Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Distribuição de Relatórios de *Bugs* para Desenvolvedores de *Software* Usando um Método Evolutivo Multipopulacional

Kannya Leal Araujo

Teresina-PI, 25 de fevereiro de 2022

Kannya Leal Araujo

Distribuição de Relatórios de *Bugs* para Desenvolvedores de *Software* Usando um Método Evolutivo Multipopulacional

Dissertação (Mestrado) apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ricardo de Andrade Lira Rabêlo

Teresina-PI

25 de fevereiro de 2022

FICHA CATALOGRÁFICA
Universidade Federal do Piauí
Sistema de Bibliotecas da UFPI – SIBi/UFPI
Biblioteca Setorial do CCN

A663d Araújo, Kannya Leal..
Distribuição de relatórios de Bugs para desenvolvedores de software usando um método evolutivo multipopulacional / Kannya Leal Araújo. – 2022.
73 f.

Dissertação (Mestrado) – Universidade Federal do Piauí, Centro de Ciências da Natureza, Pós-Graduação em Ciência da Computação, Teresina, 2022.
"Orientador: Prof. Dr. Ricardo de Andrade Lira Rabêlo".

1. Sistemas Operacionais. 2. Relatório de erros - Bugs. 3. Sistema Fuzzy. 4. Metodo Evolutivo Multipopulacional. I. Rabêlo, Ricardo de Andrade Lira. II.Título.

CDD 005.43

Bibliotecária: Caryne Maria da Silva Gomes. CRB/3-1461

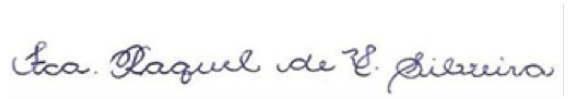
“Distribuição de Relatórios de Bugs para Desenvolvedores de Software Usando um Método Evolutivo Multipopulacional”

KANNYA LEAL ARAÚJO

Dissertação apresentada ao Programa de Pós–Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovada por:

Prof. Ricardo de Andrade Lira Rabêlo
(Presidente da banca examinadora)



Profa. Francisca Raquel de Vasconcelos Silveira
(Examinadora externa à instituição)

gov.br
Documento assinado digitalmente
Hermes Manoel Galvao Castelo Branco
Data: 08/03/2022 17:35:43-0300
Verifique em https://verificador.iti.br

Prof. Hermes Manoel Galvão Castelo Branco
(Examinador externo ao programa)

gov.br
Documento assinado digitalmente
Guilherme Amaral Avelino
Data: 03/03/2022 12:01:58-0300
Verifique em https://verificador.iti.br

Prof. Guilherme Amaral Avelino
(Examinador interno)

Teresina, 25 de fevereiro de 2022

A Deus, ao meu esposo Dalmo, meu filho Dalton Rafael, minha mãe Aldenívia e minha irmã Cinthya,
por sempre estarem comigo em todos os momentos.

Agradecimentos

Agradeço a Deus, por me guiar com sua graça permitindo a concretização de mais uma realização.

Agradeço aos meus pais, José e Aldenívia, por me ensinarem a ser resistente e determinada frente aos obstáculos da vida.

Ao meu esposo Dalmo e meu filho Dalton Rafael, pelo apoio, carinho, compreensão, e por ter me acompanhado durante todas as alegrias e dificuldades que enfrentei.

Aos meus irmãos, por me apoiarem e incentivarem durante todo o mestrado.

Agradeço ao meu orientador, Prof. Ricardo Lira, por ter acreditado em mim e pela oportunidade que me ofereceu. Obrigada por todos os conselhos, paciência, dedicação e disponibilidade determinantes para a concretização deste trabalho.

Ao professor Guilherme Avelino, por todos os conselhos e direcionamentos durante a realização da pesquisa.

Aos amigos Luiz Fernando Mendes, Euclydes Melo e Matheus Campanhã, que participaram da elaboração da abordagem que originou este trabalho. Em especial ao meu amigo Luiz Fernando, que foi chave na construção deste trabalho, obrigada pela ajuda, apoio, incentivo e disponibilidade.

Aos professores Guilherme Avelino, Hermes Manoel Branco e Francisca Raquel Silveira, por terem aceitado participar da banca e pelas relevantes contribuições dadas ao trabalho.

A todos os professores do PPGCC, pela formação e conhecimentos que me proporcionaram durante todo o mestrado.

"A nossa maior glória
não reside no fato de nunca cairmos,
mas sim em levantarmos
sempre depois de cada queda."
(Confúcio)

Resumo

A complexidade de um *software* aumenta à medida em que seu tamanho, também, aumenta. Esse aumento pode refletir um crescimento na quantidade de *bugs* ou falhas, tornando o processo manual de atribuição de *bugs* mais complexo, demorado e sujeito a erros. Técnicas existentes, de automatização desse processo, atribuem relatórios de *bugs* usando somente dados dos relatórios corrigidos anteriormente. Isso pode resultar em atribuições a desenvolvedores inativos. Adicionalmente, uma parcela significativa das atribuições normalmente não considera a carga de trabalho dos desenvolvedores, podendo sobrecarregar alguns e tornar o processo de correção mais demorado. Este trabalho propõe uma abordagem para distribuição de relatórios de *bug* que combina a experiência e as atividades recentes dos desenvolvedores, bem como, sua carga de trabalho. Quando um novo relatório é recebido, estima-se o esforço para corrigir o *bug* a partir de *bugs* semelhantes e calcula-se a afinidade de cada desenvolvedor com o arquivo que contém o erro utilizando um sistema de Inferência *Fuzzy*. Posteriormente, é utilizado a meta-heurística *Golden Ball* para atribuir esses relatórios aos desenvolvedores de acordo com a afinidade e carga de trabalho. Resultados experimentais mostram que, quando comparados com um algoritmo de força bruta, a abordagem proposta atinge valores ideais para alocação na maioria dos casos (75% dos cenários analisados). A abordagem, também, obteve médias significativamente melhores em 92,30% das instâncias quando comparado a um Algoritmo Genético e 84,61% quando comparado com um Algoritmo Genético Distribuído, sendo que em apenas 23,07% dos casos não houve uma diferença significativa entre as técnicas.

Palavras-chaves: Triagem de *Bugs*, Relatório de *Bug*, *Stack Trace*, Carga de Trabalho do Desenvolvedor, Afinidade com Arquivo de Erro, Sistema *Fuzzy*, Método Evolutivo Multipopulacional, *Golden Ball*.

Abstract

Existing approaches assign bug reports using only data from previously fixed reports. This can result in assignments to inactive developers, as well as not considering newcomers. A significant portion of assignments typically do not consider the workload of developers, which can overwhelm some and make the revision/debugging/correction process more time-consuming.This work proposes an approach for distributing bug reports that combines the experience and recent activities of developers,as well as their workload. When a new report is received, the effort to fix the bug based on similar error is estimated and each developer's affinity with the file containing the bug is calculated using a Fuzzy Inference system. Subsequently, the Golden Ball,a multi-population evolutionary meta-heuristic, is used to assign these reports to developers according to affinity and workload.Experimental results show that, when compared with a brute force algorithm, the proposed approach reach optimal values for allocation in most cases (75% of the analyzed scenarios). The approach also obtained significantly better averages in 92.30% of the instances when compared to a Genetic Algorithm and 84.61% when compared to a Distributed Genetic Algorithm, and in only 23.07% of the instances there was no significant difference between the techniques.

Keywords: Keywords: Bug Triage, Bug Report, Stack Trace, Developer Workload, Bug File Affinity, Fuzzy System, Multi-population Evolutionary Method, Golden Ball.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

GA	- Algoritmo Genético
GB	- *Golden Ball*
DGA	- Algoritmo Genético Distribuído
NT	- Quantidade de Times
PGA	- Algoritmos Genéricos Paralelos
PT	- Jogadores por Time

Sumário

Introdução

Contexto e Motivação

No contexto atual, em que fábricas de *software* têm que entregar produtos cada vez mais complexos, a priori, espera-se um aumento na presença de *bugs* ou falhas, produzindo resultados imprecisos ou comportamentos inesperados durante a execução do sistema de *software* (AKILA; ZAYARAZ; GOVINDASAMY, 2015; JIMOH et al., 2018). A presença de *bug* em um sistema de *software* afeta a confiabilidade e a qualidade do sistema, tornando primordial a correção desses *bugs* em tempo hábil (JEONG; KIM; ZIMMERMANN, 2009; LEE; SEO, 2020; BANI-SALAMEH; SALLAM et al., 2021).

Para apoiar o processo de correção dessas falhas, é comum o uso de sistemas de rastreamento de *bugs*, como o *Bugzilla (2021)*, *Jira (2021)*, *Redmine (2021)* e outros sistemas proprietários (TIAN; LO; SUN, 2013), nos quais testadores, desenvolvedores e usuários podem relatar os defeitos ou falhas ocorridos no sistema (SINGH, 2014; TIAN et al., 2015). Cada relatório de *bug* registrado deve ser analisado e atribuído a um desenvolvedor identificado como mais adequado para realizar a correção no menor tempo (BHATTACHARYA; NEAMTIU, 2010). É comum que essa análise seja realizada de forma manual e a partir de seu conteúdo seja efetivada a atribuição do relatório ao desenvolvedor mais adequado. A tarefa de atribuir manualmente um relatório pode ser muito complexa, demorada e sujeita a erros (BHATTACHARYA; NEAMTIU; SHELTON, 2012; ZHU et al., 2021; LEE; SEO, 2020; SAHU; LILHORE; AGARWAL, 2018).

Definição do Problema

Para ajudar a triagem de *bugs*, muitos estudos propuseram sistemas automatizados capazes de identificar possíveis desenvolvedores para corrigir um determinado *bug* (ZAIDI et al., 2020; ZHANG; GONG; VERSTEEG, 2013; KASHIWA; OHIRA, 2020; GUO et al., 2020). Todavia, alguns destes sistemas atribuem relatórios de *bugs* usando somente dados dos relatórios corrigidos anteriormente resultando em atribuições a desenvolvedores inativos (YIN; DONG; XU, 2018; MANI; SANKARAN; ARALIKATTE, 2019). Adicionalmente, uma parcela significativa das atribuições normalmente não considera a carga de trabalho dos desenvolvedores, podendo sobrecarregar alguns e tornar o processo de correção mais demorado. Isso motiva pesquisadores a continuarem estudando e desenvolvendo abordagens baseadas, principalmente, em recuperação de informações e aprendizado de máquina para automatizar total ou parcialmente o processo de atribuição de relatórios de *bugs* (SINGH, 2014; MOHAN; SARDANA et al., 2016; BHATTACHARYA; NEAMTIU, 2010).

Com isso, o problema a que este trabalho se propõe solucionar apresenta a seguinte definição formal: dado um conjunto de relatórios de *bugs* novos não distribuídos, encontrar o desenvolvedor com maior afinidade (i.e. *expertise*) para corrigir cada relatório, de forma a manter equilibrada a carga de trabalho, considerando informações disponíveis nos relatórios de *bugs*, como comentários dos desenvolvedores e código de erro, além de métricas obtidas por meio da análise do repositório de código fonte.

Visão Geral da Proposta

Para solucionar esse problema, o trabalho propõe uma abordagem baseada em meta-heurística que utiliza informações relacionadas à carga de trabalho dos desenvolvedores e dados sobre *bugs* previamente corrigidos para sugerir o desenvolvedor mais apto a corrigir um novo *bug*.

A abordagem proposta é dividida em três etapas: pré-processamento, cálculo das métricas e modelo meta-heurístico de otimização. Na etapa de pré-processamento são estimados o esforço para correção do novo *bug*, o conhecimento do domínio, a frequência e a recência. Na etapa de cálculo das métricas são realizados os cálculos da carga de trabalho de acordo com o esforço estimado para resolver cada relatório, aberto, atribuído a ele e da afinidade do desenvolvedor com o arquivo que está apresentando erro. Finalmente, na última etapa é efetivada a atribuição dos relatórios aos desenvolvedores mais aptos para corrigi-los.

Justificativa

A atribuição de relatórios de *bugs* consiste em designar o desenvolvedor mais apropriado para a correção de um determinado *bug*. Muitas técnicas têm sido aplicadas para automatizar total ou parcialmente a atribuição de relatórios, a maioria considera apenas o conhecimento dos desenvolvedores (MANI; SANKARAN; ARALIKATTE, 2019; TAMRAWI et al., 2011; KHATUN; SAKIB, 2018; ZHANG, 2020). No entanto, isso pode provocar uma sobrecarga de alguns desenvolvedores e manter outros ociosos. Com base nesta lacuna, a abordagem proposta neste trabalho atribui um conjunto de relatórios considerando duas métricas: a afinidade do desenvolvedor em relação ao arquivo que está apresentando o erro e a carga de trabalho de cada desenvolvedor. Essas métricas são utilizadas, pois a abordagem proposta visa não só priorizar o desenvolvedor com maior afinidade para corrigir o *bug*, como também, manter uma uniformidade da carga de trabalho dos integrantes do time de desenvolvimento.

Objetivos

Esse trabalho objetiva realizar a atribuição de relatório de *bug* de forma a alocar o desenvolvedor mais apropriado para a sua correção. Essa atribuição leva em consideração a afinidade e a carga de trabalho de cada desenvolvedor. Para alcançar o objetivo geral necessita-se atingir os seguintes objetivos específicos:

- Extrair dados, tipo de erro ou exceção e arquivo que apresenta o erro, do *stack trace* (GU et al., 2019) gerado no momento em que uma exceção não tratada é ativada;
- Estimar o esforço de correção do novo *bug*, por meio do histórico de outros resolvidos anteriormente associados ao mesmo tipo de erro;
- Propor um método de inferência *Fuzzy* de Mamdani (1977) de modo a calcular a afinidade de cada desenvolvedor em relação ao arquivo que contém o erro, a partir da frequência, recência e conhecimento do domínio;
- Aplicar uma meta-heurística evolutiva multipopulacional capaz de atribuir um conjunto de relatórios de *bug* para um grupo de desenvolvedores, utilizando afinidade e carga de trabalho de cada desenvolvedor;
- Realizar experimentos de atribuição de relatórios de *bugs*;
- Coletar e analisar dados do experimento para avaliar o desempenho da abordagem proposta.

Contribuições

A principal contribuição deste trabalho é o desenvolvimento de um processo automatizado para alocação de relatórios de *bugs*. Para isso, são extraídas métricas relacionadas aos *commits* (WAZLAWICK, 2019) realizados no arquivo que contém o erro para estimar o nível de afinidade e a carga de trabalho de cada membro da equipe de desenvolvimento.

Este trabalho também considera o problema de atribuição de relatórios de *bugs* como um problema de otimização combinatorial e usa a meta-heurística *Golden Ball* (GB) (OSABA; DIAZ; ONIEVA, 2013) para efetuar atribuições em lote. Complementarmente, visa não apenas priorizar os desenvolvedores mais capazes de corrigir um *bug*, mas também fornecer uma distribuição uniforme da carga de trabalho entre os membros da equipe de desenvolvimento.

