linha de marcadores a mais acima da linha atual sempre que a linha anterior está em um nível de indentação diferente da linha atual.
(2) Chama o *merge* textual do *Diff3*, passando como entrada os arquivos gerados por 1;
(3) No arquivo resultante do Passo 2, remove as linhas extras e marcadores adicionados no Passo 1.

Listagem 3.1: Processo do *CSDiff* considerando mudanças de indentação

[1] https://www.regular-expressions.info/charclass.html

```
1  def to_string
2  $$$$$$$(
3  $$$$$$$l: List[str]
4  $$$$$$$)
5  $$$$$$$ -> str:
6  $$$$$$$
7      if len
8  $$$$$$$
9  $$$$$$$(
10 $$$$$$$l
11 $$$$$$$)
12 $$$$$$$ == 0:
13 $$$$$$$
14         return ""
15 $$$$$$$
16     return "..".join
17 $$$$$$$
18 $$$$$$$(
19 $$$$$$$l
20 $$$$$$$)
21 $$$$$$$
```

Figura 3.1 Arquivo *base* após o Passo 1 da Listagem 3.1

Como podemos ver nas Figuras 3.1, 3.2 e 3.3, foi adicionada uma nova linha de marcadores entre os dois *return*s, de forma que todas as linhas conflitantes deixaram de estar em posições consecutivas, pois a linha de marcadores (mais especificamente o da linha 15) é idêntica em todos os arquivos. Isso foi suficiente para que o Passo 2 da Listagem 3.1 resolvesse todos os conflitos de forma automática.

Por questão de simplicidade, utilizamos apenas dois separadores nos exemplos acima “(“ e “)“ (abre parênteses e fecha parênteses). Entretanto, Python também possui outros separadores possivelmente relevantes, como “,“ (vírgula) e “:“ (dois pontos), que utilizamos na avaliação desse trabalho. Apesar disso, vale resslatar que na linguagem, o separadoor “:“ normalmente vem precedido de um “)“ e/ou seguido de uma mudança de indentação. Isso faz com que o *CSDiff* crie linhas marcadoras redundantes, e como veremos mais a frente, tem influência direta na geração de falsos conflitos.

Com essas modificações, executamos o experimento descrito na seção seguinte.

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if l is null or len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return ""
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.2 Arquivo *left* após o Passo 1 da Listagem 3.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return self.D
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.3 Arquivo *right* após o Passo 1 da Listagem 3.1

```
def to_string
$$$$$$$(
$$$$$$$l: List[str]
$$$$$$$)
$$$$$$$ -> str:
$$$$$$$
    if l is null or len
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$ == 0:
$$$$$$$
        return self.D
$$$$$$$
    return "__".join
$$$$$$$
$$$$$$$(
$$$$$$$l
$$$$$$$)
$$$$$$$
```

Figura 3.4 Arquivo resultante após a execução do Passo 2 da Listagem 3.1 nos arquivos *base*, *left* e *right*. Note que agora o *Diff3* conseguiu resolver todos os conflitos de forma automática.

```
def to_string(l: List[str]) -> str:
    if l is null or len(l) == 0:
        return self.D
    return "__".join(l)
```

Figura 3.5 Arquivo resultante após o Passo 3 da Listagem 3.1

CAPÍTULO 4

Avaliação

Para avaliar esta nova versão do *CSDiff* (que aqui chamaremos de *CSDiffI*) e responder as perguntas de pesquisa mencionadas anteriormente, repetimos o experimento feito por Clementino [1], [11], que compara os resultados de utilizar *CSDiff* com o resultado de utilizar *Diff3*, analisando o potencial para resolução de conflitos sem gerar impacto negativo na corretude do processo de *merge*. Como meio de facilitar a execução desta análise, foi utilizado o *mining-framework*,[1] que automatiza o processo de coletar os cenários de *merge*, além de executar as ferramentas *CSDiffI* e *Diff3* em cada arquivo de cada cenário.

4.1 CONCEITOS

A seguir, alguns conceitos relevantes para a avaliação serão definidos.

4.1.1 Cenário de Merge

Em um sistema de controle de versão, um *commit* é uma versão que agrupa mudanças em determinados arquivos de um projeto [12]. Considerando essa definição, um Cenário de Merge é definido como uma quádrupla de *commits*, que chamaremos aqui de *base*, *left*, *right* e *merge*. O *base* representa o *commit* de onde as modificações *left* e *right* partiram, enquanto o *merge* representa a versão final onde a integração das mudanças foi feita no repositório.

4.1.2 Falso Positivo Adicionado

Seguindo as mesmas definições descritas em [11], um falso positivo para uma dada ferramenta em relação a outra ocorre quando a ferramenta de *merge* relata um conflito que na verdade não deveria ter ocorrido. Para comparar duas ferramentas de merge X e Y, usamos o conceito de Falso Positivo Adicionado *aFP*, que acontece quando a ferramenta X relata conflito em um determinado cenário de *merge*, enquanto a ferramenta Y não relata conflito no mesmo cenário. É importante usar o conceito de *aFP* porque o conjunto de falsos conflitos de uma ferramenta não é um subconjunto da outra.

[1] https://github.com/spgroup/miningframework

4.1.3 Falso Negativo Adicionado

Um falso negativo ocorre quando a ferramenta de *merge* não relata um conflito que deveria ter sido reportado. Ao comparar duas ferramentas de *merge* X e Y, usamos o conceito de Falso Negativo Adicionado (*aFN*), que acontece quando a ferramenta X não relata conflito em um determinado cenário de *merge*, enquanto a ferramenta Y relata conflito no mesmo cenário. É importante usar o conceito de aFN porque o conjunto de falsos negativos de uma ferramenta não é um subconjunto da outra.

4.2 PERGUNTAS DE PESQUISA

A avaliação desta pesquisa é baseada em responder as seguintes perguntas de pesquisa.

