

**UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA**

**UMA ABORDAGEM PARA
SELEÇÃO AUTOMÁTICA DE BUGS EM
PROJETOS DE SOFTWARE
BASEADOS EM COMPONENTES**

GEORGES DE ARAÚJO LIMA

JOÃO PESSOA-PB

Julho de 2018

GEORGES DE ARAÚJO LIMA

**UMA ABORDAGEM PARA
SELEÇÃO AUTOMÁTICA DE BUGS EM
PROJETOS DE SOFTWARE
BASEADOS EM COMPONENTES**

Dissertação submetida à Coordenação do
Curso de Pós-Graduação em Informática da
Universidade Federal da Paraíba como parte
dos requisitos necessários para obtenção do
título de Mestre em Informática.

Orientador: Prof. Dr. Gledson Elias da Silveira

JOÃO PESSOA-PB

Julho de 2018

L732a Lima, Georgenes de Araújo.

Uma abordagem para Seleção Automática de Bugs em
Projetos de Software baseados em Componentes /
Georgenes de Araújo Lima. - João Pessoa, 2018.
103 f. : il.

Orientação: Gledson Elias da Silveira.
Dissertação (Mestrado) - UFPB/Informática.

1. Engenharia de Software Baseada em Buscas. 2.
Sistemas de Rastreamento de Bugs. 3. Seleção de Bugs.
4. Componentes de Software. I. Silveira, Gledson Elias
da. II. Título.

UFPB/BC

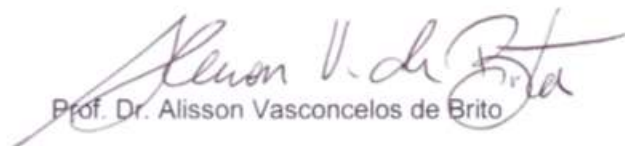


UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA



Ata da Sessão Pública de Defesa de Dissertação de Mestrado de Georgenes de Araújo Lima, candidato ao título de Mestre em Informática na Área de Sistemas de Computação, realizada em 11 de julho de 2018.

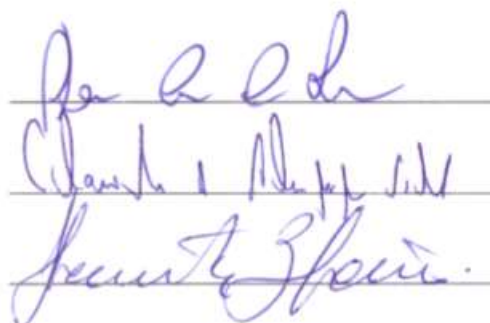
1 Aos onze dias do mês de julho, do ano de dois mil e dezoito, às nove horas, no Centro de
2 Informática da Universidade Federal da Paraíba, em Mangabeira, reuniram-se os membros
3 da Banca Examinadora constituída para julgar o Trabalho Final do Sr. Georgenes de Araújo
4 Lima, vinculado a esta Universidade sob a matrícula nº 20161020158, candidato ao grau de
5 Mestre em Informática, na área de "Sistemas de Computação", na linha de pesquisa de
6 "Sistemas Distribuídos", do Programa de Pós-Graduação em Informática, da Universidade
7 Federal da Paraíba. A comissão examinadora foi composta pelos professores: Gledson Elias
8 da Silveira (PPGI-UFPB), Orientador e Presidente da Banca, Claurton de Albuquerque
9 Siebra (PPGI-UFPB), Examinador Interno ao programa, e Heremita Brasileiro Lima (IFPB),
10 Examinadora Externa à Instituição. Dando início aos trabalhos, o Presidente da Banca,
11 cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a
12 palavra ao candidato para que o mesmo fizesse a exposição oral do trabalho de dissertação
13 intitulado "Uma Abordagem para Seleção Automática de Bugs em Projetos de Software
14 Baseados em Componentes". Concluída a exposição, o candidato foi arguido pela Banca
15 Examinadora que emitiu o seguinte parecer: **"aprovado"**. Do ocorrido, eu, Alisson
16 Vasconcelos de Brito, Vice-Coordenador do Programa de Pós-Graduação em Informática,
17 lavrei a presente ata que vai assinada por mim e pelos membros da banca examinadora.
18 João Pessoa, 11 de julho de 2018.


Prof. Dr. Alisson Vasconcelos de Brito

Prof. Dr. Gledson Elias da Silveira
Orientador (PPGI-UFPB)

Prof. Dr. Claurton de Albuquerque Siebra
Examinador interno ao programa (PPGI-UFPB)

Profa. Dra. Heremita Brasileiro Lira
Examinadora externa à instituição (IFPB)



Dedico este trabalho a Deus, aos meus pais, a minha esposa Clênia Dias,
ao meu filho Nathan Dias e ao Prof. Dr. Gledson Elias.

AGRADECIMENTOS

A DEUS, por me guiar com sua graça permitindo a concretização de mais uma realização.

À minha mãe Maria Cleonice por me ensinar a fazer as coisas com amor e sempre acreditar que eu venceria e meu Pai por me ensinar a ser resistente frente aos obstáculos da vida e ser determinado.

A minha esposa Clênia Dias, por me acompanhar no meu crescimento profissional.

Aos professores que contribuíram com seus conhecimentos para meu aprendizado, em especial ao meu orientador professor Dr. Gledson Elias, que extraiu o melhor de mim em várias áreas do conhecimento, por sua paciência e dedicação e pela sua energia colocada neste trabalho e na orientação, determinantes para que este trabalho fosse concretizado. Ao professor Dr. Clauirton Siebra por ter me incentivado a trabalhar nesta área de seleção de bugs.

Ao meu amigo e professor Dr. Carlos Davidson por me iniciar na carreira de informática.

Ao amigo Daniel Miranda de Brito pela revisão textual e seu grande apoio no fim desta jornada, e aos analistas de sistemas que participaram do experimento manual em seleção de bugs.

Aos membros da banca examinadora, professor Dr. Clauirton Siebra e professora Dra. Heremita Lira, por terem aceitado participar da banca e pelas relevantes contribuições dadas ao trabalho.

Aos meus irmãos Teogenis e Cleogenes, minha tia Gorette e prima Maria da Paz que me incentivaram durante todo o mestrado.

LISTA DE FIGURAS

Figura 1 - Ciclo de vida de um bug	18
Figura 2 - Cadastro de relatório de bug no <i>Redmine</i>	21
Figura 3 - Dependência cíclica	27
Figura 4 - Dependência estrela	27
Figura 5 – Dependência linear.....	28
Figura 6 – Dependência estrela e linear.....	28
Figura 7 - Cone da incerteza.....	32
Figura 8 - Representação de um cromossomo.....	38
Figura 9 – Fluxograma de Execução do Algoritmo Genético	40
Figura 10 - Cruzamento baseado em maioria.....	42
Figura 11 - Cruzamento em um ponto.....	42
Figura 12 - Cruzamento uniforme	43
Figura 13 - Mutação binária simples	44
Figura 14 - Mutação Swap.....	44
Figura 15 - Mutação Creep.....	44
Figura 16 - Validação de Bug.....	59
Figura 17 - Arquitetura em 3 camadas	60
Figura 18 - Dependência entre componentes e bugs relacionados.....	62
Figura 19 - Dependência de bugs baseada em componentes.....	62
Figura 20 - Recomendação das Dependências de Bugs	63
Figura 21 - Matriz Conceitual BA de associação de bugs com componentes.....	65
Figura 22 - Matriz BA de associação de bugs com componentes	65
Figura 23 - Diagrama de Componentes.....	67
Figura 24 - Matriz Conceitual AD de dependência direta de componentes.....	67
Figura 25 - Matriz AD de dependência direta de componentes	67
Figura 26 - Recomendação das Prioridades e Severidades dos Bugs.....	69
Figura 27 - Matriz Conceitual de avaliação dos bugs pelos clientes.....	72
Figura 28 - Recomendação das Tarefas de Correção dos Bugs	74
Figura 29 - Experiência em desenvolvimento ou testes de software.....	83
Figura 30 - Experiência no módulo de graduação da UFPB	84
Figura 31 - Comparação da Busca Genética e Busca Aleatória para 20 bugs.....	85
Figura 32 - Quantidade de votos para cada bug	93

LISTA DE TABELAS

Tabela 1 - Comparação com trabalhos relacionados	58
Tabela 2 - Valores de prioridade e severidade convertidos em número.....	70
Tabela 3 - Organização dos experimentos.....	79
Tabela 4 - Configuração do algoritmo genético	82
Tabela 5 – Resultados da Busca Genética e Busca Aleatória para 20 bugs	86
Tabela 6 - Resultados da Busca Genética e Busca Exaustiva para 20 bugs	87
Tabela 7 - Tempo de execução do cenário de 20 bugs	88
Tabela 8 – Resultados da Busca Genética e Busca Aleatória para 50 bugs	89

SUMÁRIO

RESUMO.....	7
ABSTRACT	7
1 INTRODUÇÃO	9
1.1 OBJETIVOS	15
1.2 ESTRUTURA DO TRABALHO	15
2 FUNDAMENTAÇÃO TEÓRICA.....	17
2.1 SELEÇÃO DE BUGS	17
2.1.1 SISTEMAS DE RASTREAMENTO DE BUGS	20
2.1.2 PRIORIDADE E SEVERIDADE.....	23
2.1.3 DEPENDÊNCIAS DE BUGS	25
2.1.4 CUSTO DO BUG	30
2.2 ENGENHARIA DE SOFTWARE BASEADA EM BUSCA	35
2.2.1 ALGORITMO GENÉTICO	37
2.3 CONSIDERAÇÕES FINAIS	45
3 TRABALHOS RELACIONADOS	47
3.1 PRIORIZAÇÃO DE BUGS NO REPOSITÓRIO KATE EDITOR	48
3.2 PRIORIZAÇÃO E ALOCAÇÃO MULTI-OBJETIVA DE BUGS.....	49
3.3 SELEÇÃO MULTI-OBJETIVA DE BUGS	51
3.4 AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE CORREÇÕES DE BUGS POR HABILIDADE E DISPONIBILIDADE DO DESENVOLVEDOR	53
3.5 AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE CORREÇÃO DE BUGS POR EXPERIÊNCIA DO DESENVOLVEDOR EM ATIVIDADE ANTERIOR	55
3.6 CONSIDERAÇÕES FINAIS	56
4 ABORDAGEM PROPOSTA	59
4.1 RECOMENDAÇÃO DAS DEPENDÊNCIAS DOS BUGS.....	61
4.2 RECOMENDAÇÃO DAS PRIORIDADES E SEVERIDADES DOS BUGS.....	69
4.3 RECOMENDAÇÃO DAS TAREFAS DE CORREÇÃO DOS BUGS	72

4.3.1	GERAÇÃO DE UMA SOLUÇÃO CANDIDATA VÁLIDA	74
4.3.2	AVALIAÇÃO DE UMA SOLUÇÃO CANDIDATA VÁLIDA	75
4.4	CONSIDERAÇÕES FINAIS	77
5	ESTUDO DE CASO.....	78
5.1	IMPLEMENTAÇÃO E CALIBRAÇÃO	79
5.2	RESULTADOS E AVALIAÇÃO COMPARATIVA.....	84
5.2.1	AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE BUGS.....	84
5.2.2	ABORDAGENS AUTOMÁTICAS NO CENÁRIO COM 50 BUGS.....	88
5.2.3	ABORDAGEM BASEADA NO JULGAMENTO DE ESPECIALISTAS	90
5.3	CONSIDERAÇÕES FINAIS	93
6	CONCLUSÃO	95
	REFERÊNCIAS	98

RESUMO

Vive-se uma época em que a tecnologia está presente em quase todos os processos de trabalho e a demanda por recursos tecnológicos é cada vez mais frequente, com mais mudanças nos sistemas de software, mais requisições de manutenção, e, conseqüentemente, uma maior quantidade de bugs para correção. Como a correção de bugs é um dos fatores que impacta na melhoria da qualidade de software, mas representa um alto dispêndio de recursos em um projeto de software, a atividade de correção de bugs representa também um obstáculo crítico para a manutenção e o desenvolvimento de sistemas. Tendo em vista que a quantidade de bugs pode ser maior que os recursos necessários para corrigi-los e atualmente as empresas trabalham com entregas antecipadas chamadas de *releases* que permitem a entrega de partes do software, então, nas tarefas de correções de bugs, os gerentes de projetos podem enfrentar o desafio de selecionar manualmente os bugs a serem corrigidos na próxima versão do produto de software, em geral com base nos critérios de prioridade e severidade indicados nos relatórios de bugs, bem como prazo ou custo. Para tal, os gerentes de projetos adotam ferramentas, denominadas Sistemas de Rastreamento de Bugs, do inglês *Bug Tracking Systems* (BTS), que estabelecem canais de comunicação entre fornecedores e clientes. Para sistemas grandes e complexos, nos quais é comum a existência de um número considerável de relatórios de bugs, uma abordagem de seleção de bugs por julgamento ad-hoc de especialistas não é eficiente e eficaz, representando uma estratégia demorada, intensiva de mão-de-obra e propensa a erros. Neste contexto, explorando técnicas de Engenharia de Software Baseada em Buscas, do inglês *Search Based Software Engineering* (SBSE), este trabalho propõe uma abordagem automatizada para a seleção de bugs interdependentes em projetos de software baseados em componentes, adotando um modelo multicliente para selecionar os bugs mais relevantes do ponto de vista tanto do produtor de software como de seus clientes, mas com custo total limitado pelo orçamento disponível no projeto do software. Para avaliação da proposta, foram realizados dois estudos de casos, onde a estratégia de busca baseada em algoritmo genético é comparada com as estratégias baseadas em algoritmo exaustivo e algoritmo aleatório, e, de forma mais realista, também comparada com a seleção baseada no conhecimento e experiência de analistas de sistemas. Os resultados expressivos que foram obtidos nos estudos de casos expressam a viabilidade, praticidade e efetividade da abordagem proposta.

Termos chaves: Engenharia de Software Baseada em Buscas. Sistemas de Rastreamento de Bugs. Seleção de Bugs. Componentes de Software.

ABSTRACT

Nowadays, technology is present in almost every work process and consequently the demand for technological resources is more and more frequent, leading to further changes in software systems, mainly due to additional maintenance requests, and, therefore, a greater number of bugs for correction. Taking into account that bug fixing is one of the factors that impacts on software quality improvement, but at the same time represents a high expenditure of resources in a software project, bug fixing processes represent a critical hurdle for software maintenance and development. Given that the number of existing reported bugs may be greater than the resources needed to correct them, and the fact that many software companies currently work with successive deliveries, called releases, which allow the delivery of parts of the software product, project managers may face the challenge of manually selecting the bugs to be fixed in the next software version, usually based on the criteria of priority and severity, indicated in the respective bug reports, as well as delivery time or cost. For this, project managers adopt the so-called Bug Tracking Systems (BTS), which establish communication channels between software producers and their customers. For large and complex systems, where a substantial number of bug reports are common, a bug selection approach based on the ad-hoc judgment of experts is not efficient and effective, representing a time-consuming, labor-intensive and error-prone strategy. In such a context, exploring Search Based Software Engineering (SBSE) techniques, this dissertation proposes an automated approach for selecting interdependent bugs in component-based software projects, adopting a multiclient model to select the most relevant bugs from the point of view of both the software producer and its customers, but with total cost limited by the budget available in the software project. In order to evaluate the proposed approach, two case studies were carried out, where the search strategy based on genetic algorithm is compared with other strategies based on both exhaustive and random algorithms, and, more realistically, also compared with the selection based on knowledge and experience of systems analysts. The expressive results obtained in the case studies express the feasibility, practicality and effectiveness of the proposed approach.