Os resultados experimentais mostram que, quando comparados com um algoritmo de força bruta, a abordagem proposta atinge valores ideais para alocação em 3 de 4 instâncias simuladas. Além disso, a abordagem obteve médias significativamente melhores

em 12 de 13 instâncias quando comparada ao um algoritmo genético e 11 de 13 quando comparada com um algoritmo genético distribuído, sendo que nas outras instâncias não houve uma diferença significativa entre as técnicas.

Estrutura do Trabalho

O restante deste trabalho está organizado da seguinte forma. O Capítulo 1 apresenta uma retrospectiva sobre os algoritmos utilizados nesta pesquisa. No Capítulo 2 são resumidos alguns trabalhos relacionados a atribuição automática ou semiautomática de relatórios de *bugs*. No Capítulo 3 é descrita a proposta de solução para a atribuição de relatórios de *bugs*. O Capítulo 4 traz uma justificativa sobre o problema abordado, a solução proposta e a experimentação. Em seguida, o Capítulo 5 apresenta os experimentos em um conjunto de dados proprietários e demonstra os resultados e a discussão sobre as descobertas. Por fim, o Capítulo 6 conclui o trabalho, identificando algumas ameaças a validade e trabalhos futuros.

1 Referencial Teórico

Neste capítulo, é realizada uma retrospectiva sobre os algoritmos utilizados nesta abordagem: Sistema de Inferência *Fuzzy* de Mamdani, Algoritmo Força Bruta, Algoritmos Genéticos e *Golden Ball*.

1.1 Sistemas de Inferência *Fuzzy*

Sistemas *fuzzy* possuem a capacidade de lidar com processos complexos, cujas informações são imprecisas, qualitativas e aproximadas (ZADEH, 1999; MCNEILL; THRO, 2014). Ao lidar com valores linguísticos, um sistema de inferência *fuzzy* facilita o mapeamento da experiência de especialistas do domínio.

Sistemas de inferência *fuzzy* de Mamdani (1977) são baseados em um conjunto de regras do tipo *if-then*, que fazem o tratamento das informações imprecisas descritas em linguagem natural, para o formato numérico facilmente manipulado por computador. Normalmente, essas regras são extraídas do conhecimento de especialistas no tema abordado, e representam as ações que o sistema deve tomar em relação às variações de valores que as variáveis de entrada e saída podem assumir.

A estrutura básica de um sistema de inferência *fuzzy* é composta de três processos: Fuzzificação, Inferência e Defuzzificação. Esses processos são representados pelo diagrama da Figura 1, no qual o mecanismo é alimentado com valores numéricos precisos resultantes de medições ou observações. A seguir, é realizada uma breve descrição dos processos.

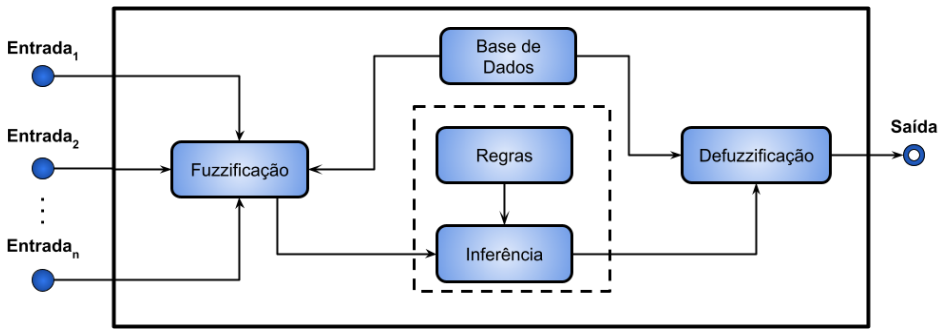


Figura 1 – Sistema de inferência *fuzzy*.

- **Fuzzificação:** converte os valores numéricos precisos de entrada, que resultam de medições ou observações, em valores linguísticos como "Alto", "Médio", "Baixo". Esses valores servem de entrada para o processo de inferência;

- **Inferência:** baseado nos valores de entrada e na base de regras, definida previamente, é produzido um valor linguístico de saída;
- **Defuzzificação:** tansforma o valor de saída do processo de inferência em um valor numérico preciso, para que possa ser utilizado por um especialista ou um sistema.

1.2 Algoritmo Força Bruta

O algoritmo de força bruta (ou busca exaustiva) é uma técnica de solução de problemas de uso geral, que consiste em enumerar todas as possíveis soluções e verificar cada uma para saber qual a solução satisfaz o problema. Esse tipo de algoritmo é simples de implementar e sempre encontrará uma solução se esta existir (LEARY, 2020).

A principal desvantagem do uso do método por força bruta em problemas reais, é que o número de instâncias tendem a crescer exponencialmente, demandando um custo computacional muito elevado. Deste modo, a força bruta é tipicamente utilizada em problemas cujo tamanho é relativamente pequeno ou quando um algoritmo mais eficiente não é conhecido.

1.3 Algoritmos Genéticos

O algoritmo genético (GA) (HOLLAND, 1992; JONG; SPEARS, 1990; GOLDBERG; HOLLAND, 1988) é uma meta-heurística baseada no processo genético dos organismos vivos e na seleção e adaptação das espécies. Esses algoritmos trabalham com uma população única de indivíduos. Cada indivíduo representa uma possível solução para um problema e são criados de forma aleatória respeitando algumas restrições de inicialização.

Essa técnica inicia-se criando e avaliando uma população inicial. Depois, a cada interação os indivíduos mais aptos são selecionados para o processo de cruzamento, no qual será gerado os novos indivíduos. A seguir, realiza-se o processo de mutação, no qual alguns indivíduos selecionados aleatoriamente passam por pequenas modificações. Finalmente, os melhores indivíduos são selecionados em um processo chamado seleção de sobrevivência.

Apesar da ampla aplicação de GA's em trabalhos científicos, eles apresentam algumas desvantagens, sendo a mais conhecida sua rápida convergência para um ótimo local, o que dificulta achar o ótimo global exato e requer um grande número de avaliações da função de aptidão. Assim, para mitigar as limitações do GA foram propostos os algoritmos genéticos paralelos (PGA). Os PGA's são de simples implementação e em geral proporcionam melhoras significativas nos resultados. Essas meta-heurísticas podem ser classificadas em: granulação fina (MANDERICK; SPIESSENS, 1989), mestre-escravo unipopulação, (REEVES, 1997) e baseados em ilhas ou distribuídos (WHITLEY; RANA;

HECKENDORN, 1999; CANTÚ-PAZ et al., 1998). Desses o distribuído é o mais comumente utilizado.

Os algoritmos genéticos paralelos distribuídos (DGA) são técnicas que trabalham com multiplas populações de indivíduos, chamadas subpopulação. Cada subpopulação evolui separadamente na maioria do tempo, e ocasionalmente trocam indivíduos entre si.

1.4 *Golden Ball Optimization*

O GB é uma meta-heurística evolutiva multipopulacional baseada em conceitos de futebol, adequado para resolver problemas de otimização combinatorial (OSABA; DIAZ; ONIEVA, 2013). Esse método, assim como outras técnicas baseadas em populações, inicia sua execução criando uma população de possíveis soluções, chamadas jogadores. Em seguida, divide essa população em subpopulações, chamadas de times. Cada time possui seu próprio treinador e treina de forma independente. O treinamento visa aperfeiçoar a qualidade dos jogadores. Finalmente, os times se enfrentam em competições da liga. A competição é fundamental para definir as transferências e o treinamento para cada time. A Figura 2 mostra uma visão geral das etapas que compõem o algoritmo GB. A seguir uma breve descrição de cada etapa.

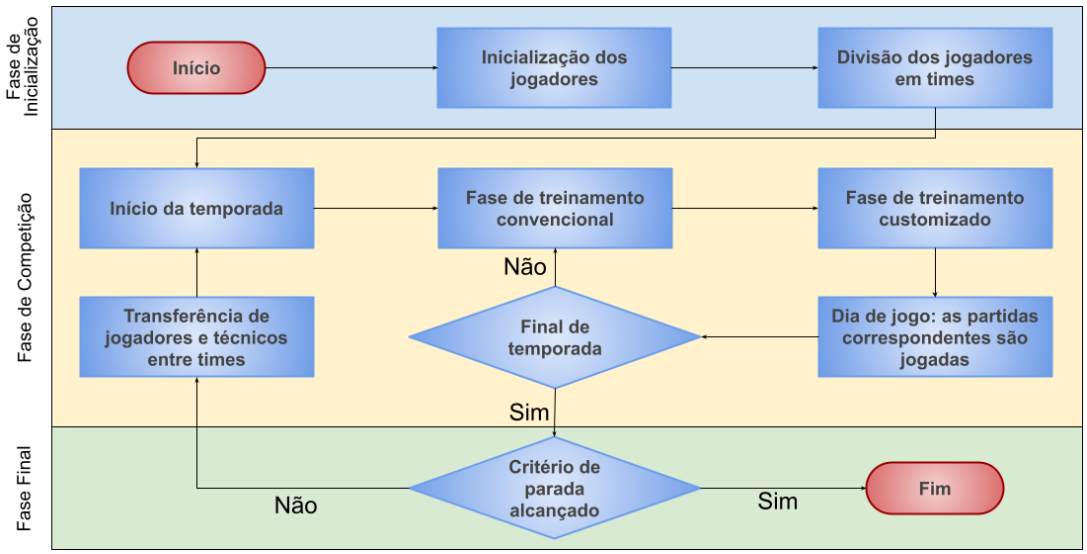


Figura 2 – Fluxograma do algoritmo *Golden Ball*.

- **Fase de inicialização**: nesta fase são definidos a quantidade de times (NT), quantidade de jogadores por time (PT), bem como o treinamento que cada time receberá.
- **Fase de Treinamento**: existem dois tipos de treinamento: o convencional no qual os jogadores de cada time treinam diariamente em busca de melhorar sua qualidade e o personalizado aplicado aos jogadores que não conseguem mais melhorar apenas com treinamento convencional.

- **Fase de Competição**: aqui todos os times devem se enfrentar duas vezes por temporada. O time vencedor ganha 3 pontos e o perdedor 0. Em caso de empate cada time ganha 1 ponto. A pontuação final obtida por cada time é utilizada para realizar uma classificação, que ordena os jogadores de forma decrescente de qualidade.

- **Fase de Transferência**: nessa fase os times procuram se fortalecer trocando seus jogadores. Existem três tipos de transferências. 1) Transferência por temporada: no meio e no final de cada temporada, os times são classificados de acordo com sua pontuação e posição na tabela. Os times que se encontram na metade superior são reforçados com os melhores jogadores dos times da metade inferior, enquanto os times da metade inferior recebem os piores jogadores dos melhores times. 2) Trocas especiais: quando um jogador não consegue mais melhorar mesmo recebendo treinamentos personalizados, ele troca de time de forma aleatória a fim de obter novos treinamentos. Essa troca pode acontecer a qualquer momento da temporada. 3) Transferência de Treinador: a cada período de transferência os times da metade inferior da tabela podem trocar aleatoriamente de treinamento, na esperança de obter outro método que melhore seu desempenho.

2 Trabalhos Relacionados

Vários trabalhos modelam a distribuição automática ou semiautomática de um relatório de *bug* como um problema de classificação (MURPHY; CUBRANIC, 2004; MANI; SANKARAN; ARALIKATTE, 2019; JEONG; KIM; ZIMMERMANN, 2009; SINGH, 2014; YADAV; SINGH; SURI, 2019). Com base no histórico de relatórios de *bugs* corrigidos, esses trabalhos utilizam classificadores tradicionais, redes neurais ou sistemas *fuzzy*, para sugerir o desenvolvedor que pode corrigir o relatório recém recebido, com sucesso. Este capítulo apresenta uma revisão destes trabalhos.

Singh (2014) propõe uma abordagem que modela um perfil para cada desenvolvedor com base no histórico de *bugs* resolvidos. Posteriormente, mapeia esse perfil para uma Matriz de Mapeamento de Domínio. Em seguida, prevê a lista de desenvolvedores mais adequada que poderia resolver os *bugs* relatados recentemente. Mani, Sankaran e Aralikatte (2019) propuseram a técnica baseada em Rede Neural Recorrente Bidirecional com Atenção para triagem automática de *bugs*. A rede aprende a sintaxe e o significado de representações textuais de relatórios de *bug* de forma não supervisionada e, em seguida, recomenda os desenvolvedores. Yadav, Singh e Suri (2019) verificaram em seu estudo que a experiência do desenvolvedor tem um papel importante para reduzir as reatribuições e diminuir o tempo de correção de um *bug*. Então, sugeriram uma abordagem que calcula uma pontuação de experiência do desenvolvedor a partir de *bugs* corrigidos anteriormente, efetua a classificação dos desenvolvedores com base em suas pontuações e finalmente atribui o *bug* àquele que obtiver a melhor pontuação. Entretanto, as três abordagens citadas neste parágrafo consideram apenas o histórico de *bugs*, em consequência, pode haver sobrecarrega de alguns desenvolvedores devido não considerarem a carga de trabalho dos mesmos e atribuição de relatórios a desenvolvedores inativos por não considerarem a recência de suas contribuições, retardando assim o processo de correção de *bugs*.

Tamrawi et al. (2011) propuseram o Bugzie, que recomenda desenvolvedores capazes de corrigir um *bug*. Bugzie utiliza conjuntos *fuzzy* para modelar a correlação/associação de correção de *bugs* entre desenvolvedores com atividades de conserto mais recentes, e termos técnicos com base nas atividades de correção dos desenvolvedores. Mohan, Sardana et al. (2016) propuseram o modelo de triagem de *bug* Visheshagya baseado em tempo, que considera vários meta-campos do relatório de *bug* para atribuição a um desenvolvedor. O Visheshagya monta perfis de atividades para os desenvolvedores e utiliza o tempo, em que um termo foi usado pela última vez por cada desenvolvedor, para calcular sua experiência mais recente. A análise experimental demonstrou que a inclusão do fator tempo junto com a frequência de uso dos termos melhora os resultados da triagem de *bugs*. Ambos os trabalhos consideram apenas a experiência e a recência, e isso pode causar concentração

de tarefas em alguns membros do time de desenvolvimento, uma vez que não levam em consideração a carga de trabalho dos desenvolvedores.

Xi et al. (2019) propõem a abordagem de triagem de *bugs* iTriage que modela simultaneamente o conteúdo textual, os recursos dos metadados e a sequência de realocação dos relatórios de *bugs*. Segundo os autores, os três aspectos são importantes para a precisão da triagem de *bugs*. Em seguida, usa Rede Neural Recorrente Bidirecional com neurônio GRU para prever os desenvolvedores para corrigir o *bug*. Porém, Khatun e Sakib (2018) em seu estudo, verificaram que considerar apenas atividades de correção de *bugs* ou *commits* dos desenvolvedores pode levar a sugestão de desenvolvedores inativos ou inexperientes. Então, propuseram uma técnica de atribuição de *bug*, que considera a experiência e a recência tanto da correção de *bugs* quanto dos *commits* dos desenvolvedores. Por não considerarem a carga de trabalho dos desenvolvedores, as abordagens propostas nesses trabalhos podem levar a uma concentração de tarefas em um pequeno número de desenvolvedores experientes.