4.2.1 PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?

Para avaliar o número de conflitos gerados pelo *Diff3* e *CSDiff*, contamos o número de conflitos em cada ferramenta para cada cenário de merge. Para isso, executamos a ferramenta para os conjuntos de arquivos de *left*, *right* e *base* em cada cenário, resultando em um conjunto de arquivos combinados. Em seguida, contamos a ocorrência de marcadores de conflito para cada arquivo presente nesses conjuntos, que são sequências de caracteres apresentados no formato de conflito descrito na Figura 2.4.

4.2.2 PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?

Para avaliar o número de cenários de *merge* com conflitos gerados pelo *Diff3* e *CSDiff*, contamos o número de cenários em cada ferramenta em que houve conflito. Para isso, executamos a ferramenta para os conjuntos de arquivos de *left*, *right* e *base* em cada cenário de *merge*, resultando em um conjunto de arquivos combinados. Um cenário é considerado com conflito se pelo menos um arquivo no conjunto resultante do *merge* tiver um conflito.

4.2.3 PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?

Para responder a esta pergunta, foram contabilizados os casos em que uma ferramenta apresentou um falso positivo adicionado (*aFP*) em relação à outra. Para verificar se um cenário continha um *aFP* do *Diff3* em comparação com o *CSDiff*, foi verificado se o *Diff3* retornou pelo menos um conflito em pelo menos um dos arquivos integrados no cenário, enquanto o *CSDiff* não retornou nenhum conflito para todos os arquivos do cenário e obteve o resultado correto do merge. Neste caso, foi contabilizado que o *Diff3* tinha um cenário com *aFP* em relação ao *CSDiff*. O mesmo procedimento foi realizado para encontrar *aFPs* adicionados pelo

CSDiff em comparação com o *Diff3*.

4.2.4 PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?

Para verificar se um cenário possui um *aFN* para uma ferramenta em comparação com outra, o *merge* foi executado usando tanto o *Diff3* quanto o *CSDiff*. Se o *CSDiff* não retornasse nenhum conflito em nenhum arquivo resultante do *merge* no conjunto de arquivos do cenário, enquanto o *Diff3* retornasse conflito e o *CSDiff* falhasse na integração das mudanças, esse cenário seria contado como um *aFN* para o *CSDiff*. Falhar na integração significa que a ferramenta resultou em um código sintaticamente incorreto ou que não preserva os comportamentos esperados individualmente pelas mudanças de *left* e *right*. Esse procedimento também foi realizado para encontrar falsos negativos adicionados pelo *Diff3* em comparação com o *CSDiff*. Para verificar esses casos de falsos negativos, os códigos foram analisados manualmente.

4.2.5 PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

Além das 4 primeiras perguntas de pesquisa, foi criada uma outra forma de analisar os benefícios do *CSDiff* em relação ao *Diff3*. Para isso, foram definidas as 4 situações abaixo, e cada situação foi associada a uma pontuação, de forma que uma maior pontuação em um determinado arquivo ou cenário, indica um aumento na produtividade do desenvolvedor ao utilizar o *CSDiff* como ferramenta ao invés do *Diff3*. As situações foram selecionadas ao comparar visualmente as diferenças de resultados entre as ferramentas e suas pontuações foram escolhidas arbitrariamente pelo autor.

A identificação das situações foi feita comparando manualmente, em cada arquivo de cada cenário, os conflitos relatados pela ferramenta *CSDiff* com os relatados pela ferramenta *Diff3*, bem como foi comparado seus resultados com o resultado final do *merge*, nos casos onde um deles não relatava conflitos. Essa análise foi possível pois, apesar de a amostra ter sido consideravelmente grande (mais de 3000 cenários de *merge*), houveram somente 32 cenários (com um total de 255 arquivos) onde tais comportamentos fossem possíveis de acontecer. Isso será melhor explicado na seção 4.4.

Essa análise é importante pois considera também falsos conflitos que ocorrem quando as duas ferramentas (*CSDiff* e *Diff3*) acusam conflitos, diferentemente do conceito de Falso Positivo Adicionado definido em 4.1, que considera somente conflitos quando uma das ferramentas acerta o resultado do merge. Além disso, consideramos outras situações que são definidas a seguir e não são consideradas nas outras perguntas de pesquisa. Dessa forma, conseguimos analisar com um pouco mais de detalhes as vantagens e desvantagens de se utilizar o *CSDiff* ao invés do *Diff3*.

4.2.5.1 Conflito Reduzido

Um Conflito Reduzido é definido como um conflito que ocorre na ferramenta *Diff3* e na ferramenta *CSDiff*, mas que no resultado do *CSDiff* esse conflito está consideravelmente reduzido (possui um tamanho menor). Dessa forma, a pontuação escolhida para o aumento de produtividade nessa situação foi +1, dado que uma redução no tamanho de um conflito implica numa resolução mais rápida pelo desenvolvedor.

4.2.5.2 Conflito Resolvido

Um Conflito Resolvido significa um conflito que é relatado pela ferramenta *Diff3*, mas não é relatado pela ferramenta *CSDiff*, e além disso, o resultado do *CSDiff* para o bloco de código associado a esse conflito é o mesmo resultado observado no resultado final do merge. Este é a melhor das situações analisadas nessa pergunta de pesquisa, pois indica um conflito a menos para o desenvolvedor resolver (um conflito resolvido automaticamente pelo *CSDiff* mas nao pelo *Diff3*). Dessa forma, escolhemos a pontuação +2 para essa situação.

4.2.5.3 Falso Conflito Real

Um Falso Conflito Real é definido como um conflito que não existe no *Diff3*, mas que existe no *CSDiff* do arquivo em questão. Essa situação pode ser causada por conflito que foi automaticamente resolvido pelo *Diff3*, mas não pelo *CSDiff*, indicando um conflito a mais para o desenvolvedor resolver caso ele utilize o *CSDiff* ao invés do *Diff3*. Por isso, escolhemos a pontuação -1 para essa situação.