Keywords: Search Based Software Engineering. Bug Tracking Systems. Bug Selection. Software Components.

1 INTRODUÇÃO

Vive-se uma época em que a tecnologia está presente em quase todos os processos de trabalho, nos quais ocorre o uso intensivo de sistemas de software em diversos contextos, ambientes e dispositivos, demandando, portanto, esforços de especificação e desenvolvimento cada vez mais complexos, que, por sua vez, levam a maiores demandas por mudanças frequentes nas fases de manutenção e evolução dos produtos de software. Tais mudanças podem ocorrer por diversos motivos, incluindo desde problemas relacionados à engenharia de requisitos até problemas relacionados à evolução tecnológica, como também mudanças ligadas aos domínios de negócio dos sistemas de software ou mesmo processos de negócio das empresas.

Por um lado, em relação à engenharia de requisitos, Christel e Kang (1992) apontam que a existência de problemas relacionados à definição de escopo, problemas ligados à dificuldade de compreensão da comunidade de clientes, e problemas associados à natureza volátil dos requisitos dos sistemas de software, isolada ou conjuntamente, podem levar ao desenvolvimento de sistemas insatisfatórios ou até mesmo inaceitáveis, que são sujeitos a mudanças frequentes, gerando altos custos de manutenção. Por outro lado, segundo Koscianski e Soares (2007), a evolução tecnológica teve um efeito dramático na produção de software, uma vez que ocorreram avanços muito rápidos dos recursos de hardware em um curto intervalo de tempo, favorecendo e viabilizando o desenvolvimento de sistemas de software cada vez mais complexos, considerado um dos fatores que exerce influência negativa sobre a qualidade de um projeto de software, dificultando a tarefa de garantir que os requisitos especificados sejam consistentes, e, por conseguinte, levando ao desenvolvimento de sistemas que também são alvo de mudanças frequentes e possuem altos custos de manutenção.

Considerando as mudanças relacionadas aos domínios de negócio dos sistemas de software, bem como as mudanças associadas aos processos de negócio das empresas, as leis de Lehman sobre evolução de software (LEHMAN et al., 1997) denotam a constante necessidade de mudanças e crescimento contínuos para manter a utilidade e qualidade dos sistemas de software. A primeira lei, denominada *mudanças contínuas*, afirma que um sistema de software deve ser continuamente adaptado, senão torna-se aos poucos cada vez menos satisfatório. De forma complementar, a sexta lei, chamada *crescimento contínuo*, estabelece que todo sistema de software deve ter as funcionalidades continuamente ampliadas durante seu ciclo de vida para manter a satisfação dos seus clientes. Consequentemente, o caráter dinâmico dos domínios de negócio e dos processos de negócio impõe a constante necessidade de manutenção e evolução

dos sistemas de software, também corroborado pela sétima lei de Lehman, denominada *qualidade decrescente*, que preconiza que a qualidade dos sistemas de software apresenta uma depreciação progressiva a menos que sejam rigorosamente submetidos a manutenção e adaptação em relação às mudanças no ambiente operacional.

Nesse cenário de mudanças frequentes e contínuas dos sistemas de software durante todo o ciclo de vida, Javed (2004), conforme citado por Dev e Awasthi (2012), acrescenta ainda que há uma significativa relação entre as solicitações de mudanças e os defeitos de software. Observe que, alguns anos antes, Jones (1997) já defendia que as mudanças nos sistemas de software podem provocar bugs, enfatizando que as mudanças nos requisitos durante a fase de desenvolvimento tendem a aumentar a quantidade de bugs encontrados na fase de testes. De forma complementar, Javed (2004) argumenta que o tempo insuficiente gasto na fase de projeto e a comunicação inadequada com o cliente podem ser causas de mudanças de requisitos e assim favorecem o surgimento de bugs nos sistemas de software. Consequentemente, pode-se inferir que as frequentes mudanças nos sistemas de software podem provocar bugs durante todo o ciclo de vida, impactando diretamente nas estimativas de custos e prazos, como também na qualidade dos produtos de software. Desta forma, as mudanças nas fases de especificação, projeto, desenvolvimento, teste, manutenção e evolução têm impactos significativos na quantidade e na severidade dos bugs, sendo mais expressivos quanto mais tardias forem as mudanças em relação a concepção inicial do produto de software.

Dentre os aspectos que levam ao surgimento de bugs, a complexidade dos sistemas associada ao rápido desenvolvimento das tecnologias de software são provavelmente os principais aspectos elencados até aqui, pois, segundo Peng et al. (2013), com o rápido desenvolvimento das tecnologias de software, a escalabilidade e a complexidade dos sistemas de software aumentaram tão fortemente que os desenvolvedores não conseguem controlar os sistemas na perspectiva global, impactando negativamente no planejamento do projeto de software e na qualidade do produto de software resultante. Enfim, em sistemas complexos, a identificação e a avaliação de todos os casos de testes pelos testadores é simplesmente impossível, até mesmo em sistemas aparentemente simples (MYERS, 2004), onde podem existir centenas ou até mesmo milhares de combinações de entradas e saídas, o que pode favorecer e potencializar a ocorrência de erros ou falhas dos respectivos sistemas.

Além dos aspectos já citados, os prazos estabelecidos nos cronogramas dos projetos também representam obstáculos na identificação e avaliação dos casos de testes, contribuindo assim para a existência de mais casos de testes não avaliados nos projetos, e, portanto,

favorecendo também a ocorrência de bugs. Ou seja, considerando os prazos limitados, é importante que a equipe de testes priorize a avaliação das funcionalidades principais, deixando de lado as funcionalidades secundárias (LOUZADA, 2004). O problema do prazo pode estar ligado diretamente com o problema da quantidade de mão-de-obra disponibilizada, pois a medida que a equipe de testes é menor em relação ao tamanho da demanda para um dado prazo, a equipe de testes vê-se obrigada a convergir seus testes para casos essenciais como testes exploratórios ou de unidades.

Por conseguinte, a ocorrência dos bugs leva a outros problemas em projetos de software, tais como *restrições orçamentárias* e *interdependências de bugs*. Em se tratando de restrições orçamentárias ou mesmo de custo da produção de software, de acordo com Boehm (2001), as correções de um sistema podem consumir até 70% do orçamento previsto para o projeto de desenvolvimento do software. Presume-se que dois fatores tendem a potencializar ou contribuir com o aumento deste custo, que são: (i) a complexidade intrínseca relacionada às dependências de bugs; e (ii) sistemas de grande porte, que podem ter maior probabilidade de ocorrência de bugs, requerendo gerenciamento das atividades de correção e por fim mais recursos de mão-de-obra que podem não ser atendidos.

Neste cenário de restrições orçamentárias ou de prazo, é razoável pressupor que, no modelo de desenvolvimento baseado em *releases* ou de entrega incremental, apenas algumas funcionalidades ou bugs são selecionados em cada versão implementada do sistema. Esse modelo, além de amplamente utilizado, permite que os clientes recebam antecipadamente apenas parte da implementação e/ou correção do software. Segundo Sommerville (2011), a atribuição de serviços ou funcionalidades aos incrementos depende da ordem de prioridade atribuída pelos clientes – os serviços ou funcionalidades de mais alta prioridade são implementados e entregues em primeiro lugar. Sendo assim, considerando uma grande quantidade de bugs para corrigir, somada a escassez de recursos financeiros e a exigência de cumprimento dos prazos, a tarefa de correção de bugs configura um cenário com potencial de concorrência entre os bugs nas escolhas dos gerentes de projetos, forçando-os a eleger quais bugs são mais importantes para correção com base em critérios como prioridade e severidade, bem como a avaliação relacionada às dependências entre os bugs.

Acerca da manutenção de sistemas, sabe-se que em projetos de software grandes e complexos como Sistemas Integrados de Gestão Empresarial (*Enterprise Resource Planning - ERP*), é comum a existência de um número considerável de relatórios de bugs. Por exemplo, AVIK et al. (2006) revelam o caso do *Projeto Eclipse* no qual os desenvolvedores contribuintes

relatarem 3246 bugs durante um período de quatro meses, com média de 29 relatórios de bugs por dia. Consequentemente, em sistemas de software grandes e complexos, as abordagens de seleção de bugs baseadas meramente no julgamento ad-hoc de especialistas não são eficientes e eficazes, representando uma tarefa demorada, intensiva em trabalho e propensa a erros, principalmente quando se consideram dependências de bugs que estabelecem relações de precedência e concorrência entre tarefas de correção de bugs.

Deve ser ressaltado que o processo de seleção de bugs também é afetado por outros critérios de avaliação relacionados a escolhas do próprio produtor de software como também seus clientes, por exemplo, a avaliação da prioridade e severidade dos bugs. Neste caso, bugs que tenham uma baixa prioridade e severidade, definidas pelos clientes e/ou fornecedor, terão menos chances de serem selecionados para a próxima *release* do sistema. Por outro lado, bugs com altos níveis de prioridade e severidade terão mais chances de serem selecionados para correção na próxima *release* do sistema.

Outro aspecto sobre os bugs já falado aqui é o critério da dependência entre eles, que afeta de forma direta a ordem que devem ser resolvidos. Pois, quando a dependência de um bug é muito alta (ele depende de vários bugs), isso poderá ensejar na correção prévia de vários bugs para que o mesmo seja corrigido. Este tipo de critério, quando não explorado nas abordagens de seleção de bugs, pode produzir resultados que não atendem de forma satisfatória a um ambiente real de planejamento. Por exemplo, uma solução que não inclui outros bugs precedentes pode incorrer em gerar um retrabalho para tratar novamente uma funcionalidade que já foi corrigida, resultando posteriormente em um gasto maior dos recursos financeiros e de tempo. Observe que, na ausência de tratamento das dependências entre bugs, o impacto negativo nas abordagens de seleção de bugs é cada vez maior quanto mais elevada é a quantidade de relatórios de bugs e quanto mais complexo é a arquitetura do respectivo sistema de software em termos da quantidade de componentes de software e as relações entre suas interfaces providas e requeridas.

Portanto, considerando que a seleção de bugs para correção envolve vários aspectos e restrições, podemos inferir que é uma tarefa complexa para ser realizada puramente baseada no conhecimento, experiência e julgamento de um especialista ou gerente de projetos. Por exemplo, no momento da seleção de um conjunto de bugs para correção, o gerente de projetos pode fazer uma escolha, dada a complexidade da ação, que não leva em consideração todos os fatores e restrições que envolvem o problema, e, assim, entregar a versão final sem considerar

a dependência de bugs e sem representar uma boa seleção em relação aos interesses dos clientes e até do próprio produtor de software.

Essa realidade pode ser compreendida como sendo um *Problema de Seleção de Bugs* (PSB), termo aqui definido, que compartilha algumas semelhanças com o Problema do Próximo Release, do inglês *Next Release Problem* (NRP) (BAGNALL, 2001). Ambos, PSB e NRP, foram propostos para modelar a decisão de balancear os lucros dos produtores de software e a satisfação dos clientes, sob a restrição de um orçamento predefinido do projeto de software, ou seja, ambos possuem características de precedência e concorrência na seleção de bugs e requisitos, respectivamente.

Por um lado, o NRP foca na necessidade de determinar o conjunto de requisitos que devem ser adicionados ao produto de software como parte de sua próxima versão na engenharia de requisitos. Por outro lado, o PSB foca na necessidade de determinar o conjunto de bugs que devem ser corrigidos em um produto de software como parte de sua próxima versão durante a fase de manutenção, mais precisamente durante o processo de correção de bugs.

Normalmente, o NRP é investigado como um problema NP-Difícil na Engenharia de Software Baseada em Buscas, do inglês *Search-Based Software Engineering* (SBSE). Assim, considerando as semelhanças de representação e o compartilhamento de diversos aspectos e restrições a serem considerados, o PSB também pode ser considerado um problema NP-Difícil, e, como tal, pode ser modelado e resolvido usando técnicas SBSE.

Na teoria da complexidade computacional, um problema NP-Difícil (GAREY e JOHNSON, 1979) se caracteriza por ser intratável por algoritmos determinísticos em tempo polinomial (HOANG, 2008), e, portanto, enseja a necessidade de ser tratado por técnicas de busca metaheurística, que, em geral, utilizam métodos de otimização para reduzir o problema NP-Difícil para um problema polinomial, que se caracteriza por prover um balanceamento satisfatório entre qualidade das soluções e tempo de execução (HOANG, 2008), fornecendo soluções próximas da ótima em tempo computacional aceitável e viável (GAREY e JOHNSON, 1979, apud BANOV, 2005).

Considerando o problema de seleção de bugs, com seus vários critérios e restrições de recursos, a proposta deste trabalho é uma abordagem automatizada para a seleção de bugs a serem corrigidos na próxima versão do sistema de software. Esta abordagem busca simultaneamente avaliar e balancear vários fatores concorrentes e conflitantes que podem impactar no problema PSB, que possui critérios tais como: (i) dependências de bugs;

(ii) prioridade e severidade dos bugs do ponto de vista tanto do produtor de software quanto de seus clientes; (iii) custo associado a cada tarefa de correção de bugs; e (iv) orçamento do projeto de software. Portanto, ao avaliar todos estes fatores, a abordagem automatizada pode recomendar um conjunto de bugs que, de forma adequada e aceitável, maximiza a satisfação dos clientes, mantendo o custo dentro do orçamento estabelecido pelo produtor de software.

Nesse sentido, conceitos e técnicas de Engenharia de Software Baseada em Buscas (SBSE) parecem ser uma escolha natural para investigar tal abordagem automatizada para a seleção de bugs de software. Como definido por Harman e Jones (2016), SBSE é uma área relativamente recente que adota algoritmos de otimização baseados em buscas para resolver problemas da engenharia de software. SBSE oferece então um conjunto de soluções automatizadas, semi-automatizadas e adaptativas em situações tipificadas por grandes e complexos espaços de busca com múltiplos objetivos concorrentes e conflitantes (HARMAN, MANSOURI e ZHANG, 2009). Portanto, técnicas SBSE se adequam satisfatoriamente ao problema de seleção de bugs, que inclusive é pouco explorado nas propostas da academia e indústria. E, dentre as propostas encontradas na literatura, nenhuma delas infere a dependência de bugs a partir da arquitetura do projeto de software baseado em componentes.

No entanto, resolver o problema PSB envolve não só definir os critérios estabelecidos para avaliar os melhores conjuntos de bugs para seleção. Envolve também definir e dispor de mecanismos automatizados para colher, organizar e gerenciar os dados que servem para mensurar tais critérios. Para tal, surge a necessidade da adoção de ferramentas para obtenção desses dados dos clientes (equipes de TI das empresas que representam as partes interessadas no produto de software) e do próprio produtor de software (gerente de projeto ou membro(s) da equipe de desenvolvimento da empresa responsável pelo produto de software), servindo assim como canal de comunicação.