Guo et al. (2020) propuseram um método de Rede Neural Convolucional baseado em atividades do desenvolvedor que recomenda uma lista de desenvolvedores para correção de um *bug*. A proposta utiliza o método Word2Vec para representar o vetor de palavras das informações textuais do relatório de *bug* e, finalmente, considera o nível de atividade dos desenvolvedores para reordenar a lista de recomendação. Segundo o autor, uma ameaça à validade da abordagem proposta, é a necessidade de treinamento da rede sempre que um novo desenvolvedor é integrado ao time.

Zaidi e Lee (2021) propuseram um método baseado em *one-class support vector machine* para a realização da triagem de *bugs*, o qual modela os dados de *bug* de cada desenvolvedor de forma a incluir desenvolvedores recém-adicionados. Como este método utiliza um classificador separado para cada desenvolvedor, relatórios são atribuídos tanto a desenvolvedores experientes quanto a desenvolvedores novatos. Porém, o método possui algumas restrições como: um conjunto de dados balanceados e pelo menos 100 relatórios por desenvolvedor. Como muitos desenvolvedores possuem um número pequeno de relatórios a classificação feita pelo método pode ser prejudicada.

Zhang (2020) sugeriu uma solução de aprendizado de máquina, que utiliza recursos baseados em *Latent Dirichlet Allocation* e *Deep Neural Network*, para classificar e atribuir relatórios de *bugs* ao componente associado ao *bug*, e não ao desenvolvedor diretamente. Atribuir os relatórios aos componentes que apresentam o erro pode evitar uma carga de trabalho desequilibrada, pois é possível envolver desenvolvedores experientes e novos.

Rahman et al. (2012) propuseram uma abordagem baseada em Algoritmo Genético para atribuição de *bugs* a desenvolvedores. Utilizaram uma otimização de objetivo único para minimizar a distância entre as competências (combinação de conhecimentos, habilidades e experiência na área em consideração) solicitadas pelos *bugs* e disponíveis do desenvolvedor atribuído. As competências relevantes são determinadas antecipadamente

pelo gerente de projeto, e consideradas estáticas para o conjunto de *bugs*. A abordagem, apesar de utilizar apenas desenvolvedores disponíveis, não considera a carga de trabalho dos desenvolvedores, isso pode levar a uma sobrecarga de relatórios em um pequeno número de desenvolvedores.

Yin, Dong e Xu (2018) propuseram um método híbrido baseado em uma seleção diversificada de recursos e uma máquina de aprendizado extremo. Primeiro, é extraído resumo, descrição e comentários dos desenvolvedores dos relatórios de *bugs*. Então, um modelo de espaço vetorial é montado com base nestas informações. Por fim, um conjunto menor de recursos representativos não redundantes com o máximo de informação estatística é selecionado. Esses recursos são usados para treinar um classificador de conjuntos ELM baseado em GA. Porém, o método não considera a carga de trabalho dos desenvolvedores, desse modo, pode haver sobrecarrega de alguns desenvolvedores. Também, não considera a recência das atividades dos desenvolvedores, isso pode levar a atribuição de relatórios a desenvolvedores que não estão mais contribuindo com o desenvolvimento.

Khatun (2017) introduziu a abordagem de alocação de equipe para garantir a atribuição de *bugs* a desenvolvedores existentes e novos, que agrupa os desenvolvedores por tipo de *bug* e adiciona cada novo desenvolvedor ao grupo de tipo de *bug* escolhido. Com a chegada de um novo *bug* uma equipe de desenvolvedores é sugerida de acordo com a sua pontuação, sua carga de trabalho atual e a gravidade do relatório. No entanto, o trabalho utiliza apenas relatórios que já foram tratados anteriormente.

Kashiwa e Ohira (2020) formularam a triagem de *bugs* como um problema de mochila múltipla. Para isso, propuseram o método *Release-Aware Bug Triaging* que coloca um limite máximo no número de relatórios que são atribuídos a cada desenvolvedor durante um determinado período. Além disso, considera a carga de trabalho de cada desenvolvedor. No entanto, delimitar o número de relatórios atribuídos a cada desenvolvedor permite que alguns relatórios fiquem sem atribuição. Apesar da abordagem evitar que os relatórios se concentrem em um pequeno número de desenvolvedores experientes, ela não impede a atribuição a desenvolvedores inativos.

Este trabalho difere dos demais, por utilizar a afinidade e carga de trabalho dos desenvolvedores para otimizar a distribuição dos relatórios de *bug* e o balanceamento da carga de trabalho do time. Adicionalmente, a fim de evitar que relatórios de *bugs* sejam atribuídos a desenvolvedores inativos, antes da atribuição, é efetuada uma verificação da recência de suas atividades e se estes continuam contribuindo com o código. Outra diferença, é a utilização do *stack trace* gerado por uma exceção não tratada e dados obtidos da análise dos logs de *commit* (WAZLAWICK, 2019), ao invés de dados textuais dos relatórios. Por fim, a abordagem modela a distribuição de relatórios de *bug* como um problema de otimização combinatorial, ao contrário dos trabalhos relacionados, que formulam como um problema de classificação.

A Tabela 1 resume os artigos selecionados neste trabalho considerando quatro aspectos: informações do *bug*, técnica, objetivo e conjuntos de dados usados em cada um dos trabalhos. Com esta tabela, é possível observar que a maioria dos trabalhos utiliza apenas dados textuais extraídos do relatório de *bug* e projetos *open source*.

Tabela 1 – Revisão da literatura de estudos recentes relacionados à triagem de *bug*.

Autor	Informação do *Bug*	Técnica	Objetivo	*Dataset*
Singh (2014)	Identificador do relatório, tipo, componente, resumo, sistema operacional	*Domain Mapping Matrix*	Utilizar técnicas de mineração de frequência de termo para recomendar a lista de desenvolvedores mais adequados para corrigir o novo *bug*	Projetos Google Chromium
Mani, Sankaran e Aralikatte (2019)	Resumo e descrição	Rede Neural Recorrente Bidirecional com Atenção	Melhorar o modelo de representação de relatórios de *bugs* e o desempenho dos classificadores existentes	Projetos Google Chromium, Mozilla Core e Mozilla Firefox
Yadav, Singh e Suri (2019)	Prioridade, produto, tempo relatado	Pontuação de experiência do desenvolvedor combinando experiência, contribuição e desempenho	Reduzir o tempo geral de correção de *bugs* diminuindo a quantidade de reatribuições	Mozilla, Eclipse, Netbeans, Firefox, e Freedesktop
Tamrawi et al. (2011)	Resumo e descrição	Experiência do desenvolvedor por meio de conjuntos *fuzzy*	Melhorar a eficiência e precisão na triagem de *bugs*	Projetos Firefox, Eclipse, Apache, Net Beans, FreeDescktop, GCC e Jazz
Mohan, Sardana et al. (2016)	Componente, prioridade, gravidade e sistema operacional	Ponderação com reconhecimento de tempo	Encontrar o desenvolvedor mais competente para um relatório de *bug* recém-chegado	Projetos Mozilla e Eclipse
Xi et al. (2019)	Resumo, descrição, produto e componente	TF-IDF e Rede Neural Recorrente Bidirecional com Neurônio GRU	Melhorar a precisão da triagem de *bug* considerando os metadados e as sequências de reatribuições, além das informações dos recursos textuais dos relatórios de *bugs*	Projetos Mozilla e Eclipse
Khatun e Sakib (2018)	Relatório de *bugs*: resumo, descrição e tempo relatado. Logs do *commit*: autor e data.	TF-IDF	Atribuir relatórios de *bugs* a desenvolvedores ativos	Projetos Eclipse JDT e SWT
Guo et al. (2020)	Resumo e descrição	Rede Neural Convolucional baseado em atividades do desenvolvedor	Melhorar o desempenho da triagem automática de *bug*	Projetos Eclipse, Mozilla e NetBeans

Autor	Informação do *Bug*	Técnica	Objetivo	Dataset
Zaidi e Lee (2021)	Resumo e descrição	*One-Class Support Vector Machine*	Atribuir relatórios de *bug* a desenvolvedores experientes e novos	Projetos Firefox e Eclipse (Platform)
Zhang (2020)	Descrição, componente e desenvolvedor atribuído	*Latent Dirichlet Allocation* e Rede Neural Profunda	Tornar a atribuição de *bugs* mais eficiente	Conjunto de Dados Proprietário e Projeto Firefox
Khatun (2017)	Relatório de *bug*: resumo, descrição e gravidade. Código-fonte e Logs de *commit*	*Latent Dirichlet Allocation*	Atribuir relatórios de *bugs* a desenvolvedores experientes e novos mantendo a carga de trabalho equilibrada	Projeto Eclipse JTD
Kashiwa e Ohira (2020)	Resumo e descrição	*Latent Dirichlet Allocation* e *Support Vector Machine*	Mitigar o problema de concentração de tarefas e aumentar a quantidade de *bugs* corrigidos em um determinado período	Mozilla Firefox, Plataforma Eclipse, Projeto GNU GCC
Rahman et al. (2012)	Relatório de *bugs*: resumo, descrição e tempo relatado. Código fonte	Algoritmo Genético	Encontrar uma atribuição viável, que minimize o tempo total necessário para corrigir todos os *bugs* fornecidos	Eclipse JDT
Yin, Dong e Xu (2018)	Relatório de *bug*: resumo, descrição e comentários dos desenvolvedores	TF-IDF, Máquina de Aprendizado Extremo e Algoritmo Genético	Corrigir de forma rápida e eficiente os *bugs*	Bugzilla, Eclipse, NetBeans, GCC
Abordagem Proposta	Relatório de *bug*: comentários dos desenvolvedores e código de erro. Código-fonte. Logs de *commit*: autor, data, linhas afetadas	Sistema de Inferência *Fuzzy* de Mamdani e *Golden Ball Optimization*	Realizar a atribuição de relatório de *bug* considerando o grau de afinidade e a carga de trabalho de cada desenvolvedor	Base de Dados Privada

3 Abordagem Proposta

Este capítulo apresenta a abordagem proposta para distribuição de relatórios de *bug*. A abordagem proposta é dividida em três etapas: pré-processamento, cálculo das métricas e modelo meta-heurístico de otimização. A etapa de pré-processamento (Seção 3.1) recebe como entrada os relatórios recém cadastrados, a exceção/ tipo de erro e o arquivo que apresenta erro referente a cada relatório. Essas estradas são utilizadas para estimar o esforço para corrigir o *bug*, calcular a frequência, recência e conhecimento do domínio de cada desenvolvedor. O cálculo das métricas (Seção 3.2) realiza a estimativa da carga de trabalho do desenvolvedor e da afinidade do desenvolvedor com o arquivo que está apresentando erro. Por fim, no modelo meta-heurístico (Seção 3.3), é realizada a atribuição dos relatórios aos desenvolvedores mais aptos para corrigi-los considerando a afinidade do desenvolvedor e sua carga de trabalho. A Figura 3 mostra uma visão geral das etapas que compõem o processo de atribuição de relatórios de *bugs*.

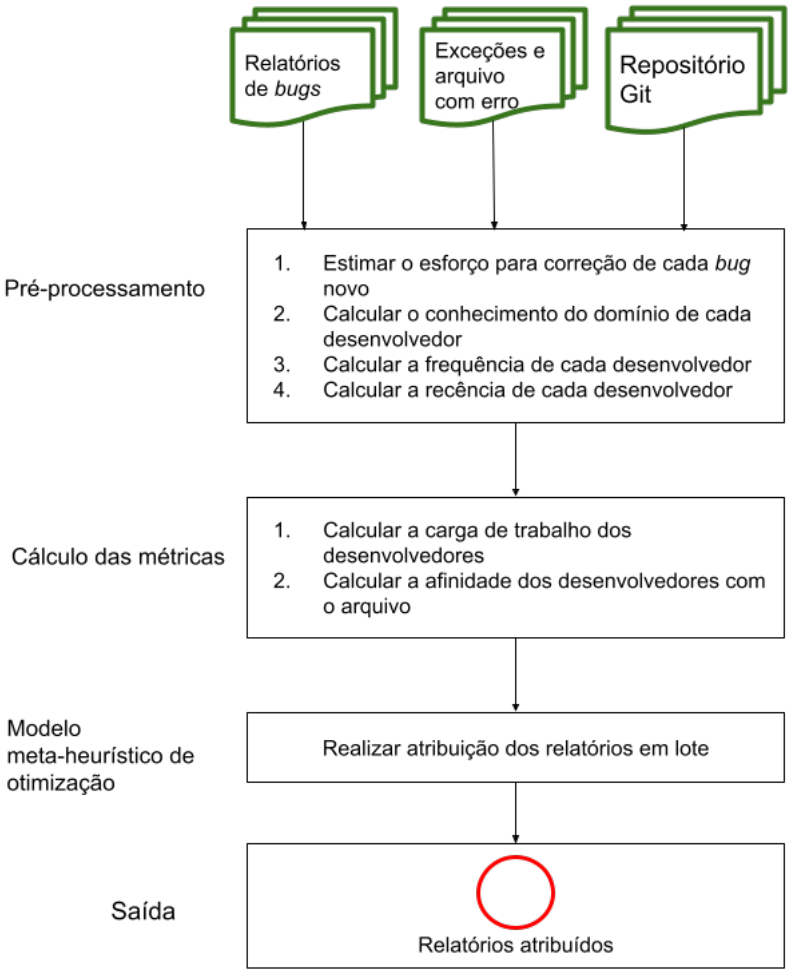


Figura 3 – Fluxo de execução e saída do processo.

3.1 Pré-Processamento

A etapa de pré-processamento recebe como entrada os relatórios de *bugs* recém cadastrados, a exceção/tipo de erro e o arquivo que apresenta erro referente a cada relatório. Esta etapa estima o esforço para corrigir cada relatório e calcula as métricas: conhecimento do domínio, frequência e recência (MOHAN; SARDANA et al., 2016; TAMRAWI et al., 2011; ZHANG; GONG; VERSTEEG, 2013; AVELINO et al., 2018), que são utilizadas no cálculo da carga de trabalho e afinidade de cada desenvolvedor em relação ao arquivo que apresenta erro. Essas variáveis são detalhadas a seguir:

- **Esforço**: O esforço para corrigir um *bug* é estimado a partir dos *bugs* semelhantes resolvidos anteriormente. Este trabalho considera como *bugs* semelhantes aqueles que possuem o mesmo tipo de erro (exceção). Para cada *bug* semelhante encontrado são identificados os arquivos que foram alterados em sua correção. Para cada arquivo é obtido o número de linhas de código afetadas (modificadas, inseridas e/ou removidas) para corrigir o *bug*. Por fim, o esforço para corrigir um *bug* que surgiu é estimado, somando-se todas as linhas de código afetadas na correção dos *bugs* e dividindo pelo número de *bugs* semelhantes (THUNG, 2016; BOEHM et al., 2000);

- **Conhecimento do Domínio**: representado pela quantidade de interações, ou seja, anotações ou comentários que cada desenvolvedor realizou em relatórios de *bugs* anteriores associados ao componente que apresenta o erro. O conhecimento do domínio permite mensurar a experiência do desenvolvedor em relação ao componente que está apresentando mau funcionamento. Um alto conhecimento do componente resulta em uma alta afinidade do desenvolvedor;

- **Frequência**: obtida por meio da quantidade de *commits* que cada desenvolvedor ativo, ou seja, que continua contribuindo com o componente, realizou no arquivo que apresenta o erro. A frequência é importante pois permite ilustrar a contribuição/conhecimento do desenvolvedor no arquivo que contém o erro. Quanto mais *commits* um desenvolvedor realizou em um arquivo, maior a sua afinidade com esse arquivo;

- **Recência**: corresponde à diferença de dias entre a data atual e o último *commit* realizado por cada desenvolvedor no arquivo que contém o erro. A recência permite identificar e priorizar desenvolvedores ativos. Um valor baixo da recência indica uma maior afinidade e atividade recente do desenvolvedor.