4.2.5.4 Falso Negativo Real

Um Falso Negativo Real nesse contexto indica um conflito que foi relatado no *Diff3*, mas não no *CSDiff* (indicando que o *CSDiff* resolveu um conflito que o *Diff3* não resolveu), e o resultado dessa resolução de Conflito é diferente do resultado observado no *merge* do repositório. Essa é a pior situação possível, dado que quando um conflito é resolvido de forma errada, o código resultante poderá apresentar comportamento inesperado ou não funcionar, além de possivelmente passar despercebido pelo desenvolvedor. Por isso, escolhemos a pontuação -2 para esse caso.

4.3 AMOSTRA

Como amostra para essa pesquisa, utilizamos os critários também usados em trabalhos anteriores, cuja ideia é procurar por projetos com boas chances de ter commits de merge, ou seja, projetos com muitos contribuidores e muitas estrelas no GitHub. Foram selecionados 10 projetos open source majoritariamente escritos em Python, cada um com mais de 13000 estrelas no github e mais de 350 contribuidores cada. Os números podem ser conferidos na Figura 4.1.

Projeto	Estrelas	Contribuidores
matplotlib	16.9k	1264
tensorflow	172k	3318
certbot	29.7k	462
flask	62k	693
ipython	15.7k	821
salt	13.1k	1414
scrapy	46.3k	507
sentry	33.4k	613
tornado	21k	370

Tabela 4.1 Projetos selecionados para o experimento

4.4 METODOLOGIA

Para comparar o *CSDiff* e o *Diff3*, primeiro mineramos os commits de *merge* para os 10 projetos escolhidos, considerando um intervalo de um ano (entre 1/1/2021 e 1/1/2022). Esta mineração foi feita utilizando o *miningframework*, como nos trabalhos anteriores e, com o objetivo de analisar o novo algoritmo (Figura 3.1.2) em comparação com o já existente (Figura 2.3), bem como analisar a influência da escolha de certos separadores para a linguagem Python, essa mineração foi feita 4 vezes, a saber:

(1) Algoritmo antigo, com os separadores "( ) , :". Chamemos de *CSDiff+*
(2) Algoritmo antigo, com os separadores "( ) ,". Chamemos de *CSDiff-*
(3) Algoritmo novo, que considera indentação, com os separadores "( ) , :". Chamemos de *CSDiffI+*
(4) Algoritmo novo, que considera indentação, com os separadores "( ) ,". Chamemos de *CSDiffI-*

Decidimos testar a influência do separador ":" (dois pontos) pois em Python este separador usualmente vem seguido de uma mudança de indentação, então nesses casos, o algoritmo novo irá adicionar linhas marcadoras de forma redundante (pois durante a execução irá detectar tanto o separador quanto a mudança de indentação, ao invés de somente um dos dois).

Após a mineração, o *miningframework* executa automaticamente as duas ferramentas em todos os arquivos de cada cenário de merge minerado, e em seguida cria uma tabela com dados relevantes como número de conflitos por ferramenta, por cenário, número de arquivos com conflitos, etc. Para este passo, somente foi necessário utilizar o CSDiffModule, um módulo do *miningframework* já existente e que já foi utilizado também em estudos anteriores [11].

Além disso, alguns scripts feitos em Bash pelo autor foram necessários para complementar os dados que não eram obtidos diretamente destas tabelas geradas pela ferramenta. Todos esses scripts estão disponibilizados em um repositório público.[2]

Para contar conflitos por arquivo, cenário, etc., os scripts buscam por marcadores de conflito nos textos dos arquivos. Para identificar se as ferramentas deram o mesmo resultado, ou

[2]https://github.com/zegabr/miningframework/tree/test-branch

resultado idêntico ao do merge do repositório, comparamos textualmente (ignorando espaços em branco) o arquivo de *merge* do repositório, e os arquivos resultante da execução de cada ferramenta no cenário de *merge* correspondente.

Todos esses passos foram executados localmente em uma máquina operando com sistema operacional Ubuntu 22.04.2, com 16GB de memória RAM e um processador Intel Core i7.

4.5 RESULTADOS

Para a nossa amostra, foram coletados 3788 cenários de *merge* (contendo 968 arquivos no total). Destes, apenas 32 cenários foram considerados (contendo 255 arquivos no total), por possuírem resultado diferente entre as duas ferramentas. Todos os casos onde o resultado do *CSDiff* era o mesmo do *Diff3* foram deletados pois não fariam diferença para a análise comparativa (essa filtragem foi feita por um dos scripts mencionados).

Como poderemos ver nas seções seguintes, foi observado uma peculiaridade na nossa amostra. Um dos projetos escolhidos, o matplotlib, possui uma quantidade muito maior de conflitos e *aFPs* relatados do que todos os outros projetos juntos. Ao remover as quantidades relativas a este projeto, obtemos um resultado compatível com os resultados obtidos em [1], [11]. A causa desse problema será discutida posteriormente.

4.5.1 PP1: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de conflitos reportados em comparação ao *merge* puramente textual?