Na prática, este canal de comunicação pode ser definido por um Sistema de Rastreamento de Bugs, do inglês *Bug Tracking System* (BTS), que é um tipo de sistema para cadastramento dos bugs, reportados com as suas respectivas características. Existe no mercado diversos sistemas de software que abordam este tema, e alguns deles bem conhecidos são: *Redmine* (2017), *Jira* (2017) e *Bugzilla* (2017).

Enfim, explorando técnicas SBSE, este trabalho propõe uma abordagem automatizada para a seleção de bugs em projetos de software baseados em componentes, adotando um modelo multicliente para selecionar os bugs mais relevantes do ponto de vista tanto do produtor de software quanto de seus clientes, mas com custo total limitado pelo orçamento disponível no

projeto de software. A proposta avalia restrições relacionadas as dependências de bugs, estabelecendo relações de precedência e de concorrência entre suas respectivas tarefas de correção de bugs. Como uma contribuição adicional, assumindo que as dependências entre componentes de software implicam dependências entre os bugs associados, a abordagem proposta explora a arquitetura baseada em componentes do projeto de software como um meio para identificar dependências entre bugs.

1.1 OBJETIVOS

O objetivo geral deste trabalho é elaborar e avaliar uma abordagem automática de seleção de bugs, que, explorando técnicas da Engenharia de Software Baseada em Buscas, e, empregando conceitos da Engenharia de Software Baseada em Componentes, maximiza a satisfação dos clientes e mantém o custo dentro do orçamento estabelecido pelo produtor de software, minimizando assim o esforço do gerente de projetos na seleção de bugs.

De forma mais detalhada, os objetivos específicos são:

- Explorar conceitos que advêm de modelos baseados em componentes para identificar as dependências entre bugs;
- Definir e adaptar uma metaheurística de busca para o problema de seleção de bugs;
- Identificar critérios de avaliação de bugs que serão os parâmetros de entrada do modelo matemático da abordagem proposta;
- Realizar estudos de casos para avaliação da abordagem proposta, incluindo uma análise comparativa com a seleção de bugs baseada no julgamento de especialistas;

1.2 ESTRUTURA DO TRABALHO

O restante do trabalho está organizado da seguinte forma. No Capítulo 2 é apresentada a fundamentação teórica, abordando os principais conceitos das áreas de pesquisa relacionadas à proposta deste trabalho, a saber, seleção de bugs e algoritmos genéticos. No Capítulo 3, uma comparação com os trabalhos relacionados é apresentada, evidenciando as contribuições da abordagem proposta. No Capítulo 4, a abordagem automática para seleção de bugs é apresentada, detalhando sua estrutura, fases, parâmetros e modelagem matemática. Em seguida, no Capítulo 5, a abordagem proposta é avaliada em dois estudos de casos, baseados em um projeto de software real e organizados em 14 experimentos, onde a estratégia de busca baseada em algoritmo genético é comparada com as estratégias baseadas em algoritmo exaustivo e

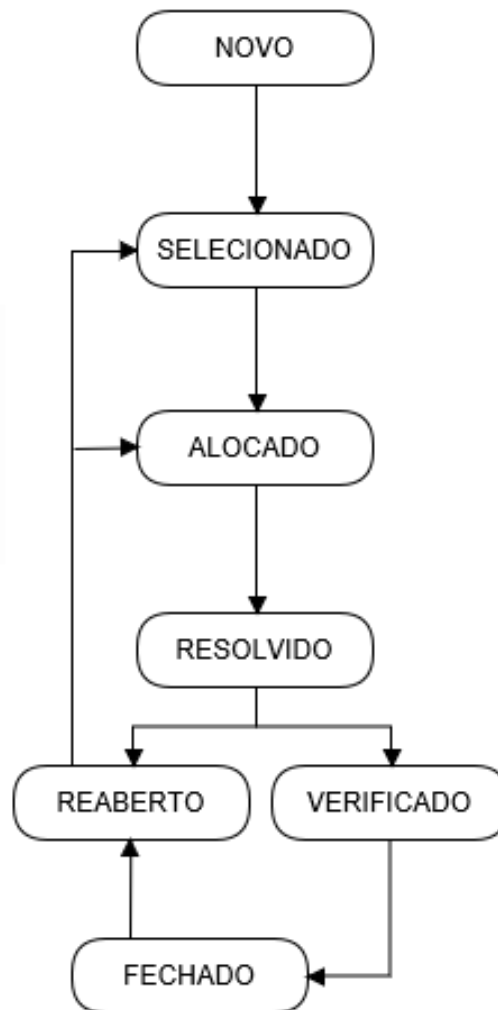
algoritmo aleatório, e, de forma mais realista, também comparada com a seleção baseada no conhecimento, experiência e julgamento de especialistas da equipe de manutenção do referido projeto de software. Por fim, o Capítulo 5 apresenta as considerações finais, identificando e discutindo algumas limitações e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção são abordados os temas inerentes ao conhecimento necessário para entender a proposta do presente trabalho, a saber: (i) Seleção de Bugs; e (ii) Engenharia de Software Baseada em Buscas. Com relação à seleção de bugs, inicialmente, esta seção apresenta as fases do processo de correção de bugs e os estados do ciclo de vida dos bugs, tornando evidente a fase em que este trabalho está inserido. Em seguida, ferramentas para gerenciamento e cadastramento de bugs são brevemente discutidas, sinalizando critérios e restrições que podem ser adotados para qualificar e quantificar os bugs reportados e que podem ser explorados durante o processo de seleção de bugs. Por fim, ainda em relação à seleção de bugs, o conceito de dependência entre bugs é apresentado, introduzindo o enfoque em projetos de software baseados em componentes. Posteriormente, com relação à Engenharia de Software Baseada em Buscas, inicialmente é apresentado um panorama geral das áreas e problemas da engenharia de software que são tratados por técnicas de busca metaheurística, e, em seguida, é realizado um aprofundamento da discussão sobre algoritmos genéticos, uma vez que é a metaheurística adotada na presente proposta.

2.1 SELEÇÃO DE BUGS

Processos de correção de bugs são adotados pelas equipes de desenvolvimento para realizar o cadastramento e o gerenciamento de relatórios de bugs durante todo o ciclo de vida dos bugs nos processos de manutenção e evolução dos sistemas de software, cobrindo desde o relato inicial até a resolução final dos bugs. Segundo Mamdouh (2013), durante o ciclo de vida de um bug, existem diferentes estados que um relatório de bug pode experimentar. Como ilustrado na Figura 1, cada bug pode passar por até sete estados: *NOVO*, *SELECIONADO*, *ALOCADO*, *RESOLVIDO*, *VERIFICADO*, *REABERTO*, *FECHADO*. É importante ressaltar que a Figura 1 é uma tradução, adaptação e extensão da proposta de ciclo de vida apresentada por Mamdouh (2013), que originalmente indica apenas seis estados (*NEW*, *ASSIGNED*, *RESOLVED*, *VERIFIED*, *REOPEN*, *CLOSED*). No entanto, considerando o foco do trabalho na fase de seleção de bugs, o estado *ASSIGNED* foi desmembrado em dois estados (*SELECIONADO* e *ALOCADO*), permitindo uma clara separação entre as etapas de seleção de bugs e alocação de desenvolvedores.

Figura 1 - Ciclo de vida de um bug

Fonte: adaptado de Mamdough (2013).

Inicialmente, quando o relatório de um bug é criado e preenchido, seu estado é atribuído como *NOVO*. Neste ponto, inicia-se o processo de triagem ou validação do bug, que, de acordo com Netto et al. (2010), é a fase na qual os membros das equipes de qualidade e de desenvolvimento avaliam se o novo relatório de bug pode de fato ser classificado como defeito de software. Após o correto cadastramento e validação do bug, conjuntamente, o gerente de projetos e a equipe de desenvolvimento podem associar informações que caracterizam a relevância e o impacto do bug no sistema de software. Além disso, ainda nesta fase, o esforço requerido para resolução do bug também pode ser estimado, possivelmente preenchendo o relatório do bug com outras informações complementares, por exemplo, relacionadas ao custo financeiro e ao tempo requerido para realizar a correção do bug.

Neste ponto, considerando todos os relatórios de bugs que devem ser corrigidos, incluindo aqueles no estado *NOVO* e também aqueles no estado *REABERTO* (discutido mais adiante), o gerente de projetos inicia a fase de seleção dos bugs que serão corrigidos na próxima versão do sistema de software, possivelmente avaliando e correlacionando as diversas informações técnicas e não-técnicas presentes nos relatórios de bugs, incluindo o contexto e as restrições atuais relacionadas à equipe de desenvolvimento e à própria organização desenvolvedora do produto de software, e, até mesmo, associadas aos seus atuais e potenciais clientes. Concluída a fase de seleção, os bugs selecionados para correção passam para o estado *SELECIONADO*. É importante destacar que a fase de seleção de bugs é o foco principal da abordagem proposta nesta dissertação.

Em seguida, considerando os bugs selecionados para correção, o gerente de projetos deve dar início a fase de alocação de bugs, cujo propósito é designar os desenvolvedores que serão responsáveis pela correção de cada bug previamente selecionado. Aqui, o gerente de projetos deve avaliar e correlacionar as informações presentes nos relatórios dos bugs selecionados com as informações relativas ao perfil técnico de cada membro da equipe de desenvolvimento, além da necessidade de balancear conflitos entre os prazos estipulados e a disponibilidade da equipe de desenvolvimento. Após a realização da fase de alocação de desenvolvedores, os bugs alocados para correção passam para o estado *ALOCADO*.

Agora, os desenvolvedores alocados fazem suas respectivas tarefas de correção dos bugs, que, após resolvidos, passam para o estado *RESOLVIDO*. A partir deste momento, existem duas possibilidades no ciclo de vida dos bugs. A primeira possibilidade é seguir o caminho natural, no qual a equipe de testes deve realizar a verificação e a validação dos bugs para atestar a efetiva correção dos mesmos, quando então podem passar para o estado *VERIFICADO*. A outra possibilidade é a própria equipe de desenvolvimento rapidamente perceber que determinados bugs não foram de fato corrigidos, e, assim, modificar seus estados para *REABERTO*, quando então voltam a ser candidatos aos processos de seleção (*SELECIONADO*) ou alocação de desenvolvedores (*ALOCADO*).

Considerando que a verificação e a validação dos bugs foram realizadas com sucesso, os bugs podem finalmente passar para o estado *FECHADO*. Caso contrário, os bugs podem ser reclassificados, passando assim para o estado *REABERTO*, e, desta forma, tornam-se novamente alvo dos processos de seleção (*SELECIONADO*) ou alocação de desenvolvedores (*ALOCADO*).

Por fim, observe que, mesmo no estado *FECHADO*, bugs podem voltar para o estado *REABERTO* sempre que a equipe de desenvolvimento ou a equipe de testes detectarem que as atividades de verificação e validação não identificaram que, sob certas circunstâncias, os problemas decorrentes dos bugs ainda persistem parcial ou integralmente.

Como previamente indicado, o foco da abordagem proposta nesta dissertação é o processo de seleção de bugs. Portanto, os processos de triagem e alocação estão fora do escopo desta dissertação, embora estejam bastante relacionados. Por um lado, durante o processo de triagem, o gerente de projetos e a equipe de desenvolvimento alimentam os relatórios de bugs com informações importantes que caracterizam os bugs candidatos e que são imprescindíveis à abordagem aqui proposta para selecionar o conjunto de bugs a serem corrigidos na próxima versão do sistema de software em questão. Por outro lado, os bugs selecionados pela abordagem proposta podem ser adotados como ponto de partida pelo processo de alocação de desenvolvedores, que já pode ser instanciado conhecendo os bugs mais relevantes para a próxima versão do sistema de software, e, por serem selecionados considerando as relações de precedência e concorrência entre bugs, facilitam inclusive a elaboração do cronograma de tarefas de correção dos bugs.

2.1.1 SISTEMAS DE RASTREAMENTO DE BUGS

Em sistemas de médio e grande porte, a quantidade de relatórios de bugs pode se tornar bastante elevada, e, além disso, diversos colaboradores devem ter acesso às informações durante todo o ciclo de vida dos bugs. Consequentemente, neste cenário, há a necessidade de utilização de ferramentas para cadastramento e gerenciamento dos bugs, denominadas Sistemas de Rastreamento de Bugs, do termo original em inglês *Bug Tracking Systems* (BTS), permitindo a automatização de várias atividades do processo de correção de bugs, de modo que facilite a utilização das informações contidas nos relatórios de bugs (SERRANO, 2005).

Como meio de automatizar e suportar processos de correção de bugs, um BTS é uma ferramenta de software que armazena relatórios de bugs, fornecendo uma visão clara e centralizada de solicitações de desenvolvimento e seus estados, e, adicionalmente, disponibilizando informações sobre a produtividade da equipe de desenvolvimento (D'AMBROS, LANZA, PINZGER, 2007). Alguns dos sistemas utilizados para gerenciamento dos relatórios de bugs são: *Redmine* (2017), *Jira* (2017), *Bugzilla* (2017), *Lean Testing* (2018), *Mantis Bug Tracker* (2018) e *Lighthouse* (2018).

É importante destacar que, tanto na prática das equipes de desenvolvimento quanto na literatura da academia e indústria relacionada a processos de correção de bugs, as informações disponibilizadas nos relatórios de bugs dos BTSs são amplamente adotadas como dados de entrada para definir a importância e o impacto de cada bug no sistema. Portanto, é necessário identificar e entender o significado das informações que em geral estão presentes nos relatórios de bugs dos BTSs.

Conceitualmente, um relatório de bug é um documento ou registro que explica os detalhes de uma falha ou erro encontrado em um pacote ou componente de software que faz parte do respectivo sistema de software. Por um lado, em geral, bugs podem ser identificados e cadastrados tanto pelos colaboradores, equipes de TI que fazem parte das organizações clientes, quanto pelos desenvolvedores da organização produtora. Por outro lado, bugs são posteriormente validados apenas pela equipe de desenvolvimento da organização produtora.

Nos BTSs atualmente disponíveis, os relatórios de bugs normalmente incluem uma grande variedade de informações. Para exemplificar na prática, a Figura 2 exibe um relatório de bug na ferramenta *Redmine*, que é uma ferramenta de código livre amplamente adotada em projetos de software da academia e indústria, e, especificamente, adotada pela Universidade Federal da Paraíba (UFPB) no projeto de software usado nos estudos de caso desta dissertação.

Figura 2 - Cadastro de relatório de bug no Redmine

The screenshot shows the 'Nova tarefa' (New task) form in the Redmine application. The top navigation bar includes 'Graduação' and a search box. Below it, a menu bar contains 'Visão geral', 'Atividade', 'Planejamento', 'Tarefas', 'Nova tarefa' (selected), 'Documentos', 'Arquivos', and 'Configurações'. The form itself is titled 'Nova tarefa' and contains the following fields and options:

- Tipo ***: A dropdown menu with 'Bug' selected.
- Título ***: A text input field.
- Descrição**: A rich text editor with various formatting icons (bold, italic, underline, strikethrough, link, unlink, list, etc.).
- Situação ***: A dropdown menu with 'Aberta' selected.
- Prioridade ***: A dropdown menu.
- Atribuído para**: A dropdown menu.
- Versão**: A dropdown menu with a green plus icon next to it.
- Tipo do bug**: A dropdown menu.
- Versão do bug**: A dropdown menu.
- Testador**: A dropdown menu.
- Tarefa pai**: A dropdown menu.
- Início**: A date/time picker.
- Data prevista**: A date/time picker.
- Tempo estimado**: A text input field with a unit selector set to 'Horas'.
- Ticket Otrs**: A text input field.
- Ticket iProject**: A text input field.
- Impacto**: A text input field.