3.2 Cálculo das Métricas

Esta etapa recebe como entrada os valores do esforço, do conhecimento do domínio, da frequência e da recência obtidos na etapa de pré-processamento. Esses valores são utilizados para calcular a carga de trabalho do desenvolvedor e a afinidade do desenvolvedor em relação ao arquivo que está apresentando o erro. Aqui, a carga de trabalho de cada desenvolvedor é obtida somando-se o esforço previsto para correção de cada relatório aberto atribuído a ele. Já, para extrair um único valor que representa a afinidade do desenvolvedor, foi construído um sistema de inferência *fuzzy* de Mamdani (1977).

Como pode ser observado na Figura 4, as variáveis de entrada são Frequência, Recência e Conhecimento do Domínio. Essas variáveis foram utilizadas como entrada do sistema fuzzy, por serem consideradas relevantes no cálculo da afinidade e neste trabalho não possuírem sobreposição entre si. A afinidade é a variável de saída do sistema de inferência *fuzzy* e, consequentemente, uma alta afinidade implica em um maior conhecimento (*i.e. expertise*) do desenvolvedor em relação ao arquivo que contém o erro e maior a capacidade deste em corrigir o problema.

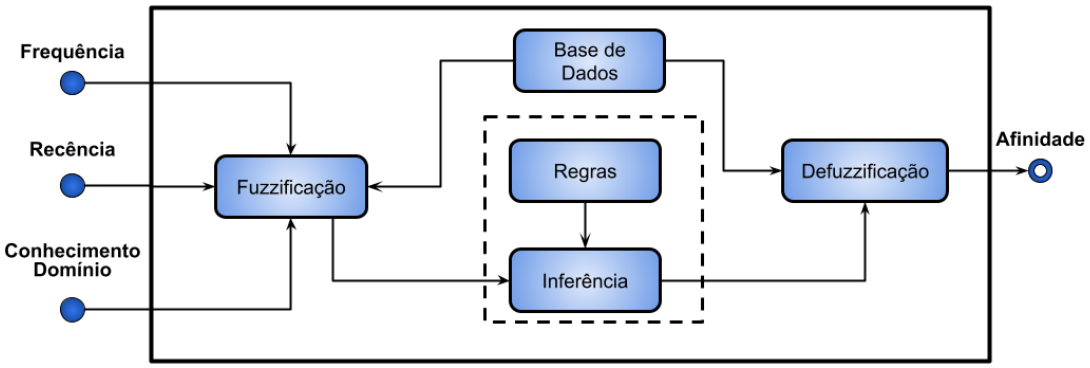
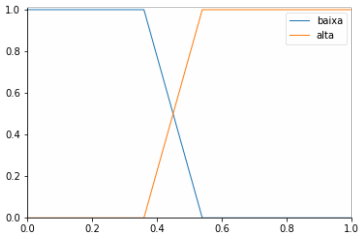


Figura 4 – Sistema de inferência *fuzzy* para estimativa da afinidade dos desenvolvedores em relação ao arquivo que apresenta o erro.

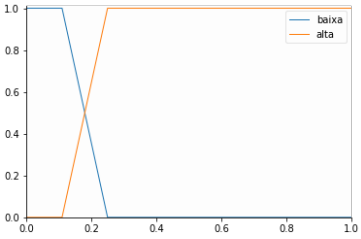
Foram usadas funções de pertinência trapezoidal para mapear empiricamente conjuntos *fuzzy* para as variáveis de entrada e de saída. Para cada variável de entrada foram mapeados dois conjuntos *fuzzy*: Baixo e Alto. Já para a variável de saída mapeou-se três conjuntos *fuzzy*: Baixo, Médio e Alto. O universo de cada conjunto foi definido a partir do conhecimento de especialistas.

A Figura 5 ilustra os conjuntos *fuzzy* associados às variáveis linguísticas. Onde, as subfiguras (a) (b) e (c) representam os conjuntos *fuzzy* associados às variáveis de entrada frequência, recência e conhecimento do domínio, respectivamente. E a subfigura (d) ilustra os conjuntos associados a variável de saída afinidade. Como os valores das variáveis possuem intervalos muito diferentes, optou-se em normalizá-los para uma escala comum entre [0,1], porém sem distorcer as diferenças nos intervalos de valores e para isso

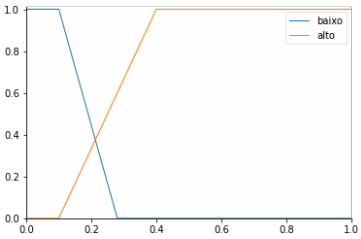
foi utilizada a regra min-max.



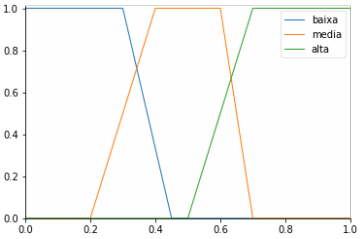
(a) Frequência



(b) Recência



(c) Conhecimento do Domínio



(d) Afinidade

Figura 5 – Distribuição dos Conjuntos *Fuzzy*.

Após a definição dos conjuntos de entrada e saída, é necessário determinar a base de regras para o sistema de inferência *fuzzy* proposto. A base de regras é representada em forma de matriz, como pode ser visto na Tabela 2.

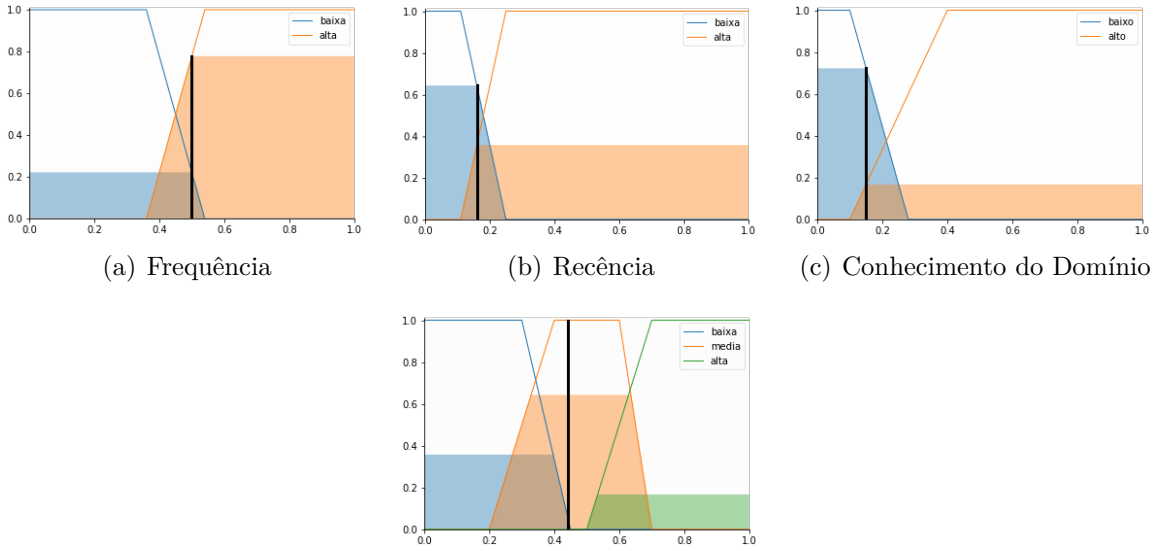
Tabela 2 – Base de regras do sistema de inferência *fuzzy* para estimar a afinidade do desenvolvedor em relação ao arquivo que contém erro. As linhas em cinza claro representam os valores linguísticos das variáveis Frequência/Recência e Conhecimento do Domínio. Enquanto, as linhas em branco representam os valores linguísticos correspondentes à variável de saída, Afinidade.

Afinidade (a)	**Frequência (f) / Recência (r)**			
Conhecimento do Domínio (d)	Alto / Baixo	Alto / Alto	Baixo / Baixo	Baixo / Alto
Baixo	Médio	Baixo	Baixo	Baixo
Alto	Alto	Médio	Médio	Baixo

A Figura 6 ilustra um passo a passo do funcionamento do sistema *fuzzy* para estimar a afinidade a partir dos valores de frequência igual a 0,5 (a), de recência 0,16 (b) e de conhecimento do domínio 0,15 (c). Como pode ser observado na figura, esses valores ativam as seguintes regras:

Se (f é alta) e (r é baixa) e (d é baixo) então (a é média)
Se (f é baixa) e (r é alta) e (d é alto) então (a é baixa)

Considerando os conjuntos *fuzzy* para a variável afinidade e os respectivos graus de pertinencias produzidos pela aplicação das regras ativadas, obtem-se o conjunto da Figura 6 (d). Após o processo de defuzzificação o resultado numérico da afinidade com a aplicação da técnica do centróide (ROSS, 2005) é igual a 0,4406.



(a) Frequência (b) Recência (c) Conhecimento do Domínio

(d) Afinidade

Figura 6 – Funcionamento do sistema *fuzzy* para estimar a afinidade.

3.3 Modelo Meta-Heurístico de Otimização

A abordagem atribui os relatórios de *bug* recém cadastrados aos desenvolvedores ativos. Cada atribuição é baseada em duas métricas que representam a afinidade do desenvolvedor em relação ao arquivo que contém o erro do novo *bug* e a carga de trabalho de cada desenvolvedor, ambas obtidas na Seção 3.2 (cálculo das métricas). Essas métricas são importantes, pois além de garantirem que desenvolvedores com maior grau de afinidade têm prioridade na correção do *bug*, também visam proporcionar uma distribuição equilibrada da carga de trabalho.

Para efetivar a atribuição dos novos relatórios foi utilizada a meta-heurística *Golden Ball* (GB) (OSABA et al., 2014; OSABA; DIAZ; ONIEVA, 2013). O GB é uma meta-heurística evolutiva multipopulacional (OSABA; DIAZ; ONIEVA, 2014; MA et al., 2019) que utiliza diferentes conceitos relacionados ao futebol para resolver problemas de otimização combinatorial (OSABA; DIAZ; ONIEVA, 2013). A função objetivo para o problema abordado consiste na maximização da afinidade contemplando uma distribuição uniforme da carga de trabalho. Dessa forma, pretende-se potencializar a obtenção de melhores combinações entre relatórios e desenvolvedores em sua totalidade (em lote) e não individualmente.

Como mencionado anteriormente, este trabalho de pesquisa formula a distribuição de relatórios de *bugs* como um problema de otimização combinatorial. A atribuição de relatórios consiste em um conjunto de relatórios $R = \{R_1, R_2, ..., R_n\}$ e um conjunto de desenvolvedores $D = \{D_1, D_2, ..., D_m\}$ e a solução pode ser representada por uma sequência de alocação de desenvolvedores para cada relatório de *bug*.

A seguir, é realizada uma demosntração da aplicação prática do GB para o problema abordado neste trabalho. Supondo-se uma instância de 5 relatórios $R = \{R_1, R_2, R_3, R_4, R_5\}$ e 3 desenvolvedores $D = \{D_1, D_2, D_3\}$, uma possível solução poderia ser representada por S = $\{D_1, D_3, D_2, D_1, D_1\}$, que representa a atribuição ao desenvolvedor D_1 dos relatórios R_1, R_4 e R_5; ao desenvolvedor D_2 do relatório R_3; e ao desenvolvedor D_3 do relatório R_2.

O primeiro passo da implementação do GB consiste em definir os jogadores e os times. Os jogadores representam uma possível solução para o problema. Inicialmente os jogadores são criados de forma aleatória respeitando algumas restrições de inicialização. Em seguida, esses jogadores são divididos aleatoriamente em diferentes times que juntos formam uma liga. Cada time possui seu próprio treinador representado por uma função de treinamento, também, atribuída aleatoriamente. O treinamento visa aperfeiçoar a qualidade dos jogadores.

Como pode ser observado na Figura 7, para essa instância do problema a liga possui dois times. Cada time é formado por três jogadores. No caso do time t_2 pode-se observar o jogador j_{21} que é formado pela sequência de desenvolvedores D_1, D_1, D_3, D_2, D_2. Essa sequência significa que esse jogador corresponde a atribuição dos relatórios R_1 e R_2 ao desenvolvedor D_1, do relatório R_3 ao desenvolvedor D_3 e dos relatórios R_4 e R_5 ao desenvolvedor D_2.

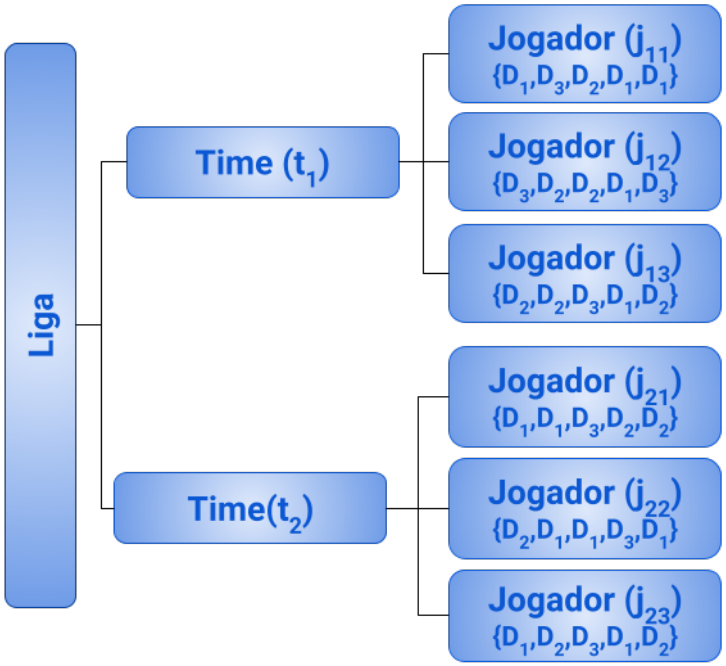


Figura 7 – Possível distribuição dos jogadores dentro dos times.

A Figura 8 (a) ilustra uma matriz na qual as linhas representam os relatórios de *bug* e as colunas caracterizam os desenvolvedores. A intersecção das linhas e colunas representa as afinidades de cada desenvolvedor em relação ao arquivo que contém o erro

do relatório. As linhas da matriz apresentada na Figura 8 (b) refletem os relatórios e a coluna os valores estimados do esforço para corrigi-lo. Por fim, a Figura 8 (c) apresenta os valores das cargas de trabalho atuais de cada desenvolvedor.