Para respoder essa pergunta, analisamos a quantidade de conflitos por cenário obtidos da execução do experimento. Os resultados para todas as execuções podem ser vistos na Tabela 4.2 e na Tabela 4.3

Tipo de conflito	***Diff3***	***CSDiff+***	***CSDiff-***	***CSDiffI+***	***CSDiffI-***
Conflitos	100	177	146	277	237
Arquivos com conflitos	52	58	52	74	76
Cenários com conflitos	30	24	23	24	26

Tabela 4.2 Quantidade de conflitos encontrados após execução do experimentos

Tipo de conflito	***Diff3***	***CSDiff+***	***CSDiff-***	***CSDiffI+***	***CSDiffI-***
Conflitos	59	51	51	54	52
Arquivos com conflitos	29	20	19	21	22
Cenários com conflitos	22	15	14	15	17

Tabela 4.3 Quantidade de conflitos encontrados após execução do experimentos, desconsiderando o projeto matplotlib

Observamos que, considerando todos os projetos (Tabela 4.2), a quantidade de conflitos

aumenta razoavelmente, apresentando um aumento mínimo de 46% para o *CSDiff*- e máximo de 177% para o *CSDiffI*+, o que é esperado dado que o *CSDiff* tende a quebrar os conflitos do *Diff3* em conflitos menores, devido a forma como ele processa os arquivos em seu algoritmo. O estudo anterior feito com Java [1] segue no mesmo caminho, apresentando um aumento de 35% para um conjunto menor de separadores e 105% para um conjunto maior de separadores.

Percebe-se também a influência do novo algoritmo no resultado. O *CSDiffI*+ e o *CSDiffI*- demonstraram aumentos de quantidade de conflitos muito maiores que os *CSDiff*+ e *CSDiff*-, indicando que o novo algoritmo não é tão benéfico quando comparado com a versão já existente.

Além disso, podemos ver na Tabela 4.3 que ao desconsiderar o projeto matplotlib, obtemos uma redução pequena no número de conflitos em relação ao *Diff3* (aproximadamente 10% menos conflitos). Isso é um resultado inesperado considerando os resultados anteriores. Isso pode ter sido causado pelo fato de que o matplotlib representa aproximadamente 70% da amostra (considerando quantidade de conflitos), indicando que sem o matplotlib a amostra talvez não seja representativa o suficiente para tirar conclusões em relação a quantidade de conflitos.

4.5.2 PP2: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de cenários com conflitos reportados em comparação ao *merge* puramente textual?

Observando a quarta linha da Tabela 4.2, conseguimos também responder a esta pergunta. Notamos que há uma redução na quantidade de cenários com conflitos, com a maior redução ocorrendo para o *CSDiff*- (23%) e a menor para o *CSDiffI*+ (13.3%), seguindo também a mesma ordem de grandeza reportado por Souza [11] em seu estudo com TypeScript e Ruby (apresentando redução de 13.78% e 16.32%, respectivamente).

Também relatamos uma redução na quantidade de arquivos com conflitos, entretanto percebemos que, para o projeto matplotlib, o oposto ocorre, como podemos ver comparando a terceira linha das duas tabelas. Essa peculiaridade ocorre devido a uma maior ocorrência de conflitos por arquivo nesse projeto (ver Tabela 4.4), que faz com que exista uma chance maior de ocorrências de certos problemas de alinhamento que ocorrem de forma não intuitiva no algoritmo do *Diff3*. Esses problemas são descritos por Khanna [5].

Não é trivial definir uma única causa específica para o maior número de falsos conflitos que acontecem no matplotlib, porém, uma das possíveis causas é o fato de que o projeto possui certos arquivos que possuem muita repetição de código. Para exemplificar, considere o arquivo `_axes.py`.[3] Esse arquivo tem em seu corpo muitas cópias da string “(x, y“, que apesar de serem relativas a definições de métodos (ou chamadas de métodos) diferentes, serão modificadas pelo *CSDiff*, se separando de seus métodos (e de outros argumentos do mesmo método) e gerando linhas consecutivas que, para o *Diff3*, são iguais. Isso faz com que, caso hajam modificações em métodos que possuem essa sequência em sua definiçao (sejam a adição ou remoção de argumentos, adição ou remoção de métodos etc.), o algoritmo do Diff3 se "confunda"na hora de decidir quais sequências de “(x, y“ fazem parte de quais métodos, causando

[3] `https://github.com/matplotlib/matplotlib/blob/main/lib/matplotlib/axes/_axes.py`

assim falsos conflitos.

Uma possível solução a ser testada, seria numerar os marcadores inseridos pelo *CSDiff*, visto que isso possivelmente reduziria a chance de o *Diff3* alinhar de forma errada os marcadores que atualmente são adicionados sem distinção entre si. Essa ideia não foi testada pelo autor devido ao tempo alocado para a pesquisa.

Projeto	**Blocos de Diff**
matplotlib	385
tensorflow	20
flask	7
ipython	1
salt	78
scrapy	25

Tabela 4.4 Quantidade de blocos de conflitos encontrados após execução do experimentos

4.5.3 PP3: A nova solução de *merge* não estruturado, utilizando indentação, reduz a quantidade de falsos conflitos e cenários com falsos conflitos reportados (falsos positivos) em comparação ao *merge* puramente textual?

Para responder a PP3, observamos a quantidade de *aFP* de cada ferramenta. O que importa para a nossa investigação aqui é o conjunto de falsos positivos reportados (*aFP*) por uma ferramenta, mas não pela outra. Isso ajuda a demonstrar uma possível desvantagem de uma ferramenta em relação a outra.

	CSDiff+	***Diff3***
aFP cenários	1	3
aFP arquivos	16	7
aFN cenários	4	0
aFN arquivos	4	0

Tabela 4.5 Resultados do *CSDiff+* em comparação ao *Diff3*

Analisando as tabelas 4.5, 4.6, 4.7 e 4.8, conseguimos notar uma pequena desvantagem da ferramenta *Diff3* em relação ao *CSDiff*, visto que enquanto a ferramenta *Diff3* relata falsos positivos em aproximadamente 10% dos cenários, o *CSDiff* relata em aproximadamente 3.2%, configurando uma redução de aproximadamente 68% ao utilizar o *CSDiff*.