Fonte: UFPB (2016).

É importante ressaltar que, nas diversas ferramentas disponíveis, não existe uma padronização na terminologia adotada para as informações que compõem um relatório de bug. Portanto, a seguir será apresentada uma breve descrição de algumas das principais informações presentes nos relatórios de bugs, a saber:

- Título – é a identificação textual do bug. Pode auxiliar na verificação de duplicidade de relatórios de bugs, uma vez que, ao ser informado o título, a ferramenta pode habilitar recursos de auto complementação a partir de informações de outros bugs previamente cadastrados.
- Versão – identifica a versão do sistema de software onde o bug foi identificado.
- Descrição – apresenta com mais detalhes o cenário em que ocorre o bug relatado.
- Prioridade – identifica a importância do bug na perspectiva do produtor e seus clientes. Em geral, a prioridade é sinalizada usando termos ou valores simples tais como *baixa/média/alta* ou *1/2/3*.
- Severidade – identifica o impacto do bug no sistema na perspectiva do produtor e seus clientes. Comumente, a severidade é indicada usando termos simples tais como *crítica/grave/trivial*.
- Artefato alvo – representa o artefato de software onde está localizado o bug reportado. É importante frisar que, dependendo da forma como o projeto de software é estruturado e codificado, o artefato alvo pode ser por exemplo uma função, classe, pacote, módulo ou componente de software.
- Dependências – define relações de precedência e concorrência do bug reportado em relação a outros bugs já existentes. Por exemplo, por um lado, um bug que depende de outro bug somente pode ser corrigido após a correção deste outro. Mas, por outro lado, bugs que não possuem relação de dependência podem ser corrigidos simultaneamente.
- Custo – representa o esforço requerido para correção do bug. Observe que o esforço pode ser mensurado de diferentes formas, por exemplo, sendo indicado em unidades de tempo, homem-hora ou até mesmo valor financeiro.

Entre essas informações, a abordagem proposta tem especial interesse nos seguintes tipos:

- (i) prioridade e severidade dos bugs – revelam a importância e o impacto dos bugs;
- (ii) dependências de bugs – impõem restrições na seleção dos bugs;
- (iii) artefatos de software alvo dos bugs – indicam os componentes de software onde ocorrem os bugs reportados;
- (iv) custos dos bugs – indicam os esforços requeridos pelas tarefas de correção dos bugs. Em

função da importância destas informações na abordagem proposta, a seguir as mesmas serão discutidas em mais detalhes.

2.1.2 PRIORIDADE E SEVERIDADE

Segundo D'Ambros, Lanza e Pinzger (2007), a prioridade e a severidade representam critérios para mensurar a criticidade dos bugs. Como já mencionado, estes critérios podem fazer parte do relatório de bug em diversas ferramentas, inclusive podendo ser visualizado no cadastro de bug do *Redmine* (Figura 2). É importante frisar que as ferramentas podem adotar diferentes nomenclaturas para a prioridade e severidade. Por exemplo, no *Redmine*, a severidade é indicada usando o campo denominado impacto. Já no *Bugzilla*, os termos prioridade e severidade são conjugados em um único campo denominado importância.

O critério de *prioridade* de um bug é bastante discutido na literatura, existindo aspectos ligeiramente diferentes entre variados autores. Por exemplo, na perspectiva de Zhang et al. (2015), a prioridade indica a ordem em que o bug deve ser corrigido. No entanto, Dreyton et al. (2016) define prioridade como o nível de precedência que a resolução de um bug apresenta em relação a outros.

A prioridade de um bug pode ser avaliada e mensurada de várias formas. Por exemplo, Yuan Tian (2013) propõe um método automático de atribuição de prioridades baseado em redes neurais. Desta forma, as prioridades dos bugs são definidas de forma automática com base na análise textual dos relatórios de bugs usando técnicas de redes neurais. Neste estudo, a prioridade é classificada usando os termos *P1*, *P2*, *P3*, *P4* e *P5*, onde *P1* é a mais alta prioridade e *P5* é a mais baixa prioridade.

De forma diferente, Netto et al. (2016) defende que o nível de prioridade pode ser atribuído pelo usuário final com base em um mecanismo de votação. Nesta proposta, é possível permitir que clientes do sistema de software votem para indicar os relatórios de bugs mais prioritários. Quanto mais votos um determinado relatório de bug receber, mais prioritária será a sua correção. Novamente, esta proposta adota a classificação de prioridades usando os termos *P1* até *P5*, que também representam a maior e a menor prioridade, respectivamente.

Como pode ser observado, em geral, as diversas propostas de atribuição de prioridade definem uma classificação baseada nos termos *P1*, *P2*, *P3*, *P4* ou *P5*. E, como já mencionado, o termo *P1* designa bugs de mais alta prioridade, enquanto o termo *P5* sinaliza bugs de mais baixa prioridade. Consequentemente, os termos *P1* até *P5* também são adotados na presente abordagem proposta.

Agora, na perspectiva de Zhang et al. (2015), a severidade indica o impacto que o bug possui em relação ao sistema. De forma mais ampla, a severidade pode ser vista como uma característica fundamental, pois pode indicar o impacto dos bugs para o usuário final, ou uma medida do nível de criticidade do bug relatado, indicando a consequência da falha no sistema na perspectiva da parte interessada, a frequência em que ocorre ou o efeito para outras funcionalidades relacionadas e interdependentes.

De forma similar ao que acontece com a prioridade, também não existe uma padronização para representação da severidade. Por exemplo, baseado no *Bugzilla*, o trabalho de Netto et al. (2010) aponta que a severidade pode ser classificada em cinco tipos, a saber:

Blocker – é o impacto mais indesejado, pois impede a continuidade das atividades de desenvolvimento e teste;

Critical – causa algum tipo de parada do sistema, com perda de informação;

Major – compromete o comportamento das funções críticas do sistema;

Minor – afeta apenas funções não críticas do sistema;

Trivial – representa problemas de menor impacto, tais como erros de ortografia e alinhamento de textos.

Por exemplo, um bug que provoca a parada completa do sistema, impedindo o usuário de realizar qualquer procedimento, pode ser considerado crítico. Já outro bug que apresenta uma mensagem com erros de ortografia ou pontuação pode ser considerado apenas trivial.

Como já sinalizado, os termos adotados para sinalizar a severidade também é definida no *Bugzilla*, porém com a inclusão dos termos *normal* e *enhancement*, que, quando posicionados para indicar a criticalidade, os termos ordenados são: *blocker*, *critical*, *major*, *normal*, *minor*, *trivial* e *enhancement*. Assim, no *Bugzilla*, o termo *blocker* representa a severidade de maior impacto, enquanto o termo *enhancement* sinaliza a severidade de menor impacto. Apesar do *Bugzilla* fornecer estas opções adicionais de severidade, a abordagem aqui proposta não as adota pelos seguintes motivos. Por um lado, a opção *normal* pode criar uma zona cinza ou uma opção padrão, como argumentou Lamkanfi et al. (2010). Por outro lado, a opção *enhancement* representa apenas uma requisição de melhoria, e, não necessariamente um problema ou falha no sistema de software.

2.1.3 DEPENDÊNCIAS DE BUGS

No contexto do Problema do Próximo Release (NRP), que, conforme discutido anteriormente, consiste em selecionar os requisitos que serão implementados na próxima entrega do sistema, Yeltsin (2013) afirma que “*um dos aspectos mais importantes para a atividade de seleção de requisitos é a análise das relações, ou dependências entre os requisitos*”. No entanto, para o referido autor, as dependências entre os requisitos fazem com que a modelagem do problema NRP torne-se ainda mais complexa.

Ainda no contexto do problema NRP, as propostas existentes na literatura tratam as dependências de requisitos de diferentes formas. Por exemplo, em sua proposta, Gonçalves (2013) adota a análise de pontos de função para definir as dependências de requisitos. De acordo com o referido autor, a análise de pontos de função permite obter soluções mais reais na produção de um software, oferecendo soluções de melhor qualidade. Portanto, o tratamento de dependências de requisitos no problema NRP tem como desafio balancear a qualidade das soluções e a complexidade na modelagem do problema.

Considerando a similaridade representacional entre o problema do próximo *release* (NRP) e o problema da seleção de bugs (BSP), que é o alvo da presente dissertação, pode-se concluir que, enquanto no problema NRP, a modelagem das dependências entre requisitos melhora a qualidade das soluções, no problema BSP, a modelagem das dependências entre bugs também pode melhorar a qualidade das soluções. Portanto, no problema BSP, o desafio também é o balanceamento entre a qualidade das soluções e a complexidade da modelagem do problema quando introduzidas as dependências entre bugs.

No sentido de identificar e incorporar diferentes tipos de dependências entre bugs na abordagem aqui proposta para o problema BSP, a seguir são introduzidos dois tipos de dependências entre requisitos que são apontados por Yeltsin (2013):

- *R1 AND R2* – requisito *R1* depende do requisito *R2* e vice-versa. Em outras palavras, no contexto do problema NRP, ambos requisitos devem ser conjuntamente selecionados para implementação no próximo *release*, definindo uma relação de concorrência obrigatória no processo de seleção.
- *R1 REQUIRES R2* – requisito *R1* depende do requisito *R2*, mas não vice-versa. Ou seja, o requisito *R1* somente pode ser selecionado para implementação no próximo *release* se o requisito *R2* também for selecionado, definindo uma relação de precedência

obrigatória no processo de seleção. No entanto, por ser independente, o requisito *R2* pode ser selecionado isoladamente para implementação no próximo *release*.

Embora não apontado por Yeltsin (2013), dois ou mais requisitos que são independentes entre si, aqui representado por *R1 OR R2*, podem ser selecionados para implementação em qualquer ordem ou mesmo simultaneamente, portanto, definindo uma relação de precedência ou concorrência opcional, a depender do interesse do gerente de projetos, dando assim mais liberdade no processo de seleção dos respectivos requisitos.

De forma análoga, os mesmos tipos de dependências e relações de precedência e concorrência, seja obrigatória ou opcional, podem ser definidas entre bugs em um processo de seleção de bugs. Desta forma, contextualizando para o presente trabalho, quando o gerente de projetos selecionar um bug para correção, ele deve considerar e selecionar também outros bugs que o primeiro possa depender. Por exemplo, no *Bugzilla*, um bug que depende de outro somente pode passar para o estado *RESOLVIDO* depois da correção do outro bug, definindo assim uma relação de precedência obrigatória do primeiro bug em relação ao segundo bug. No entanto, observe que, dois ou mais bugs que não dependem uns dos outros podem ser corrigidos em qualquer sequência ou mesmo simultaneamente, definindo assim uma relação de precedência ou concorrência opcional na correção destes bugs.

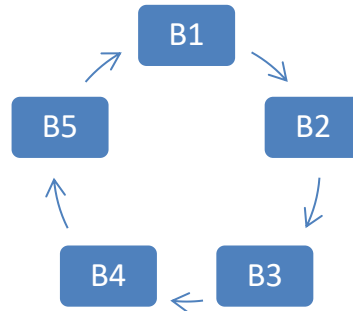
De forma mais conceitual, no primeiro caso *B1 AND B2*, onde o bug *B1* depende do bug *B2* e vice-versa, ambos devem ser conjuntamente selecionados para correção, e, assim, definem uma relação de concorrência obrigatória no processo de seleção. No segundo caso *B1 REQUIRES B2*, onde apenas o bug *B1* depende do bug *B2*, mas não o inverso, o bug *B1* somente por ser selecionado para correção se o bug *B2* também for selecionado, e, assim, definem uma relação de precedência obrigatória no processo de seleção. Por fim, no caso *B1 OR B2*, onde os bugs *B1* e *B2* são independentes, não há qualquer restrição de seleção, e, portanto, definem uma relação de precedência ou concorrência opcional no processo de seleção.

Com o entendimento firmado das dependências básicas entre bugs, pode-se introduzir alguns tipos derivados de dependências de bugs, a saber: dependência cíclica, dependência estrela e dependência linear.

Na dependência cíclica, representada na Figura 3, as dependências par-a-par entre dois ou mais bugs formam um ciclo, e, portanto, definem uma relação de concorrência obrigatória, na qual todos os bugs do ciclo devem ser conjuntamente selecionados para correção. Neste

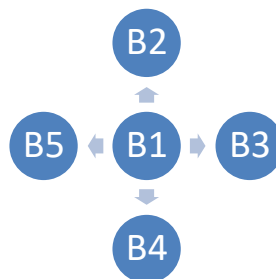
caso, por transitividade, diz-se que cada bug depende de todos os outros do ciclo. Observe que a dependência básica do tipo *B1 AND B2* é uma dependência cíclica de apenas dois bugs.

Figura 3 - Dependência cíclica



Na dependência estrela, conforme ilustrada na Figura 4, as dependências par-a-par entre dois ou mais bugs formam uma estrela, na qual um bug central depende diretamente de diversos outros bugs de borda. Neste caso, a seleção do bug central define uma relação de concorrência obrigatória com os demais bugs de borda. Porém, individualmente, por serem independentes, os bugs de borda podem ser selecionados isoladamente para correção no próximo *release*. Logo, o bug central depende de todos os demais bugs de borda, que por sua vez são independentes. Observe que a dependência básica do tipo *B1 REQUIRES B2* é uma dependência estrela de apenas dois bugs.

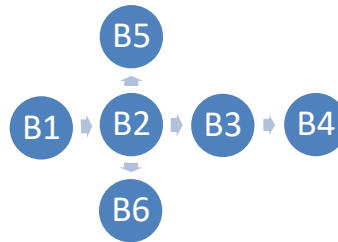
Figura 4 - Dependência estrela



Na dependência linear, como apresentada na Figura 5, as dependências par-a-par entre dois ou mais bugs formam uma lista encadeada, na qual um bug depende de outro, que, por sua vez depende de outro, e assim sucessivamente. Neste caso, a seleção de um bug em uma dada posição da lista define uma relação de concorrência obrigatória com os demais bugs sucessores da lista. Porém, por ser independente, o bug no fim da lista pode ser selecionado isoladamente para correção no próximo *release*. Observe que a dependência básica do tipo *B1 REQUIRES B2* é também uma dependência linear de apenas dois bugs.

Figura 5 – Dependência linear

As dependências compostas aqui apresentadas demonstram apenas alguns tipos de relações que podem ocorrer entre os bugs de um sistema. É importante destacar que diferentes possibilidades de entrelaçamento de variados tipos de dependências compostas podem ocorrer. Por exemplo, a Figura 6 ilustra a combinação de dependências linear e estrela. Neste caso, observe que o bug *B1* depende dos demais bugs. O bug *B2* depende dos demais bugs, exceto do bug *B1*. Já o bug *B3* depende apenas do bug *B4*. Note que os bugs *B4*, *B5* e *B6* são independentes.