	D_1	D_2	D_3
R_1	0,210381	0,443296	0,194295
R_2	0,42793	0,192157	0,194295
R_3	0,316374	0,194295	0,276234
R_4	0,795833	0,355167	0,285222
R_5	0,643284	0,471429	0,355167

(a)

	E
R_1	1
R_2	1
R_3	0,245
R_4	0,097
R_5	0,5358

(b)

	D_1	D_2	D_3
CT	0,5	0,1	0,3

(c)

Figura 8 – Afinidade do desenvolvedor com cada relatório de *bugs*.

O próximo passo é calcular a força dos times (QT_i). A métrica QT_i é avaliada de acordo com a qualidade dos jogadores do time, ou seja, quanto melhores são os jogadores, mais forte é o time, isto é, a força do time equivale a qualidade da solução na resolução do problema de otimização em questão. A força do time (QT_i) é definida pelo somatório das qualidades de cada desenvolvedor dividida pela quantidade de jogadores por time (PT). A qualidade (q_{ij}) de cada jogador (j_{ij}) é definida pela função objetivo $f(j_{ij})$, dada pela seguinte fórmula:

$$f(j_{11}) = g(A_{11}, A_{32}, A_{23}, A_{14}, A_{15})/[1 + h(CT_1, CT_2, CT_3)] \tag{3.1}$$

em que, $f(j_{11})$ representa a qualidade do jogador j_1 do time t_1, $g(A_{11}, A_{32}, A_{23}, A_{14}, A_{15})$ corresponde a soma das afinidades dos desenvolvedores em relação o novo relatório, e é representada por:

$$g(A_{11}, A_{32}, A_{23}, A_{14}, A_{15}) = A[R_1][D_1] + A[R_3][D_2] + A[R_2][D_3] + A[R_4][D_1] + A[R_5][D_1] \tag{3.2}$$

Já a função $h(CT_1, CT_2, CT_3)$ representa o desvio padrão da carga de trabalho contemplando as tarefas previamente alocadas com as tarefas propostas. Após a obtenção da qualidade é possível identificar o capitão do time (j_{icap}). Essa identificação é realizada a partir do jogador com a melhor qualidade (q_{ij}) dentre os diversos jogadores de seu time. A

Figura 9 apresenta um exemplo da qualidade de cada jogador. Como pode ser observado o capitão do time t_1 é o jogador j_{11} e do time t_2 é o jogador j_{23} uma vez que possuem a melhor qualidade do seu time. Já as forças dos times são $QT_1 = 1{,}021821$ e $QT_2 = 0{,}963071$.

Figura 9 – Cálculo da qualidade dos jogadores.

Os jogadores estão em constante treinamento a fim de melhorar sua qualidade. Quando um jogador não consegue melhorar a qualidade apenas com o treinamento convencional, ele participa de um treinamento customizado com o capitão do seu time. Após os treinamentos, iniciam-se as partidas. Como pode ser visto na Figura 10, primeiro é efetuada a ordenação dos jogadores por sua qualidade. Em seguida, a qualidade de cada jogador é comparada com o jogador correspondente do outro time e o que possuir melhor qualidade marca um gol. Ao final da partida o time que possuir mais gols, é o vencedor e recebe 3 pontos. Se a partida terminar em empate cada time recebe 1 ponto.

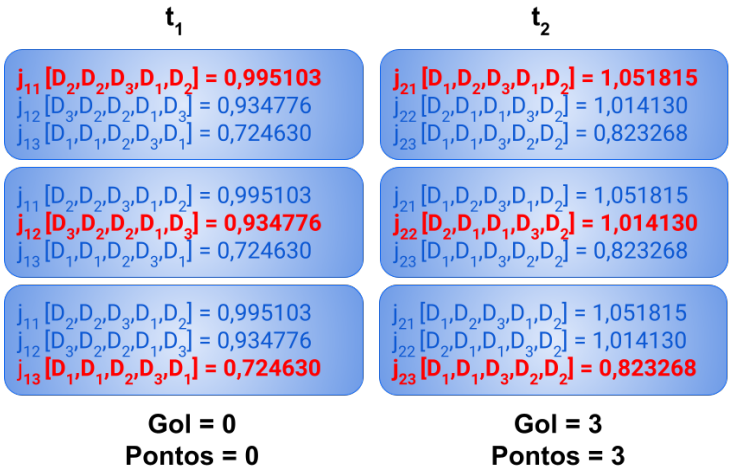


Figura 10 – Exemplo de uma partida.

Ao terminar a partida é efetuada a classificação dos times e iniciam-se as transfe-

rências dos jogadores. Como pode ser visto na Figura 11, a transferência dos jogadores se dá trocando o pior jogador de t_1 com o melhor jogador de t_2.

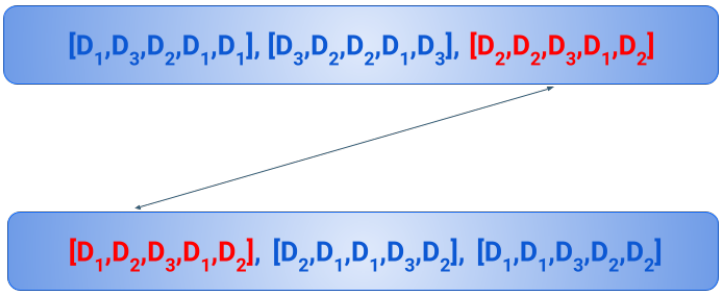


Figura 11 – Transferência dos jogadores.

O algoritmo se repete até que três critérios de parada sejam atingidos: 1) a força de todos os times não melhora em relação à temporada anterior; 2) a qualidade de todos os capitães não melhora em relação à temporada anterior e 3) não há melhora na melhor solução encontrada por todo o sistema em relação à temporada anterior. Após atingir todos esses critérios o sistema pára, e retorna a melhor solução encontrada.

4 Justificativa

Este capítulo apresenta a justificativa das decisões tomadas na construção da abordagem proposta.

4.1 Problema Abordado

A informatização de processos de trabalho, nos últimos anos, tornou-se uma atividade comum em ambientes empresariais cada vez mais competitivos onde o valor de negócio está associado à aquisição e processamento de informações (PRESSMAN, 2005). Essa informatização se concretiza no desenvolvimento de sistemas de acordo com as regras impostas pelo negócio, por meio dos processos de engenharia de *software*. Uma das atividades do processo de Engenharia de *Software* é a manutenção de código, que abrange as atividades: manutenção evolutiva, responsável por acrescentar novas funcionalidades ao sistema, manutenção adaptativa, que visa adaptar o software a novas tecnologias ou novo ambiente externo e manutenção corretiva, que serve para corrigir erros (*bugs*) durante a execução do sistema (PRESSMAN, 2005; LIENTZ; SWANSON, 1980).

Uma etapa importante da manutenção corretiva é a triagem de *bugs*, pois envolve categorizar, verificar a validade, definir um nível de gravidade e atribuir um relatório de *bug* a um desenvolvedor. A atribuição de um relatório é uma tarefa indispensável na triagem de um *bug*, pois é o processo de designar o desenvolvedor mais apto para corrigir um determinado *bug* (AKILA; ZAYARAZ; GOVINDASAMY, 2015; THUNG, 2016). Porém, essa tarefa pode ser complexa, demorada e sujeita a erros quando realizada manualmente. Apesar de vários trabalhos sobre o tema, ainda é um problema bastante estudado, uma vez que não existe uma solução que funcione em todos os cenários.

4.2 Uso do *Stack Trace* na Identificação de *Bugs* semelhantes

A maioria dos estudos sobre triagem de *bugs* utilizam os recursos textuais dos relatórios, principalmente, títulos/resumo e descrição, para identificação de *bugs* semelhantes (YIN; DONG; XU, 2018; LEE et al., 2017; TIAN et al., 2016; XI et al., 2018; CHOQUETTE-CHOO et al., 2019; ALAZZAM et al., 2020). No entanto, os recursos textuais dos relatórios podem conter ruídos, informações incompletas, trechos de código, e detalhes do *stack trace*, etc. (MANI; SANKARAN; ARALIKATTE, 2019).

Com o objetivo de recuperar informações mais precisas e com menos ruídos, para identificação de *bugs* semelhantes ocorrigidos anteriormente, essa abordagem optou pelo

uso do *stack trace* gerado no momento em que um *bug* ocorre. O *stack trace* contém as informações de exceção lançadas e uma sequência de chamadas de funções coletadas em um dado momento da execução do sistema (GU et al., 2019; XU et al., 2020; DUAN et al., 2019; ZHAO et al., 2021). O arquivo que se encontra no topo do *stack trace*, possui o método que estava sendo executado no momento em que a falha ocorreu. Desse modo, essa abordagem utiliza o arquivo versionado pelo repositório de código, do projeto analisado, mais próximo do topo para calcular as métricas da Seção 3.1.

4.3 Estimativa do Esforço com Base em Linhas de Código

Vários estudos utilizam o tempo médio de correção de *bugs* anteriores para estimar o esforço de correção de um novo *bug* (KHATUN, 2017; VIJAYAKUMAR; BHUVANESWARI, 2014; WEISS et al., 2007). Apesar de ser sensato estimar o esforço em termos de período decorrido entre a criação de um relatório de *bug* e sua correção, isso pode não refletir o esforço real necessário para corrigi-lo, visto que é possível que os desenvolvedores não comecem a trabalhar no *bug* imediatamente após recebê-lo, ou estejam corrigindo outro defeito, ou até mesmo não possam efetuar a atualização do relatório no momento em que de fato concluíram a correção (THUNG, 2016).

Devido a essas incertezas, ainda que as linhas de código afetadas nem sempre reflitam o esforço real para correção do *bug*, pois em alguns casos problemas mais complexos podem demandar mais pesquisa e testes que não comporão o código fonte, alguns trabalhos utilizam o número de linhas de código afetadas (modificadas, inseridas e/ou removidas) na correção de *bugs* semelhantes para estimar o esforço de correção do novo bug. Segundo Boehm et al. (2000), o número de linhas de código usadas para consertar um *bug* mede mais diretamente o esforço do desenvolvedor para produzir os códigos necessários.

Dese modo, como a abordagem utiliza o *stack trace* para auxiliar na extração de métricas do código fonte, optou-se por utilizar as linhas de código afetadas para mensurar o esforço de correção do novo *bug*.

4.4 Sistemas de Inferência Fuzzy

Sistemas de inferência *fuzzy* são utilizados para estimar a afinidade dos desenvolvedores em relação ao arquivo que apresenta erros. Pois, apesar de haver estudos sobre qual a melhor técnica existente utilizar para o cálculo da expertise/conhecimento (AVELINO et al., 2018), visualizou-se a aplicação de um sistema inteligente para trabalhar com conceitos linguísticos associados às métricas frequência, recência e conhecimento do domínio, tornando o processo de inferência da afinidade mais transparente.

4.5 Meta-heurística Golden Ball

Já o *golden ball optimization* foi utilizado por ser uma meta-heurística multi-populacional de otimização de natureza combinatorial, na qual cada população possui características e funções diferentes, permitindo que os indivíduos explorem o espaço de busca de forma diferenciada, evitando que caiam facilmente em ótimos locais e aumentando a capacidade de rastreamento da meta-heurística. Em outras palavras, as operações cooperativas auxiliam na capacidade de exploração da técnica e, principalmente, contribuem para melhorar a capacidade de explorar regiões promissoras do espaço de solução (OSABA; DIAZ; ONIEVA, 2013).

4.6 Conclusão

Diante do exposto neste capítulo, espera-se que a abordagem proposta possa identificar o desenvolvedor mais apto para corrigir um novo *bug*. Para esse fim, leva-se em consideração a afinidade em relação ao arquivo que contém a falha e a carga de trabalho de cada integrante do time de desenvolvimento, além de sua atuação em outros relatórios de erro em aberto.

Nesse cenário, pretende-se utilizar métricas extraídas do histórico de correções de erros e dados obtidos da análise dos logs de *commit* (como quantidade de *commits*, data do último *commit* e de linhas afetadas), para mensurar, respectivamente, a afinidade dos desenvolvedores com o arquivo que contém o erro atual e o esforço de correção da falha, bem como de todos os *bugs* que estão sendo corrigidos pelos desenvolvedores.

Uma vez otimizados a distribuição dos relatórios de *bug* e o balanceamento da carga de trabalho do time, almeja-se o aumento, na perspectiva dos usuários, da confiabilidade nos sistemas por meio de respostas mais ágeis aos incidentes. Além disso, pretende-se, também, fornecer ao negócio soluções mais eficazes às falhas, mantendo o equilíbrio das atividades e saúde do time.

5 Experimentação

Neste capítulo são mostrados detalhes da experimentação realizada. Primeiro, o conjunto de dados utilizado é apresentado na Seção 5.1. Em seguida, na Seção 5.2, é descrita a experimentação. Finalmente, na Seção 5.3 os resultados da experimentação são analisados.

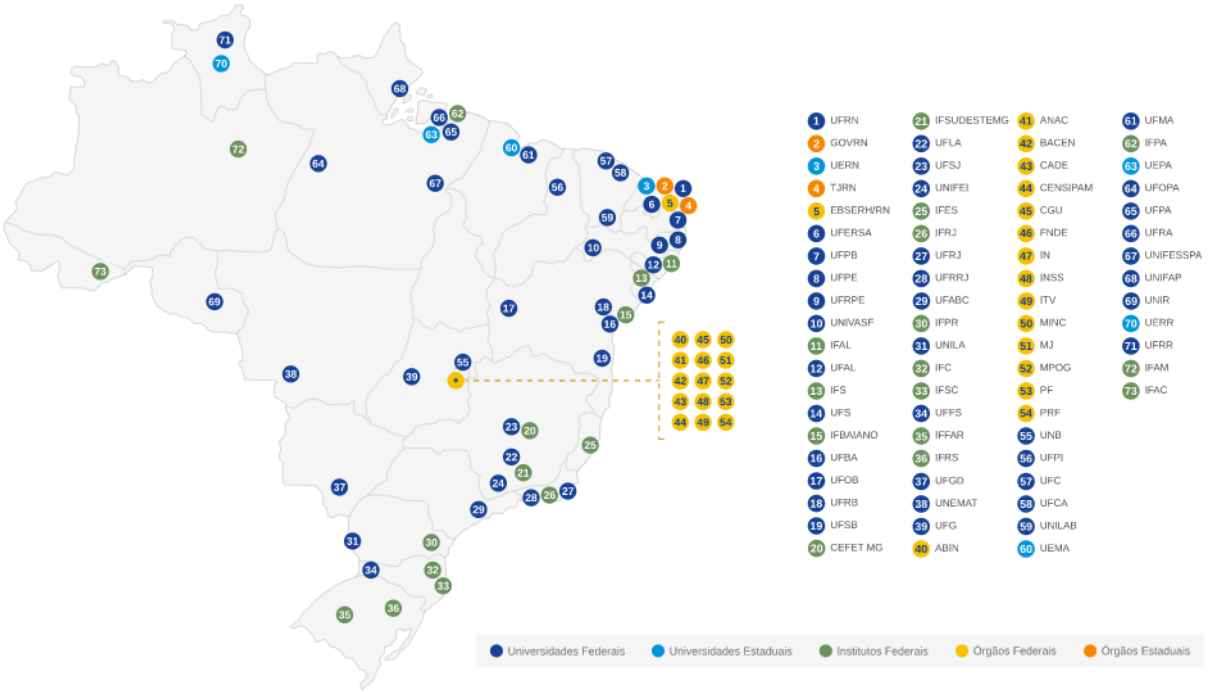
5.1 Descrição do Conjunto de Dados

A abordagem proposta está sendo aplicada em uma base de dados privada dos Sistemas Integrados de Gestão (SIG) da Universidade Federal do Piauí (UFPI). O sistema foi inicialmente desenvolvido pela Universidade Federal do Rio Grande do Norte (UFRN). Hoje, a versão utilizada pela UFPI é mantida pela Superintendência de Tecnologia da Informação/UFPI.