De forma similar, fizemos uma análise de *aFP* por arquivo, ao invés de cenário, por existir a possibilidade de um arquivo conter um *aFP* de uma ferramenta mas o cenário como um todo ter também verdadeiros positivos, o que faz com que o cenário como um todo não seja contabilizado como um *aFP*. Notamos então um aumento considerável de arquivos *aFP* (arquivos em que uma ferramenta apresenta conflito enquanto a outra acertou o resultado do *merge*) para

	CSDiff -	***Diff3***
aFP cenários	1	4
aFP arquivos	12	7
aFN cenários	2	0
aFN arquivos	2	0

Tabela 4.6 Resultados do *CSDiff* - em comparação ao *Diff3*

a ferramenta *CSDiff* em relação ao *Diff3*. Isso tem relação direta com o problema discutido em 4.5.2, visto que a grande quantidade de blocos de conflitos nos cenários de *merge* desse projeto tende a causar mais falsos positivos.

Apesar disso, e em concordância com os resultados apresentados até agora, vemos uma vantagem da versão *CSDiff* - em relação as outras três versões, pois observa-se que a quantidade de arquivos *aFP* relatados para esta é aproximadamente metade da quantidade relatada nas outras.

	CSDiffI+	***Diff3***
aFP cenários	1	3
aFP arquivos	30	6
aFN cenários	5	0
aFN arquivos	6	0

Tabela 4.7 Resultados do CSDiffI+ em comparação ao *Diff3*

	CSDiffI-	***Diff3***
aFP cenários	1	3
aFP arquivos	31	7
aFN cenários	4	0
aFN arquivos	4	0

Tabela 4.8 Resultados do CSDiffI- em comparação ao *Diff3*

4.5.4 PP4: A nova solução de *merge* não estruturado, utilizando indentação, amplia a possibilidade de comprometer a corretude do código, por aumentar o número de integrações de mudanças que interferem uma na outra, sem reportar conflitos (falsos negativos), além de aumentar cenários com falsos negativos?

Também se mostra importante considerar como a ferramenta se comporta ao tentar resolver conflitos, visto que não é interessante se a ferramenta, apesar de reduzir a quantidade de falsos positivos relatados, resolver muitos conflitos de forma errada. Dessa forma, consideramos as

mesmas tabelas analisadas em resultado da PP3, onde temos as quantidades de *aFN* para cada ferramenta. Como explicado anteriormente, um possível falso negativo para uma determinada ferramenta ocorre quando ela não reporta conflito e erra o resultado final do *merge* quando comparado com o arquivo do repositório, enquanto a outra ferramenta reporta conflito.

Com o objetivo de verificar esses valores, também foi feita uma contagem manual para confirmar quais desses casos eram falsos negativos de verdade. Como podemos ver na Tabela 4.6, a versão *CSDiff* - possui o melhor resultado, relatando apenas 2 falsos negativos, um por arquivo, um em cada cenário.

Notamos que em todas as 4 versões consideradas, há um pequeno aumento de *aFN* relatados pela ferramenta *CSDiff*. Apesar disso, esse número é relativamente pequeno, como vemos na Tabela 4.7 (que possue os maiores valores de *aFN* entre as 4 tabelas), somente 6 dos 255 arquivos considerados são *aFN*, ou seja, somente 2.35% dos arquivos que relataram comportamento diferente entre as ferramentas são arquivos contendo conflitos resolvidos de forma errada. Entretanto, observando em termos de cenários, 5 dos 32 cenários contém *aFNs*, representando aproximadamente 15.6% dos cenários onde as ferramentas se comportaram de forma diferente.

4.5.5 PP5: A nova solução de *merge* não estruturado, utilizando indentação, demonstra um aumento de produtividade considerando o ato de resolver conflitos de merge?

Como análise alternativa ao experimento utilizado nos trabalhos anteriores, foi feita uma análise manual para contar situações que encontramos quando comparamos os resultados das ferramentas. Essa análise foi possível pois a quantidade de arquivos a ser analisados para essa contagem foi relativamente pequena.

Os conceitos utilizados nessa análise foram explicados na seção 4.2.5. Para contar cada situação, cada arquivo de cada cenário foi analisado, e em seguida, a cada situação detectada pelo autor, era adicionado um texto marcador como comentário no código. Por exemplo, ao procurar pela situação Falso Conflito Real, os arquivos resultantes do *CSDiff* e do *Diff3* eram comparados e, para cada conflito que existia no resultado do *CSDiff*, mas não existia no *Diff3*, adicionava-se um comentário com a palavra "FCR" (Falso Conflito Real). Outras palavras foram usadas para marcar as ocorrências de cada uma das outras situações.

Após essa marcação, foram utilizados scripts em *Bash* utilizando a ferramenta *ripgrep*[4] para fazer a contagem de cada situação. Como definido na seção 4.2.5, cada situação estava associada a uma pontuação, que era somada a pontuação total (que nomeamos aqui de "Aumento de produtividade") do arquivo, cenário e projeto no qual a situação se encontrava. Todos os scripts usados pelo autor se encontram num repositório público no Github.[5]

Na tabela 4.9 podemos ver os resultados por projeto, cuja fórmula para os valores foi simplesmente a soma das quantidades encontradas de cada situação, multiplicadas por sua pontuação.

Por exemplo, digamos que em um arquivo qualquer foram encontrados 0 Falsos Negativos Reais, 2 Conflitos reduzidos, 0 Conflitos resolvidos e 1 Falso Conflito Real, então o Aumento de

[4]https://github.com/BurntSushi/ripgrep

[5]https://github.com/zegabr/miningframework/tree/test-branch

produtividade desse arquivo é de 0 * (-2) + 2 * (1) + 0 * (2) + 1 * (-1) = 1, indicando um aumento de produtividade baixo porém não negativo. Toda pontuação positiva é considerada uma boa pontuaçao nessa análise, enquanto que pontuações negativas indicam uma desvantagem em utilizar o *CSDiff* ao invés do *Diff3* e pontuação nula indica que, não houve melhora nem piora considerável na produtividade da pessoa que estaria possivelmente resolvendo os conflitos do arquivo em questão. Perceba que nesse exemplo descrito, pode-se dizer que o resultado foi positivo pois apesar de a ferramenta ter gerado um conflito que não existia, ela também reduziu o tamanho de dois outros conflitos possivelmente maiores que o novo conflito.