Figura 6 – Dependência estrela e linear

No que tange às técnicas adotadas para identificar as dependências, observe que, enquanto o trabalho de Gonçalves (2013) adota uma técnica baseada na análise de pontos de função para definir as dependências de requisitos no problema NRP, de uma forma análoga, mas ao mesmo tempo inovadora no contexto do problema BSP, a abordagem proposta nesta dissertação adota uma técnica baseada em componentes para rastrear as dependências de bugs.

Por um lado, considerando a analogia entre a técnica baseada em pontos de função e a técnica baseada em componentes, a semelhança reside no fato que, na proposta de Gonçalves (2013) para o problema NRP, dependências entre pontos de função de dados e transacional implicam em dependências entre requisitos a serem selecionados para a próxima versão do sistema. No caso da proposta desta dissertação para o problema BSP, de forma análoga, dependências entre os componentes de software que compõem o atual sistema de software implicam em dependências entre os respectivos bugs a serem selecionados para a próxima versão do sistema.

Como ponto de partida e inspiração, a abordagem aqui proposta baseia-se diretamente na abordagem proposta por Burity e Elias (2011) para identificar dependências entre componentes de software. Na referida proposta, Burity e Elias apresentam duas técnicas,

denominadas *dependências por interfaces* e *dependências por operações*, ambas baseadas na análise de diferentes artefatos produzidos durante a especificação dos projetos de software baseados em componentes, a saber: as especificações de interfaces; as especificações de componentes; e a arquitetura do sistema de software, que representa de forma estrutural as relações de dependências das interfaces providas e requeridas entre os componentes de software constituintes.

Por um lado, a técnica de *dependências por interfaces* explora as relações estruturais de grossa granularidade entre interfaces providas e interfaces requeridas para determinar as dependências entre componentes. Neste caso, se o componente *C1* requer uma interface *I* que é provida pelo componente *C2*, ou seja, *C1* ativa funcionalidades implementadas por *C2*, então diz-se que o componente *C1* depende do componente *C2*. Por outro lado, a técnica de *dependências por operações* explora as relações funcionais de fina granularidade entre operações providas e operações requeridas para determinar as dependências entre componentes. Agora, neste caso, se o componente *C1* requer uma operação *Op* que é provida pelo componente *C2*, ou seja, *C1* ativa uma funcionalidade específica implementada por *C2*, então diz-se que o componente *C1* depende do componente *C2*.

Como pode ser observado, as técnicas propostas por Burity e Elias (2011) diferem na granularidade do objeto avaliado, implicando em uma relação de compromisso entre precisão e esforço. Na perspectiva de esforço, por um lado, a técnica de *dependências por interfaces* impõe um baixo esforço em função de avaliar apenas interfaces providas e requeridas. Mas, por outro lado, a técnica de *dependências por operações* impõe um esforço muito maior pois é necessário avaliar individual e profundamente cada operação provida e requerida de todas as interfaces via diagramas de sequência.

Já na perspectiva de precisão, enquanto a técnica de *dependências por interfaces* pode derivar falsos-positivos em dependências transitivas, a técnica de *dependências por operações* é bastante precisa na derivação destas dependências transitivas. Em outras palavras, se o componente *C1* ativa funcionalidades do componente *C2*, que, por sua vez, ativa funcionalidades do componente *C3*, a técnica de *dependências por interfaces* infere transitivamente que *C1* depende de *C3*, pois avalia apenas relações estruturais do projeto arquitetural. Já a técnica de *dependências por operações* somente infere transitivamente que *C1* depende de *C3*, se *C1* chama uma operação de *C2*, que, por sua vez, chama uma operação de *C3*, pois avalia as relações funcionais do projeto arquitetural.

Considerando o balanceamento entre precisão e esforço, no contexto desta dissertação, a opção inicial para identificar as dependências entre componentes foi explorar a técnica de *dependências por interfaces*, pois impõe um esforço manual ou computacional consideravelmente menor na análise da especificação do projeto de software. Porém, caso seja de interesse uma melhor precisão das soluções, embora seja custoso manual ou computacionalmente, é bastante intuitivo adaptar a proposta aqui apresentada, passando a explorar a técnica de *dependências por operações* na análise da especificação do projeto de software.

Como originalmente pensado na abordagem proposta nesta dissertação, a identificação de dependências de bugs a partir das dependências de componentes impõe a necessidade do sistema possuir o artefato que apresenta e detalha a arquitetura baseada em componentes do sistema de software. No entanto, caso a especificação arquitetural não esteja disponível, as dependências entre bugs podem ser similarmente derivadas a partir de outros artefatos, como por exemplo um diagrama de submódulos, pacotes ou até mesmo classes. E, em último caso, quando não disponíveis tais diagramas, o código fonte pode ser avaliado de forma automática para mapear as dependências entre bugs.

2.1.4 CUSTO DO BUG

Conforme citado anteriormente, os relatórios de bugs também podem conter informações sobre o custo dos bugs, que representa o custo financeiro ou o esforço humano necessário para corrigir os respectivos bugs. Claramente, o custo deve ser considerado na seleção dos bugs, de modo que os gerentes de projetos possam atender restrições orçamentárias ou de disponibilidade de mão-de-obra, fornecendo o máximo valor de negócio para as organizações, e, assim, favorecendo a saúde financeira e o bem estar das mesmas a longo prazo.

Consequentemente, considerando um sistema de software que possui uma grande quantidade de relatórios de bugs para correção, e, ao mesmo tempo, a organização desenvolvedora deste sistema possui restrições de recursos orçamentários, prazo de entrega ou disponibilidade de mão-de-obra, o gerente de projetos é obrigado a realizar entregas incrementais e parciais das correções dos referidos bugs, sendo, portanto, necessário realizar a seleção do conjunto de bugs que devem ser corrigidos na próxima entrega do sistema.

Neste contexto, sabendo que nem sempre todos os bugs podem ser corrigidos na próxima versão do sistema, é necessário conhecer as principais técnicas adotadas na academia e indústria para apoiar na estimativa do esforço requerido pelas tarefas de correção dos bugs.

Observe que, se um gerente de projetos não utiliza tais técnicas de estimativa de esforço, os processos de correção de bugs tornam-se altamente imprecisos e propensos a erros nas estimativas de custo financeiro ou prazo de realização das tarefas de correção dos bugs.

É importante destacar que está fora do escopo desta dissertação propor ou adotar técnicas de estimativa relacionadas ao custo ou esforço de correção de bugs, pois a abordagem aqui proposta apenas toma como parâmetro de entrada o valor associado ao custo de cada bug, que é determinado pelo gerente de projeto adotando técnicas de estimativa que melhor se adequam aos interesses e experiências da equipe de desenvolvimento. Desta forma, o objetivo deste tópico é apenas mostrar que é possível estimar o esforço requerido, e, assim, calcular o custo de correção de um bug.

Em geral, para calcular o custo de correção de um dado bug, inicialmente, o gerente de projetos deve estimar o esforço de correção, que pode ser medido por exemplo em homem-hora ou homem-dia. Posteriormente, conhecendo o valor homem-hora adotado na organização desenvolvedora, o gerente de projetos pode estimar o custo de correção do bug. Note que o valor homem-hora pode ser estimado com base na produtividade dos membros da equipe de desenvolvimento, considerando dados históricos das atividades realizadas em projetos anteriores ou até mesmo em atividades finalizadas no projeto atual. Ao estimar o esforço de correção dos bugs, o gerente de projetos identifica os recursos necessários para execução das atividades e pode fornecer uma estimativa de entrega para os clientes com base nos recursos atualmente disponíveis (PAGNO, 2010).

Segundo Sousa (2011), a tarefa de estimar o esforço de uma atividade tem se mostrado bastante complexa, pois os métodos existentes tendem a servir mais para uma realidade específica de processo de desenvolvimento. Desta forma, cabe ao gerente de projetos adotar a técnica de estimativa de esforço e custo que melhor se adequa ao processo de desenvolvimento adotado na organização.

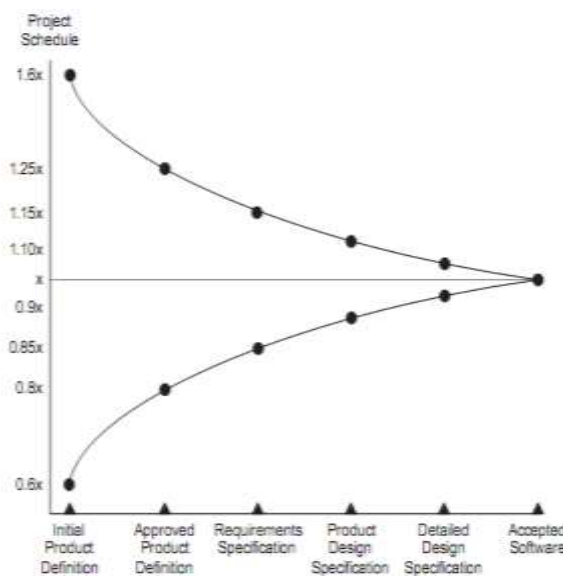
Segundo Pressman (2006), a estimativa de esforço e custo em projetos de software nunca será uma ciência exata, pois existem muitas variáveis relacionadas a aspectos humanos, técnicos, ambientais e políticos que podem impactar e modificar tais estimativas. No entanto, Pressman (2006) sugere algumas opções que podem ser adotadas de forma isolada ou combinada para estimar o esforço e o custo dentro de um patamar de risco aceitável, a saber:

- 1) Adiar a estimativa até que o projeto esteja mais adiantado;
- 2) Basear a estimativa em projetos similares já concluídos;

- 3) Usar técnicas de decomposição na estimativa;
- 4) Adotar modelos empíricos na estimativa.

Embora atrativa, a opção de adiar a estimativa não é prática, tendo em vista que o esforço e o custo devem ser conhecidos antes de iniciar qualquer atividade do processo de software, e, com relação a correção de bugs, não seria diferente. Logo, esta opção pode ser adotada apenas como uma estratégia para melhorar a precisão das estimativas de esforço e custo, recalculando-as de forma mais precisa conforme as atividades progredirem. Observe que esta estratégia se alinha com a teoria do *Cone da Incerteza*, introduzida por Barry Boehm em 1981 e amplamente mencionada na literatura, por exemplo (COHN, 2006), que reforça uma maior imprecisão das estimativas nas fases iniciais dos projetos de software. Como ilustrado na Figura 7, ao passo que o processo de software progride, produzindo seus diferentes artefatos (da esquerda para a direita no eixo horizontal), as incertezas vão diminuindo, e, assim, a precisão das estimativas vai melhorando cada vez mais (convergência no eixo vertical entre as curvas acima e abaixo do ponto x , que representam a evolução do custo estimado e real do projeto de software, respectivamente).

Figura 7 - Cone da incerteza



Fonte: Cohn (2006).

A opção de basear a estimativa em projetos anteriores pode ser razoavelmente precisa, mas apenas nas situações em que o projeto atual possui fatores de influência similares aqueles dos projetos anteriormente concluídos. Considerando a correção de bugs, que ocorre em uma fase avançada do projeto de software, e, em adição, guarda alto nível de similaridade com as atividades de desenvolvimento do próprio projeto de software, o nível de incerteza na estimativa

de esforço das tarefas de correção de bugs é possivelmente baixo (COHN, 2006), o que pode resultar em menor complexidade e melhor precisão na estimativa de esforço, pois a maioria dos indicadores adotados para estimar o esforço já podem ter sido coletados ao longo da fase de desenvolvimento do sistema. No entanto, neste caso, há um possível risco associado, pois, se o projeto de software não é bem gerenciado, os indicadores coletados podem estar desatualizados e inconsistentes.

A opção de usar técnicas de decomposição explora a abordagem denominada *dividir-para-conquistar*, amplamente adotada em várias atividades da engenharia de software. Neste sentido, processos são decompostos em fases, que, por sua vez, são divididos em atividades, e, assim sucessivamente, até alcançar uma granularidade adequada, na qual podem ser gerenciadas e realizadas de forma passo-a-passo. No caso específico de correção de bugs, por exemplo, a estimativa baseada em decomposição pode considerar o esforço requerido para analisar o código fonte, modificar o código fonte, e, posteriormente, realizar testes de validação do novo código gerado.

Por fim, a opção de adotar modelos empíricos consiste na formalização matemática ou processual de experiências vividas, onde dados históricos são coletados de forma sistemática em diferentes projetos de software, e, posteriormente, analisados para formulação do modelo de estimativa em questão. Observe que a opção de estimativa baseada em modelos empíricos representa uma generalização da opção de estimativa baseada em projetos similares anteriores.

Considerando essa gama de opções, métodos automáticos de estimativa podem explorar uma ou várias dessas opções para identificar o esforço e o custo das tarefas de correção de bugs. Por exemplo, Thung (2016) propõe um método de predição automática de estimativa de esforço para correção de bugs que é fundamentado na análise da quantidade de linhas de código que devem ser adicionadas, modificadas e removidas.

Na literatura, existem várias propostas de métodos de estimativa de esforço, sendo cada um deles apropriado para um dado contexto ou processo de desenvolvimento. Segundo Sousa (2011), também é interessante utilizar mais de um método para estimar o esforço de uma tarefa, pois uma segunda estimativa, além de poder confirmar a corretude e precisão da primeira, pode também contestá-la e, nesse caso, a possibilidade de erros no uso de um método pode ser minimizada. Portanto, diante de uma possível contestação da estimativa, e, conseqüentemente, uma análise mais aprofundada na tarefa de correção de um bug, torna-se possível chegar a um consenso em relação a estimativa de esforço requerido para efetuá-la, aumentando a probabilidade da estimativa ficar mais próxima da realidade.

Dentre os métodos empíricos propostos pela academia e indústria para estimar esforço e/ou custo de correção de bugs, as técnicas propostas em (THUNG, 2016) e (WEISS et al., 2007) são brevemente apresentadas a seguir apenas a título de exemplificação.

Thung (2016) propõe uma abordagem para predição automática do esforço para correção de bugs em termos da quantidade de linhas de código. Através do exame do relatório de bug, a proposta estima se o esforço para correção é alto ou baixo. O modelo funciona basicamente da seguinte forma, em duas fases, uma de treinamento e outra de implantação. Na fase de treinamento, um módulo extrator de características converte o histórico de relatórios de bugs anteriormente resolvidos em vetores de características, para representá-los dessa forma. Em seguida, o extrator quantifica o esforço de correção de cada bug, analisando no repositório de código fonte as mudanças ocorridas durante as tarefas de correção destes bugs. Posteriormente, para cada bug, um módulo construtor combina cada vetor de características com seu esforço correspondente para formar um vetor de características rotulada. Em seguida, empregando o algoritmo denominado Support Vector Machine (SVM), o construtor aprende todos os vetores de características rotuladas na fase de treinamento. Por fim, na fase de implantação, a abordagem estima o esforço necessário para corrigir um novo bug, usando como base os vetores de características rotuladas aprendidos na fase de treinamento. Para tal, o extrator converte o relatório do novo bug em seu vetor de características correspondente, e, em seguida, o módulo preditor de esforço usa esse vetor de características e prevê o esforço necessário para corrigir o bug com base no modelo gerado na fase de treinamento.