Como pode ser visto na Figura 12 além das instituições de ensino superior que estão no mesmo âmbito da instituição da aplicação deste trabalho, vários outros órgãos públicos brasileiros também utilizam os sistemas SIG desenvolvido pela UFRN. Esses sistemas possuem uma arquitetura de referência que utiliza um banco de dados de *log* para armazenar o *stack trace* gerado no momento em que uma exceção não tratada é ativada. É importante frisar, que os sistemas SIG são compostos por sistemas acadêmicos, recursos humanos e gestão administrativa.

No contexto da UFPI utiliza-se ainda o *Redmine (2021)* como sistema gerenciador de relatórios de *bugs*. É possível extrair do *Redmine* diversas informações relativas ao erro, como: descrições textuais, identificador do *bug*, componente, data de criação, comentários, status, anexos, e caso o erro tenha surgido a partir de uma exceção não tratada, também, é possível encontrar o identificador do *stack trace* salvo no banco de dados de *log*. Vale ressaltar que, quando uma falha não tratada ocorre, é apresentada uma tela informando do comportamento inesperado e direciona o desenvolvedor, testador, ou usuário final para o *Redmine* para que seja registrado um relatório sobre *bug*. No momento em que o relatório é registrado o identificador do *stack trace* gerado, é associado ao mesmo e os dados são armazenados em uma base de dados. Com base no *stack trace* associado ao relatório, é possível recuperar o tipo de erro ou exceção e, assim, estimar o esforço necessário para corrigir um *bug*. Adicionalmente, a partir do *stack trace* é possível obter o arquivo que contém o erro, o qual permite extrair métricas para aferir a afinidade do desenvolvedor.

Para esse trabalho são considerados apenas os *bugs* gerados por exceções não tratadas, pois apenas estes possuem um *stack trace*. De forma complementar, exceções que



Fonte: UFRN - Universidade Federal do Rio Grande do Norte (UFRN/STI, 2021)

Figura 12 – Instituições Parceiras.

já são tratadas não reverberam em problemas e não geram comportamentos inesperados durante a execução do sistema, ou seja, não impedem seu funcionamento.

Efetuou-se uma busca, no *Redmine* da UFPI, por relatórios de *bug* registrados a partir do momento em que a STI assumiu a manutenção e customização dos sistemas. Foram encontrados 2072 relatórios classificados como *bug*. Destes, alguns não possuem um *stack trace* associado e foram descartados por não permitirem a extração dos recursos necessários para inferir o esforço e afinidade. Bem como os *stack trace* cujos arquivos que apresentaram falha são de bibliotecas de terceiros e, portanto, não são versionados no repositório de código do projeto analisado. Seguindo esses critérios de inclusão e exclusão, a coleção final contém 494 relatórios de *bugs*, que poderão ser usados para estimar o esforço e afinidade de correção dos novos *bugs*.

A Figura 13 mostra a distribuição das diferentes exceções contidas nesses relatórios. Cada tipo distinto de erro foi identificado no gráfico por um índice de 1 a 57, compreendendo, assim, a quantidade de exceções distintas contidas nos relatórios de *bug* analisados. Sendo que as exceções que ocorrem com mais frequência são: *NullPointerException* com 34,39% das exceções, *GenericJDBCException* com 12,09%, *Unable to get managed connection for jdbc/SIGAADB* com 11,90% e *NonUniqueResultException* com 10,33% das exceções. Destaca-se, ainda, que das 57 classificações obtidas, 21 delas ocorreram apenas uma vez. É possível concluir, portanto, que o sistema não classifica os erros de forma genérica e nem de forma muito granular, uma vez que dos 494 relatórios analisados obteve-se 57 agrupamentos e a maioria deles associados a mais de um *bug*.

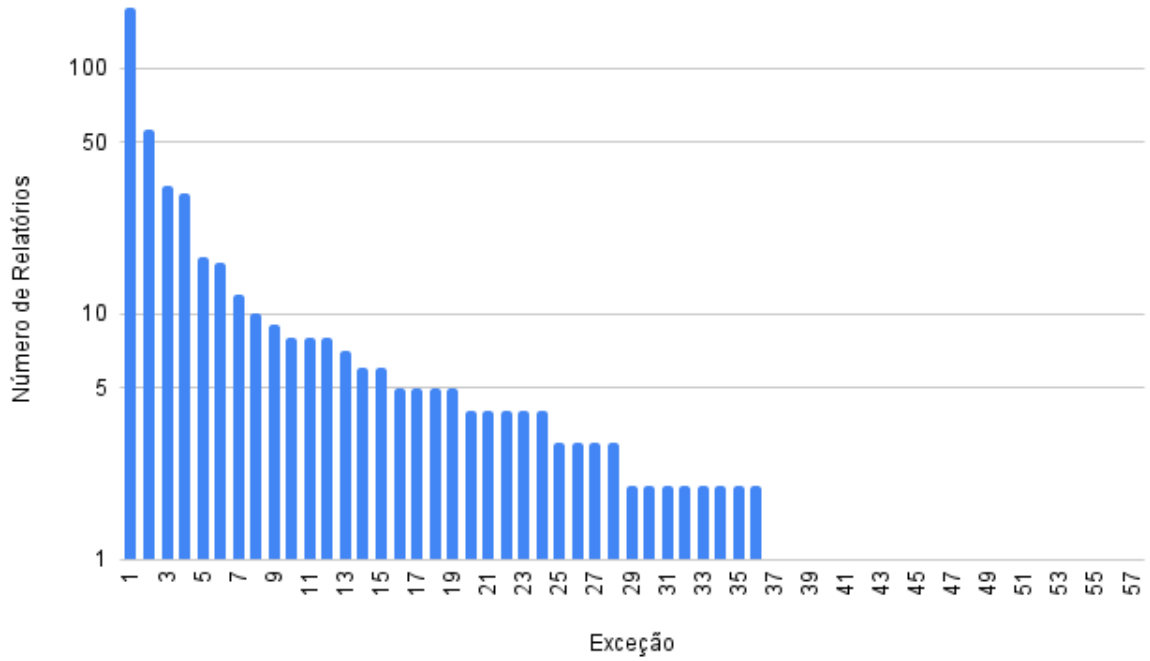


Figura 13 – Distribuição de exceção em relação aos relatórios de *bug*.

5.2 Descrição da Experimentação

Como afirmado na Seção anterior, a análise experimental do problema de atribuição de relatórios de *bugs* é baseada em dados reais da Universidade Federal do Piauí. Como supracitado, neste trabalho foi utilizado o GB para efetivar a atribuição dos relatórios de *bugs*. O GB é um algoritmo de otimização combinatorial, portanto, o objetivo da análise experimental é avaliar o quão próximo, da distribuição de relatórios ideal, estão os resultados. Assim, os resultados obtidos usando GB foram comparados com um algoritmo de força bruta. Para o problema da abordagem proposta, o algoritmo de força bruta compara todas as possibilidades de alocação de desenvolvedores para corrigir cada relatório de forma a maximizar a afinidade contemplando uma distribuição uniforme da carga de trabalho. Durante os testes, o algoritmo guarda a melhor relação entre desenvolvedores e relatórios, e ao final de todas as comparações, o resultado do algoritmo retorna a solução ótima para o problema.

A Tabela 3 mostra os resultados de um algoritmo de força bruta e do GB, utilizados para a distribuição de relatórios em instâncias menores do problema abordado neste trabalho de pesquisa. Para cada instância, o melhor resultado[1] e o tempo médio de execução são mostrados. Quando comparado com um o algoritmo de força bruta, pode-se observar que a melhor solução encontrada pelo GB para as instâncias de 7, 8 e 10 relatórios,

[1] O melhor resultado para o algoritmo de força bruta corresponde ao máximo valor encontrado após o algoritmo comparar todas possibilidades de soluções para o problema.

equivale de maneira exata à solução encontrada por aquele e no pior caso (9 relatórios) chega a 99,94% do seu valor. Também, é possível observar que o algoritmo de força bruta demanda uma escala de tempo bastante ampla, começando com 49,869 segundos para atribuição de 7 relatórios, chegando a atingir 2,40 dias para 10 relatórios. Isso permite concluir que o algoritmo GB implementado obteve resultados estritamente iguais ou muito próximo do algoritmo força bruta, para instâncias menores de atribuição de relatórios, entretanto com tempo de execução muito menor, o que permite que a abordagem proposta possa ser utilizada em um cenário real.

Tabela 3 – Resultados do algoritmo força bruta e *golden ball* para distribuição de relatórios de *bugs*.

Número de Relatórios	**Força Bruta**		**Golden Ball**	
	Melhor Resultado	**Tempo**	**Melhor Resultado**	**Tempo**
7	0,929	49,869s	0,929	0,560s
8	1,219	16,554m	1,219	0,869s
9	1,549	3,801h	1,548	1,292s
10	1,876	2,4d	1,876	1,794s

Após comprovar a aplicabilidade prática para instâncias menores do problema, torna-se importante uma comparação com outras técnicas de modo a comprovar a qualidade das atribuições de relatórios obtidos com a aplicação do GB, para instâncias maiores. Assim, uma segunda análise experimental foi realizada, agora, comparando os resultados obtidos pelo GB com os obtidos com outros métodos evolutivos. Dentre os métodos evolutivos, utilizou-se um algoritmo genético de população única (GA) e um algoritmo genético distribuído (DGA), pois essas meta-heurísticas são adequadas para resolver problemas de otimização combinatorial. Além disso, os DGA's, assim como o GB, contam com dois operadores, um local e outro cooperativo, na evolução de seus indivíduos (OSABA et al., 2014).

Com o objetivo de calibrar as meta-heurísticas, cada algoritmo foi testado com diferentes configurações. Foram utilizadas populações iniciais aleatórias de 50, 100 e 150. Com relação ao GA e ao DGA variou-se a taxa de mutação e *crossover* entre 5% e 90% com incrementos de 5%. Para o DGA e o GB foram testadas diferentes variações de distribuição da população em subpopulações e times, respectivamente. A seguir são descritas as configurações que obtiveram melhores resultados.

- A configuração final do GA utilizou como função de mutação a função sucessora (HALMOS, 2017) 2-opt, como função de *crossover* a *Half Crossover* (HX) e a função de sobrevivência 100% elitista.
- No DGA a população foi dividida em 10 subpopulações de 10 indivíduos. Cada subpopulação usa uma função de mutação distinta, dentre as funções sucessoras *2-opt*,

3-opt, *Swapping function* e *Insertion function*. Com relação às funções *crossover* e de sobrevivência, assim como o GA, foi utilizada a função *Half Crossover* (HX) e 100% elitista.

- Já para o GB a população foi dividida em 10 times de 10 jogadores - lembrando que cada jogador representa um conjunto de desenvolvedores. Cada time recebeu uma função de treinamento convencional dentre as funções sucessoras *2-opt*, *3-opt*, *Swapping function* e *Insertion function*. Para o treinamento personalizado foi utilizada a função *Half Crossover* (HX). Destaca-se ainda que, a cada 5 e 10 treinamentos sem melhoria é efetuado um treinamento personalizado e uma transferência, respectivamente.

O tamanho da população inicial para os três algoritmos é de 100 indivíduos. As funções sucessoras utilizada nas mutações do GA e DGA e treinamentos do GB foram:

- **2-opt**: dois pares de desenvolvedores são selecionados aleatoriamente de um jogador e posteriormente os desenvolvedores dentro de cada par são trocados de posição entre si;
- **3-opt**: três pares de desenvolvedores são selecionados aleatoriamente de um jogador e posteriormente os desenvolvedores dentro de cada par são trocados de posição entre si;
- ***Swapping function***: seleciona e extrai um desenvolvedor aleatório e o insere em outra posição aleatória dentro do mesmo jogador;
- ***Insertion function***: um par de desenvolvedores são selecionados aleatoriamente de um jogador e posteriormente são trocados de posição entre si.

A Tabela 4 resume as características do GA e do DGA utilizados para comparação com o GB. Enquanto a Tabela 5 mostra as características do GB utilizado para efetivar as atribuições de relatórios de *bug*.

Tabela 4 – Resumo das Características do GA E DGA.

Alg.	**População**	**Função de Sobrevivência**	**Função de Cruzamento**	**Função de Mutação**
GA	1 população, 100 indivíduos	100% elitista	*half crossover*	2-opt
DGA	10 subpopulações, 10 indivíduos	100% elitista	*half crossover*	2-opt, 3-opt, swapping, e insertion

Tabela 5 – Resumo das Características do GB.

População	10 times, 10 jogadores
Treinamentos sem Melhoria para Treinamento Personalizado	5
Treinamentos sem Melhoria para Transferência	10
Função de Treinamento Personalizado	*half crossover*
Função Treinamento Convencional	2-opt, 3-opt, swapping, e insertion

5.3 Resultados e Análise Experimental

Nesta Seção, primeiro é efetuada a apresentação e análise dos resultados obtidos pelas técnicas GB, GA e DGA. Em seguida, é analisada a distribuição da carga de trabalho após efetivação das alocações por cada técnica.

A Tabela 6, mostra o desempenho dos algoritmos na distribuição de relatórios. Para cada instância, o melhor resultado conhecido, e para cada algoritmo o seu melhor resultado, o desvio padrão (σ), a média, o tempo médio de execução em segundos e o percentual da melhor solução conhecida (%) são mostrados. Observando os resultados obtidos pelas três técnicas, percebe-se que o GB encontrou as melhores soluções em 100% das instâncias. Em uma instância, apesar de não encontrar o melhor valor conhecido, atingiu, juntamente com o GA, a melhor solução entre as técnicas. A média das execuções, para cada instância de relatórios, encontradas pelo GB é melhor que as demais técnicas em todas as instâncias. Em relação ao tempo médio de execução, o GA foi mais rápido que as demais técnicas, porém, o GB em 92,30% das soluções atingiu melhores resultados que o GA, e as médias das execuções do GB, como já foi dito, são melhores em todas as instâncias. Já o desvio padrão de todas as técnicas analisadas possui valores pequenos. Isso significa que a diferença entre o melhor e o pior resultado é baixa, o que mostra que os resultados são confiáveis.