Aumento de Produtividade/Projeto	***CSDiff+***	***CSDiff-***	***CSDiffI+***	***CSDiffI-***
matplotlib	-36	-8	-107	-75
tensorflow	20	12	19	19
flask	6	4	5	6
ipython	1	2	1	1
salt	4	14	12	2
scrapy	22	19	14	20
total	17	43	-56	-27
total sem matplotlib	53	51	51	48

Tabela 4.9 Pontuação obtida da análise do aumento de produtividade

Nos resultados da tabela, nota-se para a nova versão proposta (*CSDiffI+* e *CSDiff-*) uma desvantagem considerável em relação a versão já existente (*CSDiff+* e *CSDiff-*), considerando todos os projetos, e uma desvantagem pequena ao desconsiderar grande pontuação negativa causada pelo matplotlib, dada a quantidade de Falsos Conflitos Reais que ocorrem nele. Apesar disso, podemos ver uma vantagem clara de se usar as versões com conjunto menor de separadores em comparação com as versões com conjunto maior de separadores. Isso reforça que a nova versão proposta não é tão vantajosa, bem como o uso de ":" nos separadores também causa uma leve desvantagem.

4.6 DISCUSSÃO

Através dos resultados para as perguntas de pesquisa respondidas na seção anterior, conseguimos observar mais detalhes sobre o comportamento do *CSDiff* proposto pori Clementino [1] e extendido por Souza [11]. Pudemos verificar sua performance para integrar códigos para a linguagem Python, dadas as modificações explicitadas na seção 4.

A quantidade de conflitos relatados pelo *CSDiff* segue maior que a quantidade relatada pelo *Diff3*, como esperado devido a forma como o *CSDiff* se comporta (quebrando conflitos grandes em conflitos menores). Nota-se uma aumento ainda maior desses conflitos ao utilizar o separador ":" em Python, devido ao fato de que normalmente um separador ":" vem precedido de um ")" (outro dos separadores utilizados nos conjuntos), fazendo com que o *CSDiff* adicione marcadores mais de uma vez para uma mesma linha de código, o que se agrava mais ainda caso

consideremos a indentação, dado que normalmente o ":" vem seguido de uma mudança de indentação.

Notamos uma desvantagem em relação ao número de conflitos ao utilizar a nova versão que considera indentação no lugar da versão já existente, onde a nova versão apresenta aproximadamente duas vezes mais conflitos que a versão já existente, o que indica que ainda há um grande espaço para melhora, visto que a melhor das versões testadas nessa pesquisa apresentou Falsos Positivos Adicionados em aproximadamente 3.2% dos arquivos testados, e ao mesmo tempo, apresentou Falsos Negativos Adicionados em 2.35% dos arquivos testados. Como mencionado por Souza [11], uma forma de reduzir tais falsos conflitos seria fazer com que o *CSDiff*, numere os marcadores adicionados (ou seja, ao invés de usar '$$$$$$$$', usar '$$$$<numeração>$$$$' ou '$$$$$$$$<numeração>'). Dessa forma, o algoritmo de alinhamento do *Diff3* obterá uma maior subsequência comum, o que potencialmente reduzirá a quantidade falsos positivos causados por modificações envolvendo códigos que ficam repetidos ao serem transformados pelo *CSDiff*.

Por fim, outro trabalho futuro com potencial de melhorar a performance do *CSDiff* seria testar outras ferramentas de diff. Durante os testes foi descoberto que, para a maioria dos casos de Falso Conflitos Reais detectados durante a análise da PP5, a ferramenta *KDiff3*,[6] ao ser usada no lugar do *Diff3*, resolvia os conflitos corretamente. Entretanto, não foi possível utilizar o *KDiff3* nos scripts, pois o programa sempre abria uma GUI quando sobravam conflitos não resolvidos, requisitando assim intervenção do usuário. Por isso, seria interessante avaliar outras ferramentas de diff que poderiam ser utilizadas no lugar do próprio *Diff3*, ou com o *Diff3*, substituindo o uso interno da ferramente *Diff* através da flag "–diff-program". Alguns exemplos de ferramentas a serem testadas seria o vimdiff e o wdiff.

4.7 AMEAÇAS A VALIDADE

Os resultados aqui mostrados, como esperado, possuem potenciais ameaças a validade. Pelo fato de termos seguido a mesma metodologia que alguns estudos anteriores, apontamos algumas ameaças a validade semelhantes. Primeiro, os projetos utilizados para o estudo foram projetos abertos do GitHub, que podem ter sofrido alteração em seu histórico de commits, que por sua vez, pode acarretar em perdas de cenários de *merge*. Segundo, a avaliação manual feita pelo autor na análise do PP5 pode conter erros e não foi revisada por outras pessoas. Também não se teve validações externas além das opiniões dos orientadores e além disso, é uma análise muito arriscada e aproximada, dado que o conceito de produtividade não é trivial de se medir.

Adicionalmente temos o fato de que os scripts criados e utilizados pelo autor para ajudar na análise da pesquisa podem conter bugs. Por fim, esse estudo foca na linguagem de programação Python, logo não é garantido que pesquisas feitas em cima de outras linguagens deixem de demonstrar resultados completamente diferentes. Além disso, apesar de termos utilizado uma amostra de 10 projetos relativamente grandes e minerado commits de um intervalo relativamente grande, somente 6 dos projetos mostraram conflitos de *merge* nesse intervalo, o que pode influenciar na obtençao de uma quantidade de dados suficiente.