Weiss et al. (2007) apresenta uma proposta que tenta estimar o tempo necessário para corrigir um problema que pode ser um bug, uma solicitação de funcionalidade ou uma tarefa. Como exemplo de aplicação da proposta, os autores mostram um estudo de caso que extrai informações de relatórios de bugs antigos para predizer o esforço necessário para correção de novos bugs. O modelo é bem simples, a primeira fase é a inserção de um novo relatório no repositório de bugs; na fase dois é realizada uma busca por relatórios bugs existentes e semelhantes ao bug recém cadastrado, efetuando um cálculo de similaridade, com limitação mínima desta similaridade, e utilizou para isso, o Lucene que é uma ferramenta de medição de similaridade de texto, e os campos dados como entrada para avaliar a semelhança foram o título e a descrição; na fase três após os relatórios semelhantes serem encontrados, é realizada uma média ponderada do esforço realizado nos relatórios semelhantes e o modelo fornece uma predição de esforço para o novo bug.

Ainda merecem destaque duas técnicas de estimativa em desenvolvimento de software que podem ser utilizadas no cálculo do esforço para correção de bugs: a técnica baseada no julgamento de um especialista e a técnica por analogia. É importante ressaltar que a técnica de julgamento de um especialista foi citada no trabalho de Xiao e Afzal (2010), quando indicou em sua pesquisa, particularmente no seu estudo de caso, que a organização analisada, a qual tem seus processos de trabalhos certificados no nível 3 no *Capability Maturity Model Integration (CMMI)* (Modelo de Maturidade em Capacitação Integração), utiliza como estimativa de esforço, o julgamento de especialistas.

Segundo Lee et al., (1998), apud Pagno, (2010), a técnica baseada no julgamento de especialistas divide o desenvolvimento do aplicativo de software em tarefas utilizando uma granularidade adequada, isto é, dividindo tarefas grandes, de longa duração ou complexas, em menores e utilizando faixas de estimativas com melhor e pior caso (dividir para conquistar). Como mensurar tarefas de pequena duração é mais fácil, ao unir novamente todas as tarefas tem-se o total gasto de tempo estimado.

Esta técnica consulta vários especialistas em desenvolvimento de software e no domínio da aplicação. Posteriormente, os resultados de suas respectivas estimativas são comparados e discutidos até que se chegue a uma estimativa comum.

Na técnica por analogia, segundo Lederer et al. (1995) apud Pagno (2010), o princípio adotado é a comparação de informações obtidas de atividades finalizadas com as primeiras estimativas do projeto e ajusta os valores estimados para as futuras tarefas (que poderiam ser as correções). Este é um tipo de técnica que combinada com a técnica de um especialista pode se adaptar facilmente na utilização de estimativa de custo para correção de bugs, pois é comum estimar o custo de um novo projeto com base em projetos já concluídos. Mas se esta técnica se basear em um projeto que não foi bem documentado e estimado ela pode gerar grandes erros.

2.2 ENGENHARIA DE SOFTWARE BASEADA EM BUSCA

A SBSE surgiu com o propósito de promover trabalhos científicos que adotam algoritmos de busca para tratar questões na área de engenharia de software (FERREIRA et al. 2016), através de técnicas matemáticas empregadas em diversas naturezas de problema em todo ciclo de vida da engenharia de software (HARMAN et al. 2012) como: ferramentas e técnicas de codificação, ferramentas e técnicas de design e requisitos/especificações de software (FERREIRA et al, 2016), até a manutenção e a reengenharia (HARMAN et al. 2012). Ainda de

acordo com HARMAN et al. (2012), é comum encontrar o termo SBSE em qualquer forma de otimização em que o domínio do problema venha de engenharia de software.

Neste contexto, segundo Harman et al. (2012) a SBSE referencia técnicas metaheurísticas como forma de resolver problemas de busca baseada em otimização – *Search-Based Optimization (SBO)* E o termo “*Search*” é usado para se referir à metaheurística que vai ser utilizada para reformular problemas de Engenharia de Software – *Software Engineering (SE)*. Deste modo, esta técnica promete fornecer resultados balanceando o tempo (custo de processamento) e a qualidade das soluções.

Como este trabalho trata de um problema para selecionar um conjunto de bugs que devem ser corrigidos, ou seja, de um problema que pode conter várias combinações para encontrar uma boa solução e se caracteriza por ser um problema de busca baseada em otimização é plausível utilizar este tipo de técnica para resolver este problema de seleção de bugs.

Complementando a ideia do parágrafo anterior, entende-se que, as metaheurísticas são métodos da SBSE abordados em problemas de otimização combinatória com objetivo de buscar boas soluções em tempo de execução, utilizando ainda uma função objetivo baseada no problema para avaliar o resultado da busca. Por exemplo, para Melián et al. (2003) os métodos heurísticos são métodos de busca para resolver um problema de forma inteligente que não garante uma solução ótima ou melhor solução em relação a busca exaustiva, mas pode trazer boas soluções em um tempo rápido. E para Burity e Elias (2011), uma metaheurística é um conjunto de conceitos que são usados para definir métodos heurísticos que podem ser aplicados a uma grande variedade de problemas e o problema de agendamento, priorização e/ou seleção está dentro do escopo dos problemas que os métodos heurísticos podem tratar.

A necessidade da utilização de uma busca baseada em metaheurística se dá porque a população de bugs pode crescer de forma acelerada, fazendo com que a seleção dos bugs seja inviável em tempo de execução. Assim, segundo Netto et al. (2016), esse tipo de problema é caracterizado como NP-Difícil, pois a quantidade de combinações para construir todas as soluções do problema cresce de forma exponencial, dada a quantidade de bugs (que aumentam de forma acelerada) e os diversos critérios de seleção considerados. Sendo assim, este problema deve ser resolvido como um problema de otimização, pois se assemelha ao problema do próximo *release* que é um problema também do tipo NP-Difícil de tempo não polinomial que deve ser resolvido em tempo polinomial.

As vezes os métodos heurísticos podem ficar limitados em ótimos locais, essa é uma das principais desvantagens das buscas locais (MELIÁN et al. 2003), ou seja, é quando o algoritmo, encontra uma melhor solução em uma determinada relação de vizinhança que pode está distante de melhores soluções, implicando em uma solução de baixa qualidade.

Segundo Melián et al. (2003), uma busca com uma perspectiva global do espaço de soluções deve procurar ferramentas para escapar destas situações, e para isso as principais metaheurísticas de busca global emerge das três principais abordagens para escapar de um ótimo local:

- retornar a iniciar a pesquisa a partir de outra solução inicial, ou seja, reiniciar o processo de busca a partir de soluções candidatas iniciais diferentes;
- modificar a estrutura dos ambientes, ou mesmo aumentar o espaço de busca;
- permitir o agravamento dos movimentos da solução atual, ou seja, permitir alguns movimentos para soluções de qualidade inferior à solução corrente para chegar em uma solução melhor.

Por fim, a busca global se caracteriza por explorar um conjunto de todas as soluções possíveis em um determinado espaço de busca.

Acrescenta-se ainda, sobre a SBSE, que esse tipo de abordagem é atraente porque oferece um conjunto de soluções automatizadas e semi-automatizadas adaptativas em situações tipificadas por grandes e complexos espaços de problemas com múltiplos objetivos concorrentes e conflitantes (HARMAN et al., 2012). Para resolução destes problemas, existe uma ampla gama de diferentes técnicas de otimização e busca (metaheurísticas) que podem e têm sido usadas em diversos trabalhos. Conforme HARMAN et al. (2012), as mais utilizadas são: *Simulated Annealing (SA)*, *Genetic Algorithms (GAs)*, *Programação Genética (GP)* e *Hill Climbing (HC)*. Em especial a nossa abordagem irá adotar o Algoritmo Genético tendo em vista ser aderente ao tipo de problema abordado.

A próxima etapa então é definir uma metaheurística que seja mais apropriada para o tipo de proposta que está sendo abordada neste trabalho.

2.2.1 ALGORITMO GENÉTICO

O Algoritmo Genético (AG) é uma técnica de busca apresentada por Holland (1975) e inspirada na evolução biológica das espécies e no paradigma da seleção natural. Segundo Dreio (2005), verificou-se que nos últimos anos o número de trabalhos na área de pesquisa operacional associados ao uso de algoritmos genéticos cresceu expressivamente. O motivo para

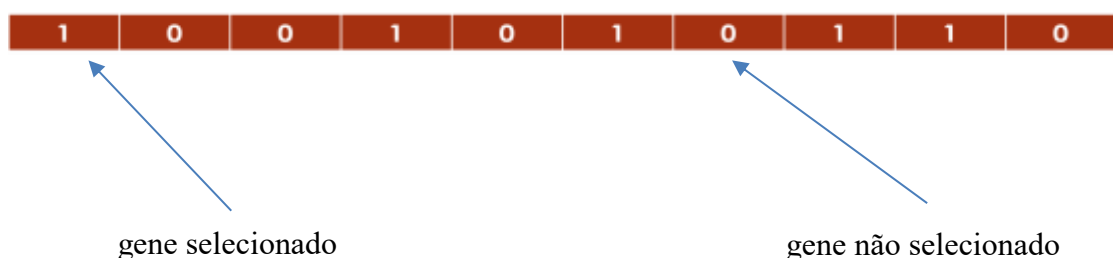
isso se dá pela flexibilização em tratar de vários problemas e trazer bons resultados. Por esta razão e pela aderência ao problema em estudo em permitir a representação de forma adequada sem perdas de dados, a proposta deste trabalho utiliza o AG como técnica de busca, não descartando a possibilidade no uso de outras metaheurísticas.

O AG é uma técnica de busca de otimização global porque explora pontos inteiramente novos dentro de um espaço de busca e possui características evolutivas ao longo das iterações porque favorece a permanência e evolução de indivíduos mais aptos ao ambiente dando a eles mais chances de sobrevivência.

Segundo Filitto (2008), o AG é iniciado através de um conjunto de soluções aleatórias e busca uma melhor solução para o seu problema, combinando os melhores representantes desta população e obtém uma nova solução que passa a substituir a anterior, ou seja, tem por objetivo gerar de forma gradativa, ao longo de diversas iterações, soluções cada vez melhores.

Existem várias formas de representação de um indivíduo ou cromossomo da população no AG, e esta representação varia conforme o tipo de problema. Para representação de um cromossomo pode-se utilizar um conjunto de *bits*, ou também uma variável numérica contínua que armazena o seu próprio valor (FILITTO, 2008). Neste caso, será demonstrado um exemplo utilizado neste trabalho em que um indivíduo no AG, ou seja, uma solução candidata ao problema, é representada por um cromossomo que por sua vez possui um conjunto de genes ou *bits*. Sendo assim, um cromossomo, solução candidata ou indivíduo é representado, neste exemplo, por uma combinação binária de 0s e 1s, conforme ilustrado na Figura 8.

Figura 8 - Representação de um cromossomo



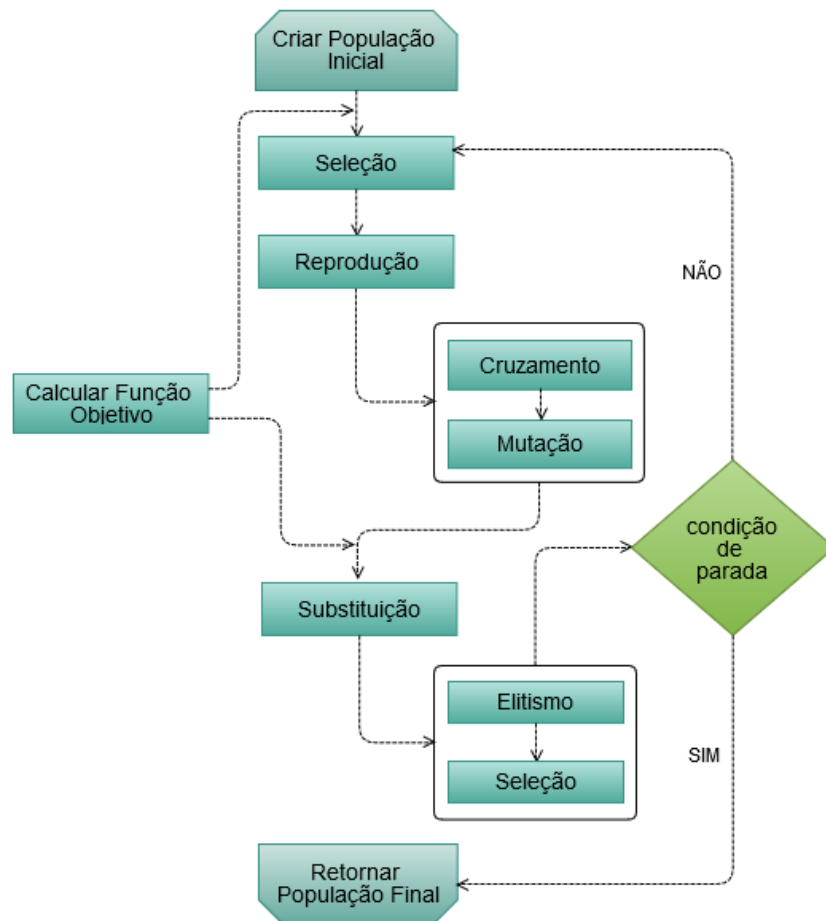
A utilização da codificação binária (Figura 8) para representar os genes dos cromossomos se caracteriza por ser uma seleção rápida e mais simples segundo Ribeiro (2013), além de ser um tipo de representação mais comum e mais adequada para problemas de seleção pela representação de dois valores que indicam a seleção e não-seleção de um gene. Esse tipo de representação binária não acarreta em nenhum tipo de perda de informação.

Após definida a representação de um cromossomo, que corresponde a uma solução candidata, é necessário inferir um valor de avaliação para mensurar a sua qualidade ou adequabilidade ao problema que deve ser resolvido, pois só assim as soluções podem ser comparadas. Este valor é representado pela função objetivo, *fitness*, ou ainda, função de avaliação. Esta função, segundo Linden (2008), deve conter parâmetros que sejam inerentes ao problema para melhor refletir a realidade da solução. Linden acrescenta que há casos em que a solução candidata deve ser penalizada, ao descumprir alguma restrição ou a um custo de reparo da solução. Esta penalidade pode ter, por exemplo, uma natureza estática, ou seja, uma penalidade para cada faixa de violação de cada restrição incidindo no valor final da função objetivo. Por fim, Linden percebe a possível necessidade de algumas tentativas, para encontrar a equação ideal da função objetivo.

Dada a representação de um cromossomo e definida a função de avaliação das soluções, é necessário compreender a execução do algoritmo genético e definir as técnicas utilizadas no seu fluxo de execução, nas fases de seleção, reprodução (recombinação ou *crossover*) e substituição do algoritmo (HOLLAND, 2005 apud FILITTO, 2008).