Para comprovar o desempenho do GB, foi determinada a significância estatística dos resultados utilizando o *Z-test* de distribuição normal. Esse teste compara os resultados obtidos pelo GB com os resultados de cada uma das outras técnicas permitindo verificar se as diferenças entre o GB e as outras técnicas são estatisticamente significativas ou não. Os resultados dos testes podem ser positivos (+), indicando que os resultados da distribuição de relatórios de *bugs* com o GB são significativamente melhores que a técnica com a qual está sendo comparado; negativos (-) significando que as distribuições realizadas com o GB são substancialmente piores; e neutra (*) indicando que a diferença entre os resultados das duas técnicas comparadas, não é significativa.

A Tabela 7 mostra uma comparação direta entre o GB e cada uma das outras técnicas, por meio do *Z-test* de distribuição normal. O intervalo de confiança foi declarado

Tabela 6 – Resultados do GB, GA e DGA para a distribuição de relatórios. Para cada instância são mostrados a melhor solução conhecida, e para cada algoritmo o seu melhor resultado, o desvio padrão, a média, o tempo de execução (em segundos) e o percentual alcançado da melhor solução conhecida.

Número Relatórios	**Melhor Resultado Conhecido**	***Golden Ball***					**Algoritmo Genético**					**Algoritmo Genético Distribuído**				
		Melhor Resultado	σ	**Média**	**Tempo**	%	**Melhor Resultado**	σ	**Média**	**Tempo**	%	**Melhor Resultado**	σ	**Média**	**Tempo**	%
7	0,929	**0,929**	(±0,003)	0,926	0,270	100	0,928	(±0,003)	0,920	0,121	99,89	**0,929**	(±0,003)	0,921	0,286	100
8	1,219	**1,219**	(±0,004)	1,211	0,386	100	1,217	(±0,004)	1,206	0,215	99,84	**1,219**	(±0,004)	1,209	0,551	100
9	1,549	**1,548**	(±0,004)	1,539	0,584	99,94	**1,548**	(±0,015)	1,526	0,315	99,94	1,545	(±0,009)	1,533	0,832	99,74
10	1,876	**1,876**	(±0,018)	1,850	0,828	100	1,859	(±0,031)	1,824	0,511	99,09	1,874	(±0,039)	1,825	1,412	99,98
11	2,189	**2,189**	(±0,054)	2,122	1,390	100	2,178	(±0,054)	2,082	0,715	99,50	2,159	(±0,046)	2,084	1,990	97,63
12	2,518	**2,518**	(±0,074)	2,379	1,805	100	2,492	(±0,084)	2,310	1,054	98,57	**2,518**	(±0,088)	2,367	2,848	100
13	2,776	**2,776**	(±0,111)	2,628	2,800	100	2,695	(±0,077)	2,542	1,357	97,08	2,732	(±0,066)	2,580	3,981	98,41
14	2,979	**2,979**	(±0,080)	2,827	3,235	100	2,906	(±0,073)	2,740	1,960	97,55	2,963	(±0,077)	2,778	5,636	99,46
15	3,265	**3,265**	(±0,096)	3,073	4,467	100	3,142	(±0,080)	2,994	2,383	96,23	3,153	(±0,076)	3,011	7,682	96,57
16	3,490	**3,490**	(±0,092)	3,292	6,057	100	3,390	(±0,069)	3,243	3,345	97,13	3,398	(±0,092)	3,234	8,723	97,36
17	3,801	**3,801**	(±0,117)	3,566	7,517	100	3,713	(±0,108)	3,494	4,169	97,68	3,798	(±0,108)	3,535	13,096	99,92
18	4,117	**4,117**	(±0,122)	3,888	8,145	100	4,057	(±0,109)	3,780	5,017	93,26	3,986	(±0,125)	3,781	14,514	96,82
19	4,350	**4,350**	(±0,112)	4,121	10,630	100	4,280	(±0,124)	3,994	6,938	94,50	**4,350**	(±0,136)	4,039	19,106	100

em 95% ($z_{0,05} = 1,96$). Como pode ser observado, o GB obteve médias significativamente melhores em 92,30% das instâncias quando comparado ao GA e 84,61% quando comparado com o DGA, sendo que em apenas 23,07% dos casos não houve uma diferença significativa entre as técnicas. Pode-se concluir portanto, que o GB se mostrou mais eficaz e eficiente, na produção de soluções estatisticamente significativas para o problema de atribuição de relatórios de *bugs*, quando comparado com as meta-heurísticas GA e DGA implementadas com os parâmetros descritos anteriormente.

Tabela 7 – *Z-test* de Distribuição Normal. '+' indica que os resultados do GB são significativamente melhores. '*' indica que a diferença entre os dois algoritmos não é significativa (para nível de confiança de 95%).

Número de Relatórios	**GB vs GA**	**GB vs DGA**
7	+(5,65)	+(5,52)
8	+(4,20)	+(2,03)
9	+(3,62)	+(2,62)
10	+(3,31)	+(2,64)
11	+(2,36)	+(2,40)
12	+(2,77)	*(0,49)
13	+(3,29)	+(2,81)
14	+(3,60)	+(1,99)
15	+(2,83)	+(2,27)
16	*(1,90)	+(2,00)
17	+(2,01)	*(0,88)
18	+(2,97)	+(2,75)
19	+(3,10)	+(2,08)

Após confirmar a confiabilidade de utilização do GB para a distribuição de relatórios de *bugs*, foram realizados testes para avaliar a distribuição da carga de trabalho entre os membros do time de desenvolvimento. Para isso, foi simulado a atribuição de 18 relatórios de *bugs*, totalizando um esforço estimado de $60,5$. A Tabela 8 mostra a carga de trabalho de cada desenvolvedor antes da simulação, onde apenas $D3, D7, D9, D11$ e $D14$ possuem relatórios em aberto.

Ao concluir a simulação foi efetuado o cálculo da nova carga de trabalho de cada desenvolvedor. A Tabela 9 relaciona os relatórios com os esforços estimados para corrigi-los e os desenvolvedores aos quais foram atribuídos. Pode-se observar que nesta simulação os relatórios $R2, R3, R8, R11$ foram atribuídos ao desenvolvedor $D6$. Como $D6$ não possuía relatórios em aberto (Tabela 8) sua carga de trabalho total é obtida somando-se os esforços estimados para corrigir cada relatório novo atribuído a ele, ou seja, $0,5+1+1+3 = 5,5$. Já ao desenvolvedor $D11$ foi atribuído o relatório $R7$ com esforço estimado em 3, entretanto $D11$ possuía relatórios em aberto totalizando uma carga de trabalho igual a 3 (Tabela 8), então sua carga de trabalho total, após a simulação, é dada por $3 + 3 = 6$.

Tabela 8 – Carga de Trabalho Atual de Cada Desenvolvimento.

Desenvolvedores	**Carga de Trabalho**
D1	0
D2	0
D3	1
D4	0
D5	0
D6	0
D7	6
D8	0
D9	6
D10	0
D11	3
D12	0
D13	0
D14	3

Tabela 9 – Associação entre os Relatórios, os Esforços Estimados para Corrigi-los e os Desenvolvedores aos Quais Foram Atribuídos.

Relatórios de *Bug*	**Esforço**	**Desenvolvedores**
R1	3	D5
R2	0,5	D6
R3	1	D6
R4	9	D12
R5	6	D10
R6	6	D8
R7	3	D11
R8	1	D6
R9	3	D5
R10	3	D2
R11	3	D6
R12	3	D14
R13	3	D3
R14	3	D2
R15	3	D4
R16	4	D13
R17	3	D4
R18	3	D3

A Figura 14 apresenta uma visão da distribuição dessa carga de trabalho entre os integrantes do time de desenvolvimento. Pode-se observar pelo gráfico que a distribuição de relatórios de *bugs* entre os desenvolvedores por meio do GB gera uma carga de trabalho mais balanceada variando entre 4 e 6. As cargas dos desenvolvedores $D3$ e $D12$ extrapolaram este intervalo pois receberam relatórios com estimativa de esforço alto. Já o desenvolvedor $D1$ não recebeu relatório, isso ocorreu devido ser recém-chegado e sua afinidade com os novos relatórios ser muito baixa em relação aos demais desenvolvedores. Porém, para uma melhor compreensão da variabilidade da distribuição da carga de trabalho, foi calculado o desvio padrão. Como pode ser observado na Tabela 10 a atribuição de relatórios com o GB produz um desvio padrão menor, o que significa que a variabilidade é menor, ou seja, a distribuição da carga de trabalho entre os desenvolvedores é mais uniforme que as cargas de trabalho geradas pelo GA e DGA.

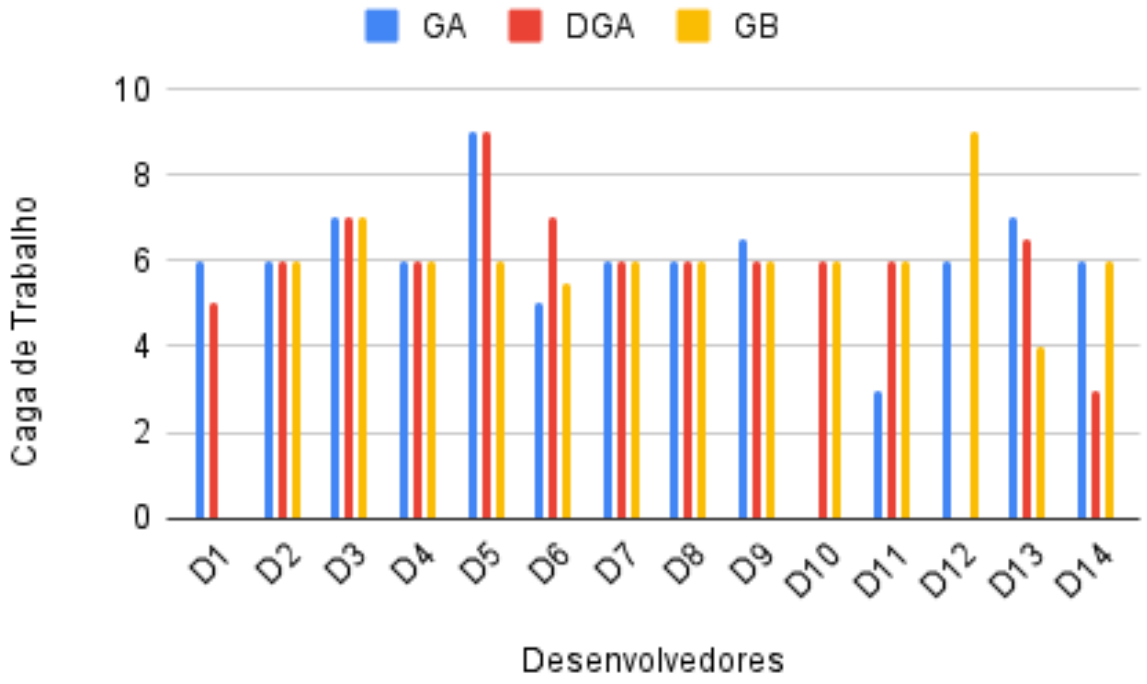


Figura 14 – Distribuição da carga de trabalho das atribuições realizadas pelo GA, DGA e GB.

Tabela 10 – Desvio Padrão da Distribuição da Carga de Trabalho dos Desenvolvedores.

Algoritmos	σ
GA	2,07
DGA	2,07
GB	1,94

Os resultados apresentados nesta seção, evidenciam que o GB se mostrou melhor que as meta-heurísticas GA e DGA para o problema de atribuição de relatórios de *bugs*. Pois, a abordagem proposta conseguiu priorizar o desenvolvedor com maior afinidade para

corrigir cada relatório e ao mesmo tempo manteve uma uniformidade na distribuição da carga de trabalho.

6 Conclusões e Trabalhos Futuros

Em consequência da necessidade de se produzir sistemas cada vez mais complexos, ocorre um aumento substancial da presença de *bugs*, produzindo resultados ou comportamentos imprecisos e inesperados durante a execução do *software*. Desenvolvedores, testadores e usuários reportam estes *bugs* registrando um relatório descrevendo o problema encontrado. Cada relatório é analisado e atribuído a um desenvolvedor para corrigi-lo. Essa análise normalmente é realizada de forma manual, tornando o processo complexo, demorado e sujeito a erros.

Este trabalho propõe uma abordagem baseada em uma meta-heurística evolutiva multipopulacional que utiliza informações relacionadas à carga de trabalho dos desenvolvedores e dados sobre *bugs* previamente corrigidos para sugerir o desenvolvedor mais apto a corrigir um novo *bug*. Ao contrário de outras abordagens, essa proposta pretende efetuar a distribuição de relatórios de *bugs* considerando o desenvolvedor com maior afinidade e ao mesmo tempo procurando manter sua carga de trabalho equilibrada.

A partir dos experimentos pode-se ponderar que é possível a extração de dados do *stack trace* para serem utilizados no cálculo da carga de trabalho e afinidade dos desenvolvedores em relação ao arquivo que apresenta o erro. Outra observação trata da utilização do número de linhas afetadas na produção dos códigos necessários para a correção de *bugs* anteriores, de modo a estimar o esforço do desenvolvedor na correção do novo *bug*. O sistema *fuzzy* utilizado para o cálculo da afinidade apresentou resultados condizentes com valores esperados pelo grupo de pesquisa com relação à manutenção corretiva de *software*.

Por fim, a meta-heurística GB utilizada para efetivar a atribuição de relatórios aos desenvolvedores, em sua totalidade (em lote), mostrou-se promissora para aplicação no problema de triagem de *bugs*. Pois, durante a experimentação, foi possível comprovar a aplicabilidade prática da abordagem para instâncias menores do problema ao comparar os resultados do GB com os de um algoritmo de força bruta. Posteriormente, para instâncias maiores, a abordagem foi comparada com dois algoritmos evolutivos: um GA e um DGA. Os melhores resultados e a média obtidos com a meta-heurística GB são melhores que os dois algoritmos em todas as instâncias. Para uma análise estatística mais aprofundada, foi realizado um *Z-test* de distribuição normal. Os resultados desse teste indicaram que o GB, quando comparado ao GA e ao DGA, se mostrou mais eficaz e eficiente, na produção de soluções estatisticamente significativas para o problema de atribuição de relatórios de *bugs*. Para finalizar, a análise da carga de trabalho demonstrou que a distribuição de relatórios de *bugs* entre os desenvolvedores por meio do GB gerou uma distribuição mais uniforme

que as cargas de trabalho produzidas pelo GA e DGA.

Diante dos resultados experimentais apresentados, evidencia-se que o uso da meta-heurística GB satisfaz os requisitos de priorizar os desenvolvedores mais capacitados, ou seja, com maior afinidade com o arquivo que contém o erro e proporciona uma distribuição uniforme da carga de trabalho dos integrantes do time de desenvolvimento.

6.1 Limitações

Uma limitação da abordagem proposta é a identificação de *bugs* semelhantes exigirem que haja um *stack trace*, gerado no momento em que a falha não tratada ocorreu, associado ao relatório de *bug*. Assim, a necessidade do *stack trace* levou a abordagem a ser validada apenas em um conjunto de dados proprietário.