[6] https://github.com/KDE/kdiff3

```
# ========================== diff3 result ==========================
(...)
def onselect(xmin, xmax):
    indmin, indmax = np.searchsorted(x, (xmin, xmax))
    indmax = min(len(x) - 1, indmax)

    region_x = x[indmin:indmax]
    region_y = y[indmin:indmax]
<<<<<<< left.py

    if len(region_x) >= 2:
        line2.set_data(region_x, region_y)
        ax2.set_xlim(region_x[0], region_x[-1])
        ax2.set_ylim(region_y.min(), region_y.max())
        fig.canvas.draw_idle()

=======

    if len(region_x) >= 2: # CReduzido
        line2.set_data(region_x, region_y)
        ax2.set_xlim(region_x[0], region_x[-1])
        ax2.set_ylim(region_y.min(), region_y.max())
        fig.canvas.draw()
>>>>>>> right.py
(...)
# ========================== csdiff result ==========================
(...)
def onselect(xmin, xmax):
    indmin, indmax = np.searchsorted(x, (xmin, xmax))
    indmax = min(len(x) - 1, indmax)

    region_x = x[indmin:indmax]
    region_y = y[indmin:indmax]

    if len(region_x) >= 2:
        line2.set_data(region_x, region_y)
        ax2.set_xlim(region_x[0], region_x[-1])
        ax2.set_ylim(region_y.min(), region_y.max())
<<<<<<< left.py
        fig.canvas.draw_idle
=======
        fig.canvas.draw
>>>>>>> right.py
()
(...)
```

Figura 4.1 Código exemplificando um Conflito Reduzido

```
# ========================= diff3 result =========================
(...)
 def _smooth_labels():
    num_classes = math_ops.cast(array_ops.shape(y_true)[-1], y_pred.dtype)
    return y_true * (1.0 - label_smoothing) + (label_smoothing /
   num_classes)

<<<<<<< left.py
  y_true = smart_cond.smart_cond(
      label_smoothing, _smooth_labels, lambda: y_true)
  return K.categorical_crossentropy(y_true, y_pred, from_logits=from_logits
    , axis=axis)
=======
# CResolvido
  y_true = smart_cond.smart_cond(label_smoothing, _smooth_labels,
                                 lambda: y_true)
  return K.categorical_crossentropy(y_true, y_pred, from_logits=from_logits
    )
>>>>>>> right.py
(...)
# ========================= csdiff result =========================
(...)
  def _smooth_labels():
    num_classes = math_ops.cast(array_ops.shape(y_true)[-1], y_pred.dtype)
    return y_true * (1.0 - label_smoothing) + (label_smoothing /
   num_classes)

  y_true = smart_cond.smart_cond(label_smoothing, _smooth_labels,
                                 lambda: y_true)
  return K.categorical_crossentropy(y_true, y_pred, from_logits=from_logits
    , axis=axis)
(...)
```

Figura 4.2 Código exemplificando um Conflito Resolvido

```
# ========================= diff3 result =========================
(...)
       if np.ndim(data) == 0 and isinstance(self, ScalarMappable):
            # This block logically belongs to ScalarMappable, but can't be
            # implemented in it because most ScalarMappable subclasses
   inherit
            # from Artist first and from ScalarMappable second, so
            # Artist.format_cursor_data would always have precedence over
            # ScalarMappable.format_cursor_data.
            n = self.cmap.N
            if np.ma.getmask(data):
                return "[]"
            normed = self.norm(data)
            if np.isfinite(normed):
                # Midpoints of neighboring color intervals.
                neighbors = self.norm.inverse(
                    (int(self.norm(data) * n) + np.array([0, 1])) / n)
                delta = abs(neighbors - data).max()
(...)
# ========================= csdiff result =========================
(...)
        if np.ndim(data) == 0 and isinstance(self, ScalarMappable):
            # This block logically belongs to ScalarMappable, but can't be
            # implemented in it because most ScalarMappable subclasses
   inherit
            # from Artist first and from ScalarMappable second, so
            # Artist.format_cursor_data would always have precedence over
            # ScalarMappable.format_cursor_data.
            n = self.cmap.N
            if np.ma.getmask(data):
                return "[]"
<<<<<<< left.py
            normed = self.norm(data)
            if np.isfinite(normed):
                # Midpoints of neighboring color intervals.
                neighbors = self.norm.inverse
# FCR
=======
            normed = self.norm
>>>>>>> right.py
(data)
<<<<<<< left.py
                    (int(self.norm
=======
# FCR
            if np.isfinite(normed):
                # Midpoints of neighboring color intervals.
                neighbors = self.norm.inverse(
                    (int(self.norm
>>>>>>> right.py
(data) * n) + np.array([0, 1])) / n)
                delta = abs(neighbors - data).max()
(...)
```

Figura 4.3 Código exemplificando um Falso Conflito Real

```
# ========================= diff3 result =========================
(...)
        # TextBox's text object should not parse mathtext at all.
        self.text_disp = self.ax.text(
<<<<<<< left.py
            self.DIST, 0.5, initial, transform=self.ax.transAxes,
            verticalalignment='center', horizontalalignment=ha)
=======
            self.DIST_FROM_LEFT, 0.5, initial,
            transform=self.ax.transAxes, verticalalignment='center',
            horizontalalignment='left', parse_math=False)
>>>>>>> right.py

        self._observers = cbook.CallbackRegistry()
(...)
# ========================= csdiff result =========================
(...)
        # TextBox's text object should not parse mathtext at all.
        self.text_disp = self.ax.text(
            self.DIST, 0.5, initial,
            transform=self.ax.transAxes, verticalalignment='center',
            horizontalalignment='left', # CaFN
<<<<<<< left.py
 horizontalalignment=ha
=======
# CReduzido
 parse_math=False
>>>>>>> right.py
)

        self._observers = cbook.CallbackRegistry()
(...)
```

Figura 4.4 Código exemplificando um Falso Negativo Real

CAPÍTULO 5

Trabalhos Relacionados

Na academia, vários trabalhos foram feitos para aprofundar os conhecimentos em cima das abordagens e ferramentas de *merge*. Entre eles, podemos citar inicialmente os dois trabalhos que serviram como base para esta pesquisa. Temos Clementino [1], que propôs uma nova ferramenta de *merge* textual que tenta simular uma abordagem semiestruturada utilizando os separadores da linguagem Java. Também temos Souza [11], que extende a mesma ferramenta e analisa os resultados dela ao ser executado em amostras de outras linguagens que ainda não tinham sido testadas (TypesCript e Ruby).