Desta forma, apresenta-se na Figura 9 uma exemplificação da execução comum de um algoritmo genético. O fluxo é iniciado com a criação da população inicial, depois é necessário o cálculo da função objetivo de todos os indivíduos e segue com o fluxo na fase de *seleção*, onde é escolhido os pais que participarão do processo de reprodução. Na fase de *reprodução* são efetuados dois operadores: o cruzamento para geração dos filhos e a mutação dos filhos gerados. Após esta fase, é necessário novamente calcular o valor da função objetivo de todos os filhos. Para a etapa seguinte, conforme a Figura 9, é realizada a substituição da população anterior a partir dos indivíduos existentes e dos filhos gerados, com aplicação de uma estratégia chamada elitismo para manter a população elite para próxima geração, que são os melhores indivíduos, e o restante da população que coexistirá pode ser completada através de uma nova seleção, dando condições de evolução da população, permitindo o desenvolvimento de indivíduos recém criados que não estão na população elite. Por fim, nesta fase se o algoritmo não satisfazer o critério de parada, nesse momento haverá um novo ciclo de execução.

Figura 9 – Fluxograma de Execução do Algoritmo Genético



No parágrafo anterior, foi analisado a execução geral do fluxo do algoritmo genético, e em sua execução foram citadas algumas técnicas, dentre elas estão a seleção, a reprodução, o elitismo e a substituição. Por conseguinte, serão descritos alguns exemplos destas técnicas que podem ser utilizadas.

a) Seleção

Para Filitto (2008), a função da seleção em um AG consiste na estratégia para escolher os indivíduos da população atual que participarão da fase de reprodução, isto é, selecionar os indivíduos (pais) que vão gerar os novos indivíduos (filhos) para compor a próxima geração da população. Esta escolha deve ser feita de tal forma que os membros da população com maiores chances de reprodução, sejam aqueles que apresentam um valor da função objetivo mais adequado. Dentre os vários tipos de técnicas de seleção pode-se destacar: seleção por torneio, seleção por roleta, e seleção por ordenação.

No método de seleção por torneio, segundo Araki e Vergilio (2010), um número p de indivíduos da população é escolhido aleatoriamente para formar uma subpopulação temporária.

Deste grupo, é selecionado o melhor indivíduo. Este tipo de seleção se caracteriza por conduzir a uma forte pressão seletiva através da competição direta pelo direito de ser pai. Por exemplo, em uma população com 100 indivíduos, seriam escolhidos 50 pais para participar da etapa de reprodução. A escolha de cada pai se dá através de dois pais selecionados aleatoriamente, e, posteriormente, aquele que tem a melhor aptidão com base na função objetivo é escolhido para participar da etapa de reprodução.

O processo de seleção por roleta, segundo Araki e Vergilio (2010), especifica a probabilidade de que cada indivíduo seja selecionado para a próxima geração. Cada indivíduo da população recebe uma porção da roleta proporcional a sua probabilidade. Assim, diferentemente da seleção por torneio, a seleção por roleta dá oportunidade também para os indivíduos com menos aptidões, de forma que a probabilidade de seleção do indivíduo se dá pela relação entre seu índice de aptidão comparado a aptidão de todos os outros indivíduos. Por exemplo, considerando três indivíduos, que possuem valores da função *fitness* iguais a 10, 30 e 60, as probabilidades de seleção destes indivíduos é 0.1, 0.3 e 0.6, respectivamente. Esse tipo de seleção parte do pressuposto de não eliminar a seleção de indivíduos menos aptos em uma comparação através da função de avaliação.

A *seleção por ordenação* consiste em escolher os melhores indivíduos baseado no resultado da ordenação dos valores de suas funções de *fitness* (do maior para o menor). Compreende-se que esse tipo de seleção pode ter a tendência de resultar em pouca diversidade, já que seleciona apenas os melhores indivíduos, e pode por consequência fazer com que o resultado da busca fique limitado a um ótimo local.

A próxima fase utiliza operadores de cruzamento e mutação e esta fase equilibra dois objetivos aparentemente conflitantes: *o aproveitamento das melhores soluções* e *a exploração do espaço de busca*. O processo de busca é, portanto, multidimensional, preservando soluções candidatas e provocando a troca de informação entre as soluções exploradas (MICHALEWICZ, 1996; VON ZUBEN, 2000) apud (GABRIEL e DELBEM, 2008).

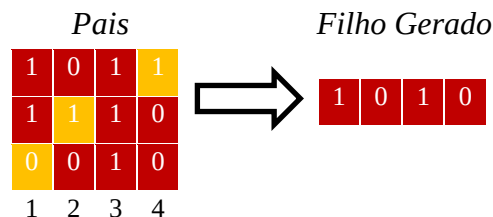
b) Cruzamento

Após a fase de seleção no AG, a próxima etapa a ser executada é o cruzamento dos pais para geração dos filhos. Na literatura podem ser encontrados diversos tipos de cruzamento e aqui serão exemplificados alguns.

Segundo Filitto (2008), o operador de cruzamento se baseia na combinação dos cromossomos dos pais para gerar os cromossomos dos filhos. Existem alguns tipos de cruzamentos como baseados em maioria, cruzamento de um ponto, uniforme, etc.

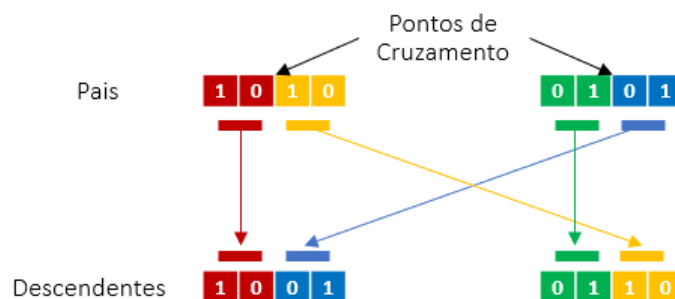
Na técnica de *cruzamento baseada em maioria*, considerando uma certa quantidade ímpar de pais, esta técnica seleciona para cada gene o valor que mais aparece nos pais em questão. Por exemplo, a Figura 10 demonstra um exemplo de cruzamento por maioria. Neste exemplo, ao observar os genes, em destaque de vermelho, dos 3 pais a maioria do valor que aparece para o primeiro e terceiro gene dos pais é um, e, portanto, o filho tem o valor 1 nos genes da posição 1 e 3. Similarmente, a maioria do valor que aparece do gene 2 e 4 para os três pais é 0, e, conseqüentemente, o filho tem o valor 0 nestes genes.

Figura 10 - Cruzamento baseado em maioria



O *cruzamento em um ponto* funciona através de um corte binário dos pais em uma posição aleatória e recombina as partes geradas (Figura 11). Ou seja, consiste em dividir os cromossomos selecionados em um ponto de sua cadeia, onde este ponto é escolhido aleatoriamente. Após isso, copiamos uma parte dos cromossomos de cada pai para gerar os cromossomos dos novos filhos (FILITTO, 2008).

Figura 11 - Cruzamento em um ponto



Fonte: Adaptação Filitto (2008).

No *cruzamento uniforme*, também denominado *cruzamento por máscara*, para Syswerda (1991) apud Filitto (2008) (Figura 12), esta técnica consiste em gerar cada gene do descendente copiando o correspondente gene de um dos pais, e este gene é escolhido de acordo com uma “máscara de cruzamento” gerada aleatoriamente. No processo de criação do

cromossomo filho, percorrem-se todas as posições da máscara analisando seus valores. Quando o valor da posição é 1, o gene do primeiro pai referente a mesma posição da máscara é copiado para o filho. Caso o valor da máscara seja 0, será copiado o gene do segundo pai.

Figura 12 - Cruzamento uniforme

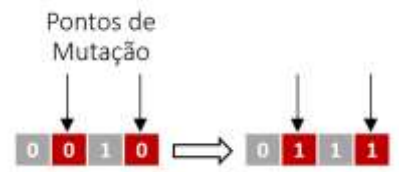
Máscara de Cruzamento	0 1 0 1
Primeiro Pai	1 0 1 0
Segundo Pai	0 1 1 0
Descendente	0 0 1 0

Fonte: Adaptação Filitto (2008).

Após o cruzamento, tem a mutação que ocorre no final da reprodução com a finalidade dos indivíduos mutantes trazerem diversidade à população ao longo das sucessivas gerações. Segundo Bean (1994) apud Netto et al. (2010), para evitar que a geração de indivíduos não convirja para um ótimo local, o operador de mutação realiza a substituição de alguns genes dos filhos, provendo maior variabilidade genética à população e normalmente na literatura a taxa de mutação deve ser em torno de 2% a 7%. Konak et al. (2006) apud Lopes et al. (2013), também ratifica que a mutação reintroduz a diversidade genética de volta à população e auxilia o algoritmo a fugir de máximos ou mínimos locais.

Segue alguns métodos que podem ser usados no algoritmo genético: mutação binária simples ou uniforme, mutação por permutação ou *swap* e mutação *creep*.

Segundo Mitchell (1996) apud Lopes et al. (2013), o operador mais utilizado é a *mutação binária simples*. Para ele, posições dos cromossomos do indivíduo são sorteadas e os genes correspondentes são invertidos, ou seja, se o valor do gene for 1, então se muda este valor para 0, e vice-versa. A Figura 13 ilustra o funcionamento deste operador. Também chamada de mutação *uniforme*. De acordo com Bäck et al. (2000) apud Gabriel e Delbem (2008), este operador seleciona aleatoriamente um componente $k \in \{1, 2, \dots, n\}$ do cromossomo $x = [x_1, \dots, x_k, \dots, x_n]$ e gera um indivíduo $x' = [x_1, \dots, x'_k, \dots, x_n]$, onde x'_k é um número aleatório no intervalo $[L_l, L_s]$, sendo L_l, L_s , respectivamente, os limites inferior e superior para o valor do alelo x_k .

Figura 13 - Muta    bin  ria simples

Fonte: Adapta    Lopes et al. (2013).

Segundo Eiden e Smith, (2003), a muta    *swap* apenas faz a troca de duas posi   es escolhidas randomicamente, ou invertendo um segmento escolhido aleatoriamente. A Figura 14 ilustra a muta    *swap*.

Figura 14 - Muta    Swap

A muta    *creep* por Barros et al. (2011), consiste em adicionar ou subtrair um valor a um gene, de outra forma, Gabriel, em 2010, diz que este operador soma-se um valor pequeno (positivo ou negativo) a um valor de um determinado gene e Morales, (2009), exemplifica quando o cromossomo sofre a muta    *creep*, conforme a Figura 15, para Morales, um bit escolhido para muta    troca com seu vizinho de valor. Assim se na posi    i do indiv  duo tem-se um valor de 1 e na posi    $l-i$ um valor de zero, o novo indiv  duo ter   na posi    i um valor de zero e na posi    $l-i$ ter   1.

Figura 15 - Muta    Creep

c) Substitui   

A fase de substitui    seria a sele     dos sobreviventes para pr  xima gera    . A estrat  gia nesta fase   guardar os melhores indiv  duos para que n  o se perca o valor m  ximo ou m  nimo j   alcan  ado da fun     objetivo e parte do restante da popula    , entre pais e filhos gerados na itera     vigente, pode ser mantida atrav  s de uma nova sele     dando possibilidade de indiv  duos menos aptos, ainda em desenvolvimento, trazer novos espa  os de busca ou melhores resultados em rela     aos j   alcan  ados.

Para garantir que os melhores indiv  duos n  o sejam perdidos, o AG pode utilizar a t  cnica de *elitismo*, que foi primeiramente introduzida por De Jong (1975), cujo papel  

melhorar ainda mais as próximas gerações, guardando os melhores indivíduos de uma geração para outra e consequentemente fazendo uma pressão que converge para uma população homogênea e, por esse fato, é interessante utilizar uma pequena porção dos melhores indivíduos da população gerada, de forma que não prejudique a diversificação das novas populações, como por exemplo de 5% a 10% da população atual. Quando se está trabalhando com uma população para melhorá-la é interessante que se tenha uma percentagem bem equilibrada de indivíduos com diferentes características. No entanto, para manter o que já conseguiu de melhoria, esta porcentagem não deve ser muito baixa com o prejuízo de perder as melhores soluções encontradas, nem deve ser muito alta para prejudicar a diversidade e ficar preso a um grupo de indivíduos semelhantes. Por isso, para complementar esta técnica e para completar a população pode-se utilizar uma técnica de migração de forma que indivíduos menos aptos sejam selecionados aleatoriamente. Esta última técnica permite uma população mais diversificada, e uma maior exploração no espaço de busca.

Todo o processo do AG é repetido até que uma condição de parada seja verdadeira e no final o algoritmo deve gerar um bom conjunto de soluções que serão representadas pela função objetivo. E em geral, a condição de parada pode ser definida por um determinado número total de iterações. Mas, opcionalmente, outra condição de parada pode ser um dado número de iterações sucessivas sem melhoria significativa das soluções.

2.3 CONSIDERAÇÕES FINAIS

Este capítulo abordou diversos conhecimentos correlacionados a este trabalho. Primeiramente, definindo em que fase do ciclo de vida o problema de seleção de bugs está inserido. Neste capítulo, também foi identificada a necessidade de considerar diversos critérios e restrições para o referido problema, baseando-se nas escolhas encontradas na literatura. Outro ponto importante destacado no capítulo é a caracterização da área em que a abordagem proposta está inserida, bem como, as técnicas que podem ser utilizadas para sua resolução.

Quanto as fases do ciclo de vida de um sistema de software, verificou-se que o problema de seleção de bugs inicia-se imediatamente após um novo relatório de bug ser gerado e antes do mesmo ser alocado, ou seja, no momento em que é realizada a análise do bug para sua alocação.

Foram verificados na literatura diversos meios que podem ser adotados para o gerenciamento de comunicação entre os clientes finais (equipes de TI das organizações) e

produtor, através de sistemas de rastreamento de bugs, bem como, foi realizada uma citação do sistema utilizado na UFPB que é o *REDMINE*.

Observou-se que os critérios de prioridade e severidade, apesar de serem bastante discutidos na literatura, possuem uma variabilidade de conceitos, mas aqui foram adotados os conceitos de importância do bug e impacto do bug para o sistema, respectivamente. Ainda neste sentido, sabe-se que já existem trabalhos aplicando técnicas em que estes critérios podem ser atribuídos de forma automática.

O problema da seleção de bugs para correção só existe porque não é possível, em um dado *release*, a incorporação de todos os bugs. Neste sentido, a restrição orçamentária é considerada, de forma que a informação do custo ou esforço pode ser utilizada em modelos de seleção de bugs, como forma de restringir a quantidade de bugs e esforço que será necessário para entregar a próxima versão do software.

Por fim, para resolver o problema de seleção de bugs, parece razoável o emprego de técnicas de otimização combinatórias da SBSE, mais especificamente o emprego do algoritmo genético, que trata de problemas com características diferentes e conflitantes e permite a entrega de soluções com uma boa qualidade e custo de processamento.

3 TRABALHOS RELACIONADOS

Este capítulo tem por objetivo descrever e comparar alguns trabalhos relacionados encontrados na literatura, que abordam os dois principais temas desta dissertação: *Seleção de Bugs* e *Engenharia de Software Baseada em Buscas* (SBSE). A análise comparativa realizada destaca as semelhanças e diferenças entre os trabalhos avaliados e a abordagem proposta nesta dissertação, e, de forma mais relevante, evidencia as contribuições da abordagem proposta.