Adicionalmente, o fato dos desenvolvedores novos possuírem uma afinidade muito baixa ou inexistente, pode causar ociosidade desses.

Outra limitação, é a estimativa do esforço ser nula quando a exceção ou tipo de erro acontecer pela primeira vez. Pois, não existirão bugs semelhantes e isso pode enviesar o balanceamento da carga de trabalho, causando possíveis atrasos na correção dos *bugs*.

6.2 Ameaças a Validade

Algumas possíveis ameaças a abordagem proposta são listadas a seguir.

O uso de linhas afetadas em correções de *bugs* semelhantes para estimar o esforço do novo, pois a utilização das linhas de código não leva em consideração o tempo necessário para análise e identificação do erro. E ainda, a forma que cada desenvolvedor codifica influência em um maior ou menor número de linhas.

Além disso, os testes experimentais foram realizados em um sistema em que o número de relatórios de *bugs* diários são relativamente pequenos. Desse modo, a abordagem pode não generalizar para projetos que possuem uma quantidade significativa de relatórios de *bugs* diários.

6.3 Trabalhos futuros

Como trabalhos futuros, pretende-se utilizar uma técnica de processamento de texto para que seja possível abranger outros tipos de relatórios além dos relatórios de *bugs*, aplicar a abordagem em projetos que possuam uma quantidade significativa de relatórios diários e aperfeiçoar a abordagem proposta para identificar e remover relatórios duplicados.

Planeja-se, também, ajustar o método proposto para ser mais sensível a novos desenvolvedores, além de aprimorar o código para estimar o esforço para corrigir um novo *bug* cuja exceção/tipo de erro surgiu pela primeira vez. Pretende-se ainda, realizar uma validação qualitativa da abordagem proposta aplicando-a e avaliando-a em um ambiente não simulado.

Referências

AKILA, V.; ZAYARAZ, G.; GOVINDASAMY, V. Effective bug triage–a framework. *Procedia Computer Science*, Elsevier, v. 48, p. 114–120, 2015. Citado 2 vezes nas páginas 1 e 25.

ALAZZAM, I. et al. Automatic bug triage in software systems using graph neighborhood relations for feature augmentation. *IEEE Transactions on Computational Social Systems*, IEEE, v. 7, n. 5, p. 1288–1303, 2020. Citado na página 25.

AVELINO, G. et al. Who can maintain this code?: Assessing the effectiveness of repository-mining techniques for identifying software maintainers. *IEEE Software*, IEEE, v. 36, n. 6, p. 34–42, 2018. Citado 2 vezes nas páginas 16 e 26.

BANI-SALAMEH, H.; SALLAM, M. et al. A deep-learning-based bug priority prediction using rnn-lstm neural networks. *e-Informatica Software Engineering Journal*, v. 15, n. 1, 2021. Citado na página 1.

BHATTACHARYA, P.; NEAMTIU, I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: IEEE. *2010 IEEE International Conference on Software Maintenance*. [S.l.], 2010. p. 1–10. Citado na página 1.

BHATTACHARYA, P.; NEAMTIU, I.; SHELTON, C. R. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, Elsevier, v. 85, n. 10, p. 2275–2292, 2012. Citado na página 1.

BOEHM, B. et al. *Software Cost Estimation With Cocomo II*. [S.l.]: Prentice Hall, Upper Saddle River, NJ, 2000. Citado 2 vezes nas páginas 16 e 26.

BUGZILLA. *Bugzilla*. 2021. Último acesso em 02 Agosto 2021. Disponível em: <https://www.bugzilla.org/>. Citado na página 1.

CANTÚ-PAZ, E. et al. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, Citeseer, v. 10, n. 2, p. 141–171, 1998. Citado na página 6.

CHOQUETTE-CHOO, C. A. et al. A multi-label, dual-output deep neural network for automated bug triaging. In: IEEE. *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. [S.l.], 2019. p. 937–944. Citado na página 25.

DUAN, J. et al. An automatic localization tool for null pointer exceptions. *IEEE Access*, IEEE, v. 7, p. 153453–153465, 2019. Citado na página 26.

GOLDBERG, D. E.; HOLLAND, J. H. Genetic algorithms and machine learning. Kluwer Academic Publishers-Plenum Publishers; Kluwer Academic Publishers ..., 1988. Citado na página 6.

GU, Y. et al. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, Elsevier, v. 148, p. 88–104, 2019. Citado 2 vezes nas páginas 3 e 26.

GUO, S. et al. Developer activity motivated bug triaging: via convolutional neural network. *Neural Processing Letters*, Springer, v. 51, n. 3, p. 2589–2606, 2020. Citado 3 vezes nas páginas 1, 10 e 13.

HALMOS, P. R. *Naive set theory*. [S.l.]: Courier Dover Publications, 2017. Citado na página 32.

HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. [S.l.]: MIT press, 1992. Citado na página 6.

JEONG, G.; KIM, S.; ZIMMERMANN, T. Improving bug triage with bug tossing graphs. In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.: s.n.], 2009. p. 111–120. Citado 2 vezes nas páginas 1 e 9.

JIMOH, R. et al. A promethee based evaluation of software defect predictors. *Journal of Computer Science and Its Application*, v. 25, n. 1, p. 106–119, 2018. Citado na página 1.

JIRA. *Jira*. 2021. Último acesso em 02 Agosto 2021. Disponível em: <https://www.atlassian.com/software/jira/>. Citado na página 1.

JONG, K. A. D.; SPEARS, W. M. An analysis of the interacting roles of population size and crossover in genetic algorithms. In: SPRINGER. *International Conference on Parallel Problem Solving from Nature*. [S.l.], 1990. p. 38–47. Citado na página 6.

KASHIWA, Y.; OHIRA, M. A release-aware bug triaging method considering developers' bug-fixing loads. *IEICE TRANSACTIONS on Information and Systems*, The Institute of Electronics, Information and Communication Engineers, v. 103, n. 2, p. 348–362, 2020. Citado 3 vezes nas páginas 1, 11 e 14.

KHATUN, A. A team allocation technique ensuring bug assignment to existing and new developers using their recency and expertise. In: *SOFTENG 2017: The Third International Conference on Advances and Trends in Software Engineering)*. [S.l.: s.n.], 2017. Citado 3 vezes nas páginas 11, 14 e 26.

KHATUN, A.; SAKIB, K. A bug assignment approach combining expertise and recency of both bug fixing and source commits. In: *ENASE*. [S.l.: s.n.], 2018. p. 351–358. Citado 3 vezes nas páginas 2, 10 e 13.

LEARY, M. 6 - topology optimization for am. In: LEARY, M. (Ed.). *Design for Additive Manufacturing*. [S.l.]: Elsevier, 2020, (Additive Manufacturing Materials and Technologies). p. 165–202. ISBN 978-0-12-816721-2. Citado na página 6.

LEE, D.-G.; SEO, Y.-S. Improving bug report triage performance using artificial intelligence based document generation model. *Human-centric Computing and Information Sciences*, Springer, v. 10, n. 1, p. 1–22, 2020. Citado na página 1.

LEE, S.-R. et al. Applying deep learning based automatic bug triager to industrial projects. In: *Proceedings of the 2017 11th Joint Meeting on foundations of software engineering*. [S.l.: s.n.], 2017. p. 926–931. Citado na página 25.

LIENTZ, B. P.; SWANSON, E. B. *Software maintenance management*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1980. Citado na página 25.

MA, H. et al. Multi-population techniques in nature inspired optimization algorithms: A comprehensive survey. *Swarm and evolutionary computation*, Elsevier, v. 44, p. 365–387, 2019. Citado na página 19.

MAMDANI, E. H. Application of fuzzy logic to approximate reasoning using linguistic synthesis. *IEEE transactions on computers*, IEEE Computer Society, v. 26, n. 12, p. 1182–1191, 1977. Citado 3 vezes nas páginas 3, 5 e 17.

MANDERICK, B.; SPIESSENS, P. Fine-grained parallel genetic algorithms. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. p. 428–433. ISBN 1558600663. Citado na página 6.

MANI, S.; SANKARAN, A.; ARALIKATTE, R. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In: *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*. [S.l.: s.n.], 2019. p. 171–179. Citado 5 vezes nas páginas 1, 2, 9, 13 e 25.

MCNEILL, F. M.; THRO, E. *Fuzzy logic: a practical approach*. [S.l.]: Academic Press, 2014. Citado na página 5.

MOHAN, D.; SARDANA, N. et al. Visheshagya: Time based expertise model for bug report assignment. In: IEEE. *2016 Ninth International Conference on Contemporary Computing (IC3)*. [S.l.], 2016. p. 1–6. Citado 4 vezes nas páginas 1, 9, 13 e 16.

MURPHY, G.; CUBRANIC, D. Automatic bug triage using text categorization. In: CITESEER. *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. [S.l.], 2004. p. 1–6. Citado na página 9.

OSABA, E. et al. Focusing on the golden ball metaheuristic: an extended study on a wider set of problems. *The Scientific World Journal*, Hindawi, v. 2014, 2014. Citado 2 vezes nas páginas 19 e 32.

OSABA, E.; DIAZ, F.; ONIEVA, E. A novel meta-heuristic based on soccer concepts to solve routing problems. In: *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. [S.l.: s.n.], 2013. p. 1743–1744. Citado 4 vezes nas páginas 3, 7, 19 e 27.

OSABA, E.; DIAZ, F.; ONIEVA, E. Golden ball: a novel meta-heuristic to solve combinatorial optimization problems based on soccer concepts. *Applied Intelligence*, Springer, v. 41, n. 1, p. 145–166, 2014. Citado na página 19.

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave macmillan, 2005. Citado na página 25.

RAHMAN, M. et al. An empirical investigation of a genetic algorithm for developer's assignment to bugs. In: *Proceedings of the First North American Search based Symposium*. [S.l.: s.n.], 2012. Citado 2 vezes nas páginas 10 e 14.

REDMINE. *Redmine*. 2021. Último acesso em 02 Agosto 2021. Disponível em: <https://www.redmine.org/>. Citado 2 vezes nas páginas 1 e 29.

REEVES, C. Modern heuristics techniques for combinatorial problems. *Nikkan Kogyo Shimbun*, 1997. Citado na página 6.

ROSS, T. J. *Fuzzy logic with engineering applications*. [S.l.]: John Wiley & Sons, 2005. Citado na página 18.

SAHU, K.; LILHORE, U.; AGARWAL, N. Survey of various data reduction methods for effective bug report analysis. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 2018. Citado na página 1.

SINGH, A. S. K. Bug triaging: Profile oriented developer recommendation. *International Journal of Innovative Research in Advanced Engineering*, v. 2, p. 36–42, 2014. Citado 3 vezes nas páginas 1, 9 e 13.

TAMRAWI, A. et al. Fuzzy set and cache-based approach for bug triaging. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.: s.n.], 2011. p. 365–375. Citado 4 vezes nas páginas 2, 9, 13 e 16.

THUNG, F. Automatic prediction of bug fixing effort measured by code churn size. In: *Proceedings of the 5th International Workshop on Software Mining*. [S.l.: s.n.], 2016. p. 18–23. Citado 3 vezes nas páginas 16, 25 e 26.

TIAN, Y.; LO, D.; SUN, C. Drone: Predicting priority of reported bugs by multi-factor analysis. In: IEEE. *2013 IEEE International Conference on Software Maintenance*. [S.l.], 2013. p. 200–209. Citado na página 1.

TIAN, Y. et al. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, Springer, v. 20, n. 5, p. 1354–1383, 2015. Citado na página 1.

TIAN, Y. et al. Learning to rank for bug report assignee recommendation. In: IEEE. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. [S.l.], 2016. p. 1–10. Citado na página 25.

UFRN/STI. *Instituições Parceiras*. 2021. Último acesso em 02 Agosto 2021. Disponível em: <https://portalcooperacao.info.ufrn.br/pagina.php?a=parceiros>. Citado na página 30.

VIJAYAKUMAR, K.; BHUVANESWARI, V. How much effort needed to fix the bug? a data mining approach for effort estimation and analysing of bug report attributes in firefox. In: IEEE. *2014 International Conference on Intelligent Computing Applications*. [S.l.], 2014. p. 335–339. Citado na página 26.

WAZLAWICK, R. *Engenharia de software: conceitos e práticas*. [S.l.]: Elsevier Editora Ltda., 2019. Citado 2 vezes nas páginas 3 e 11.

WEISS, C. et al. How long will it take to fix this bug? In: IEEE. *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. [S.l.], 2007. p. 1–1. Citado na página 26.

WHITLEY, D.; RANA, S.; HECKENDORN, R. B. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, SRCE-Sveučilišni računski centar, v. 7, n. 1, p. 33–47, 1999. Citado na página 6.

XI, S. et al. An effective approach for routing the bug reports to the right fixers. In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. [S.l.: s.n.], 2018. p. 1–10. Citado na página 25.

XI, S.-Q. et al. Bug triaging based on tossing sequence modeling. *Journal of Computer Science and Technology*, Springer, v. 34, n. 5, p. 942–956, 2019. Citado 2 vezes nas páginas 10 e 13.

XU, Z. et al. Imbalanced metric learning for crashing fault residence prediction. *Journal of Systems and Software*, Elsevier, v. 170, p. 110763, 2020. Citado na página 26.

YADAV, A.; SINGH, S. K.; SURI, J. S. Ranking of software developers based on expertise score for bug triaging. *Information and Software Technology*, Elsevier, v. 112, p. 1–17, 2019. Citado 2 vezes nas páginas 9 e 13.

YIN, Y.; DONG, X.; XU, T. Rapid and efficient bug assignment using elm for iot software. *IEEE Access*, IEEE, v. 6, p. 52713–52724, 2018. Citado 4 vezes nas páginas 1, 11, 14 e 25.

ZADEH, L. A. Fuzzy logic= computing with words. In: *Computing with Words in Information/Intelligent Systems 1*. [S.l.]: Springer, 1999. p. 3–23. Citado na página 5.

ZAIDI, S. F. A. et al. Applying convolutional neural networks with different word representation techniques to recommend bug fixers. *IEEE Access*, IEEE, v. 8, p. 213729–213747, 2020. Citado na página 1.

ZAIDI, S. F. A.; LEE, C.-G. One-class classification based bug triage system to assign a newly added developer. In: IEEE. *2021 International Conference on Information Networking (ICOIN)*. [S.l.], 2021. p. 738–741. Citado 2 vezes nas páginas 10 e 14.

ZHANG, H.; GONG, L.; VERSTEEG, S. Predicting bug-fixing time: an empirical study of commercial software projects. In: IEEE. *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.], 2013. p. 1042–1051. Citado 2 vezes nas páginas 1 e 16.

ZHANG, W. Efficient bug triage for industrial environments. In: IEEE. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2020. p. 727–735. Citado 3 vezes nas páginas 2, 10 e 14.

ZHAO, K. et al. A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. *Information and Software Technology*, Elsevier, p. 106652, 2021. Citado na página 26.

ZHU, Z. et al. A deep multimodal model for bug localization. *Data Mining and Knowledge Discovery*, Springer, p. 1–24, 2021. Citado na página 1.