Em um estudo cujo objetivo era entender as peculiaridades do *Diff3*, Khanna [5] formaliza, através de fórmulas e teoremas, algumas propriedades particulares do algoritmo do *Diff3*, que inicialmente parecem ser intuitivas e corretas, mas que no fim não são.

Considerando a área de *merge* semiestruturado, temos Apel [6], que propôe e analisa a ferramenta *FSTMerge*. Temos também o trabalho de Cavalcanti [9], onde ele discorre sobre *merge* semiestruturado e seus resultados quando comparados ao uso do *merge* não estruturado levando em consideração a redução de conflitos. Em outra publicação, Cavalvanti [8] faz uma comparação entre o *merge* semiestruturado e o estruturado. Ele aponta que o primeiro tende a apresentar mais falsos positivos, enquanto o segundo apresenta mais falsos negativos. Essa observação sugere a possibilidade de explorar um equilíbrio entre as diferentes ferramentas de abordagens distintas, buscando um meio termo entre uma abordagem textual e as abordagens semiestruturadas e estruturadas.

Por fim, em um outro trabalho, Accioly [13] conduz uma pesquisa com o objetivo de examinar e classificar os padrões de conflitos em projetos de código aberto escritos em Java, com o intuito de identificar quais tipos de padrões na estrutura do código estão associados a diferentes tipos de conflitos.

CAPÍTULO 6

Conclusão

Propomos neste artigo uma modificação sobre uma ferramenta já existente, o *CSDiff* [1], visando analisar sua performance para a linguagem Python. Testamos a versão já existente, criada em [11], e a versão com a modificação proposta neste trabalho, que faz com que a ferramenta considere mudanças de indentação na tentativa de separar escopos de programa Python durante a execução do *CSDiff*. Analisamos também a influência de utilizar dois conjuntos de separadores da linguagem. Seguimos a mesma metodologia utilizada pelos dois trabalhos, com adição de uma nova análise de aumento de produtividade criada pelo autor.

Encontramos resultados similares aos das pesquisas anteriores em relação ao aumento na quantidade de conflitos e redução na quantidade de cenários com conflitos ao comparar os resultados da ferramenta *Diff3* com os resultados da ferramenta *CSDiff*. Considerando a adição de cenários Falsos Positivos, vemos uma pequena desvantagem do *Diff3* em relação ao *CSDiff*, mas o oposto ocorrendo para o número de arquivos. Considerando Falsos Negativos adicionados, notamos uma pequena desvantagem do *CSDiff* em relação ao *Diff3* ao considerar o número de arquivos, mas uma grande desvantagem ao considerar o número de cenários.

Por fim, ao analisar o aumento de produtividade, como definido na seção 4.2.5, notamos um aumento de produtividade maior na versão que não considera indentação, e utilizando um conjunto menor de separadores.

Bibliografia

[1] J. Clementino, P. Borba e G. Cavalcanti, "Textual merge based on language-specific syntactic separators," em *35th Brazilian Symposium on Software Engineering (SBES 2021)*, 2021, pp. 243–252.

[2] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, v. 28, n. 5, pp. 449–462, 2002. DOI: 10.1109/TSE.2002.1000449.

[3] Y. Brun, R. Holmes, M. D. Ernst e D. Notkin, "Proactive Detection of Collaboration Conflicts," em *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, sér. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 168–178, ISBN: 9781450304436. DOI: 10.1145/2025113.2025139. endereço: https://doi.org/10.1145/2025113.2025139.

[4] C. Brindescu, I. Ahmed, C. Jensen e A. Sarma, "An empirical investigation into merge conflicts and their effect on software quality," *Empirical Software Engineering*, v. 25, n. 1, pp. 562–590, 2020.

[5] S. Khanna, K. Kunal e B. C. Pierce, "A formal investigation of diff3," em *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 2007, pp. 485–496.

[6] S. Apel, J. Liebig, B. Brandl, C. Lengauer e C. Kästner, "Semistructured Merge: Rethinking Merge in Revision Control Systems," em *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, sér. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 190–200, ISBN: 9781450304436. DOI: 10.1145/2025113.2025141. endereço: https://doi.org/10.1145/2025113.2025141.

[7] G. Cavalcanti, P. Accioly e P. Borba, "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment," em *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–10. DOI: 10.1109/ESEM.2015.7321191.

[8] G. Cavalcanti, P. Borba, G. Seibt e S. Apel, "The Impact of Structure on Software Merging: Semistructured Versus Structured Merge," em *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1002–1013. DOI: 10.1109/ASE.2019.00097.

[9] G. Cavalcanti, P. Borba e P. Accioly, "Evaluating and Improving Semistructured Merge," *Proc. ACM Program. Lang.*, v. 1, n. OOPSLA, out. de 2017. DOI: 10.1145/3133883. endereço: https://doi.org/10.1145/3133883.

[10] A. Robbins, *Gawk: Effective AWK Programming*. Free Software Foundation Boston, 2004.

[11] H. S. C. Souza, *Extensão e análise de performance da ferramenta de merge textual CSDiff para novas linguagens*, Graduation Thesis, 2021.

[12] A. Koc e A. U. Tansel, "A survey of version control systems," *ICEME 2011*, 2011.

[13] P. Accioly, P. Borba e G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source Java projects," *Empirical Software Engineering*, pp. 2051–2085, 2018.