As técnicas SBSE vêm sendo aplicadas com sucesso em várias atividades ao longo do ciclo de vida do processo de software, incluindo engenharia de requisitos, planejamento de projetos, avaliação de qualidade, testes e manutenção (HARMAN, 2007). Mais recentemente, alguns trabalhos relacionados ao processo de correção de bugs foram propostos, sugerindo o uso de técnicas de otimização aplicadas aos relatórios de bugs para extrair informações valiosas ou realizar a automatização de algumas atividades associadas ao ciclo de vida dos bugs, que, de forma mais detalhada, podem ser classificadas em atividades de seleção, priorização, alocação e agendamento de bugs.

Como já mencionado, o problema de seleção de bugs tem por objetivo selecionar um conjunto de bugs para correção. De forma complementar, além de selecionar os bugs para correção, o problema de priorização de bugs também recomenda a ordem ou sequência com que os bugs selecionados devem ser corrigidos. Portanto, o problema de priorização de bugs incorpora aspectos do problema de seleção de bugs.

De forma similar, em geral, os problemas de alocação e agendamento de bugs também incorporam aspectos dos problemas de seleção e priorização de bugs. Em outras palavras, a alocação de bugs pode ser vista como um problema complementar de seleção ou priorização de bugs, que tem por objetivo selecionar os bugs para correção, e, de forma complementar, atribuir as tarefas de correção dos bugs selecionados aos membros da equipe de desenvolvimento. Similarmente, o agendamento de bugs também pode ser visto como um problema complementar da seleção ou priorização de bugs, que tem por objetivo selecionar os bugs para correção, e, de forma complementar, recomendar o cronograma ou agenda de correção dos bugs selecionados. Portanto, como pode ser observado, os problemas de seleção, priorização, alocação e agendamento de bugs são complementares, e, ao mesmo tempo, podem ser integrados em diferentes combinações de suas partes ou no todo.

As abordagens existentes na literatura que tratam o problema de seleção de bugs são os trabalhos mais relacionados à abordagem proposta nesta dissertação, cuja análise comparativa

leva ao melhor entendimento dos desafios do problema e fornece subsídios para encontrar as lacunas ainda não tratadas, delineando assim oportunidades para evolução e contribuição. No entanto, considerando os aspectos complementares das atividades de seleção, priorização, alocação e agendamento de bugs, a análise comparativa aqui apresentada inclui trabalhos relacionados que cobrem quaisquer destas atividades de forma isolada ou integrada.

3.1 PRIORIZAÇÃO DE BUGS NO REPOSITÓRIO KATE EDITOR

Dreyton et al. (2015) investigam uma abordagem automatizada para priorização de bugs em repositórios de código aberto online. A proposta de Dreyton é baseada em informações identificadas e extraídas do repositório *Kate Editor*, tais como a prioridade e a severidade dos bugs. Como diferencial, a proposta de Dreyton adota o conceito de votação, onde a comunidade de desenvolvimento de código aberto pode apresentar votos para indicar os bugs de maior relevância na perspectiva dos participantes da referida comunidade. Considerando métricas de *relevância*, *importância* e *severidade*, calculadas respectivamente a partir das informações de entrada relativas à votação, prioridade e severidade dos bugs, a proposta de Dreyton adota uma função de avaliação mono-objetiva ponderada que avalia as soluções candidatas, representadas na forma de uma lista ordenada de bugs para correção.

Por um lado, na abordagem de Dreyton, a quantidade de bugs a priorizar é especificada diretamente como um parâmetro de entrada pré-configurado. Por outro lado, na abordagem proposta nesta dissertação, diferentemente, a quantidade de bugs a serem selecionados não é inicialmente pré-determinada, mas recomendada automática e indiretamente com base na restrição de orçamento do projeto de software e no custo de correção dos bugs, permitindo que diferentes soluções candidatas possam ter diferentes quantidades de bugs. Da forma aqui proposta, a estratégia é mais prática, flexível e eficaz, não implicando em esforço adicional para indicar de forma *ad hoc* uma quantidade de bugs apropriada ao orçamento do projeto. Em outras palavras, a tarefa de estimar antecipadamente a quantidade de bugs para correção exige a análise de diferentes aspectos interdependentes, tais como o orçamento do projeto e o custo de correção dos bugs, o que é uma tarefa bastante complexa, demorada e propensa a erros, principalmente quando baseada apenas no julgamento do gerente de projetos, dada a enorme quantidade de possibilidades de soluções onde o balanceamento entre quantidade de bugs, orçamento e custo de correção deve ser avaliado. Portanto, a proposta de Dreyton tem como principal limitação a exclusão de informações sobre orçamento do projeto e custo de correção dos bugs.

No que diz respeito às dependências de bugs, a proposta de Dreyton define uma restrição que impõe às soluções candidatas a necessidade de atender apenas a relação de precedência entre bugs, modelando de forma limitada as dependências de bugs como um grafo direcionado acíclico. De uma forma mais interessante, a abordagem proposta nesta dissertação adota um conjunto de operadores matriciais que derivam soluções candidatas que atendem relações de precedência e concorrência entre bugs, modelando de forma mais realista as dependências de bugs como um grafo direcionado cíclico ou acíclico.

Ainda em relação às dependências de bugs, a abordagem de Dreyton não adota ou propõe qualquer método para identificação das dependências de bugs, limitando-se a defini-la como uma variável de entrada do modelo. Diferentemente, a abordagem proposta nesta dissertação propõe um método baseado na arquitetura de projetos de software baseados em componentes para identificação das dependências de bugs. Ou seja, ao invés de adotar como variável de entrada as dependências de bugs, como uma contribuição adicional, a proposta desta dissertação propõe um método para identificação das dependências de bugs a partir das dependências de componentes que constituem a arquitetura do projeto de software.

Apesar das diferenças, percebe-se também alguns pontos similares entre as duas propostas. Por exemplo, ambas modelam o problema usando uma função de avaliação mono-objetiva ponderada. Além disso, ambas adotam a metaheurística de algoritmos genéticos para modelar e solucionar o problema. De forma similar, o conceito de votação também está presente em ambas propostas. No entanto, na proposta de Dreyton, a comunidade de desenvolvimento do sistema de software sinaliza através do voto a relevância dos bugs. Já na proposta desta dissertação, os clientes do sistema de software sinalizam através do voto a prioridade e a severidade dos bugs, permitindo a avaliação dos bugs com uma larga participação dos clientes para sugestões sobre a importância e o impacto dos bugs no negócio.

3.2 PRIORIZAÇÃO E ALOCAÇÃO MULTI-OBJETIVA DE BUGS

Tomando como base a abordagem proposta em (DREYTON et al., 2015), os autores estenderam a proposta inicial, e, na nova proposta apresentada em (DREYTON et al., 2016), reformularam matematicamente o problema de priorização de bugs como um problema multi-objetivo, onde as métricas de *relevância*, *importância* e *severidade*, presentes na proposta inicial na forma de uma função mono-objetiva ponderada, são decompostas em duas funções de avaliação independentes que avaliam cada solução candidata sob as perspectivas de *importância* e *risco*. Nesta reformulação, agora, a *importância* não é derivada apenas da

prioridade dos bugs, como era na proposta inicial, mas também considera a votação da comunidade de desenvolvimento em relação a relevância dos bugs. Ou seja, nesta nova proposta, a *relevância* e *importância* da proposta inicial são combinadas para derivar a *importância* na nova proposta. Em relação ao *risco*, há apenas uma renomeação, pois originalmente era a métrica de *severidade*.

Como forma de estender o escopo da abordagem, a nova proposta de Dreyton também inclui um novo estágio para recomendação de alocação de bugs a desenvolvedores, que consiste em selecionar uma solução que maximiza a compatibilidade entre a experiência do desenvolvedor em tarefas de correção de bugs no sistema de software em questão e o nível de experiência requerido para correção dos bugs relatados.

Embora contribua no escopo do problema de alocação de bugs, é importante destacar que o conceito de experiência adotado na proposta de Dreyton ainda é limitado, pois refere-se apenas a relação entre a quantidade de bugs resolvidos pelo desenvolvedor em diferentes áreas do sistema e a quantidade total de bugs resolvidos pelo desenvolvedor no sistema. Desta forma, é fácil perceber que desenvolvedores com experiências práticas similares podem ser erroneamente classificados como completamente diferentes. Por exemplo, considere dois desenvolvedores, onde cada um deles resolveu dois bugs em uma certa área do sistema, sendo que o primeiro resolveu 10 bugs e o segundo resolveu 100 bugs no total. No modelo de Dreyton, a experiência estimada do primeiro é 0.2 (2/10) enquanto a do segundo é 0.02 (2/100). No entanto, observe que, na prática, infere-se que ambos possuem experiências similares, ou, de forma mais refinada, o segundo tem experiência maior, pois a quantidade total de bugs resolvidos é bem mais expressiva.

Considerando que a abordagem proposta nesta dissertação não trata o problema de alocação de bugs, ou seja, não tem por objetivo atribuir as tarefas de correção dos bugs selecionados aos membros da equipe de desenvolvimento, observe que soa interessante estendê-la em um momento futuro para incorporar uma estratégia de alocação de bugs, possivelmente baseada ou até mesmo melhorada em relação a proposta de Dreyton, principalmente no que se refere a caracterização da experiência do desenvolvedor em tarefas de correção de bugs e o nível de experiência requerido para correção dos bugs.

Por fim, por ser modelada como um problema multi-objetivo, a proposta de Dreyton é avaliada comparativamente em relação a três metaheurísticas baseadas em algoritmos genéticos, a saber: NSGA-II, MOCell e IBEA. De forma complementar, a proposta de Dreyton também é avaliada em relação a competitividade humana, comparando os resultados obtidos

automaticamente pelas metaheurísticas e os resultados obtidos manualmente por cinco desenvolvedores.

É importante destacar que, por ser modelada como um problema mono-objetivo, a abordagem proposta nesta dissertação adota a metaheurística clássica de algoritmos genéticos, e, portanto, não podem ser aplicadas metaheurísticas multi-objetivas, tais como NSGA-II, MOCell e IBEA. No entanto, de forma relativamente simples, é possível em um momento futuro reformulá-la como um problema multi-objetivo, de modo que seja possível aplicá-las. No que se refere a competitividade humana, os resultados obtidos automaticamente pela abordagem aqui proposta são também comparados com os resultados obtidos manualmente por 11 desenvolvedores que compõem a equipe do projeto de software avaliado.

3.3 SELEÇÃO MULTI-OBJETIVA DE BUGS

De forma similar à abordagem proposta nesta dissertação, Andrade et al. (2011) apresentam uma abordagem para tratar o problema de seleção de bugs. Considerando métricas de *importância*, *severidade* e *tempo*, a proposta de Andrade adota uma abordagem multi-objetiva baseada em três funções de avaliação que avaliam as soluções candidatas, representadas na forma de um conjunto de bugs para correção, sem qualquer indicação da ordem ou sequência de correção dos mesmos. Portanto, de forma diferente das propostas de Dreyton, a proposta de Andrade não trata o problema de priorização de bugs.

Embora não adote explicitamente o conceito de votação, a proposta de Andrade considera que os múltiplos clientes do sistema de software em análise sinalizam suas preferências para cada bug, que são combinadas em uma função de avaliação para calcular a importância de uma dada solução candidata. Em relação ao conceito de severidade, a proposta de Andrade é um pouco confusa, pois menciona que a severidade representa um fator técnico da complexidade de correção do defeito, sem mencionar qualquer relação com o impacto do bug, que é a forma clássica adotada na literatura, e, também, na abordagem proposta nesta dissertação.

É importante frisar que a abordagem proposta nesta dissertação também adota um modelo baseado em múltiplos clientes, que, através de votação, sinalizam a prioridade e a severidade dos bugs, permitindo indicar não apenas a importância dos bugs, mas também o impacto destes bugs sob a perspectiva dos clientes. Portanto, comparativamente, a abordagem aqui proposta trata o problema de seleção de bugs de forma mais rica, refletindo de forma mais

realista e precisa os interesses e percepções tanto dos clientes quanto da organização responsável pelo desenvolvimento do sistema de software em questão.

Já em relação ao conceito de tempo, a proposta de Andrade é ainda mais confusa, uma vez que, inicialmente, menciona que representa o tempo necessário para corrigir um defeito, porém, logo em seguida, deixa o entendimento que representa o custo financeiro de correção do bug, sendo até adotado para avaliar se uma dada solução candidata possui custo total dentro do orçamento disponível no projeto.

Apesar da forma ambígua de tratar o conceito de tempo como sendo uma informação relacionada a esforço ou custo, de forma similar a abordagem aqui proposta, a proposta de Andrade também entende que é importante planejar as atividades de correção levando em consideração a restrição de tempo, recursos ou orçamento, o que implica na não correção de todos os bugs relatados. Dessa forma, é necessário selecionar determinados bugs de forma a garantir maior abrangência, maior importância, satisfação dos clientes, e otimizar o tempo gasto na atividade de correção de bugs (ANDRADE et al., 2011). Reforçando essa hipótese, segundo (CAETANO, 2007), conforme citado por Andrade et al. (2011), a seleção incorreta pode contribuir para maior insatisfação dos clientes e uso ineficiente dos recursos, inclusive tendo como efeito colateral direto a má utilização do esforço necessário em função do enfoque na correção de bugs menos prioritários para o cliente.

Outra similaridade entre a proposta de Andrade e a abordagem proposta nesta dissertação diz respeito à quantidade de bugs a selecionar para correção, que não é inicialmente pré-determinada como um parâmetro de entrada, mas recomendada automática e indiretamente pelas suas respectivas metaheurísticas. Como já discutido anteriormente, diferentemente, nas propostas de Dreyton, a quantidade de bugs é especificada como parâmetro de entrada pré-configurado, tornando-a custosa, inflexível e ineficaz, implicando em esforço adicional para indicar de forma *ad hoc* uma quantidade de bugs apropriada ao orçamento do projeto.

Em relação às dependências de bugs, da mesma forma que a proposta de Dreyton, a proposta de Andrade também define uma restrição que impõe às soluções candidatas a necessidade de atender apenas a relação de precedência entre bugs, modelando de forma limitada as dependências de bugs como um grafo direcionado acíclico. Como já sinalizado, explorando operadores matriciais, a abordagem aqui proposta deriva soluções candidatas que atendem relações de precedência e concorrência entre bugs, modelando de forma mais realista as dependências de bugs como um grafo direcionado cíclico ou acíclico.

Ainda em relação às dependências de bugs, também como a proposta de Dreyton, a abordagem de Andrade não adota ou propõe qualquer método para identificação das dependências de bugs. De forma mais prática, como também já citado, a abordagem proposta nesta dissertação propõe um método sistemático para realizar o mapeamento de dependências de componentes em dependências de bugs.

Considerando a modelagem multi-objetiva do problema de seleção de bugs, a proposta de Andrade é avaliada comparativamente em relação a duas metaheurísticas baseadas em algoritmos genéticos: NSGA-II e MOCell. Porém, diferente da abordagem proposta nesta dissertação, a proposta de Andrade não é avaliada em relação a competitividade humana, ou seja, os resultados obtidos automaticamente pelas metaheurísticas não são comparados com os resultados obtidos manualmente por membros da equipe de desenvolvimento.

Novamente, observe que, por ser modelada como um problema mono-objetivo, a abordagem proposta nesta dissertação adota a metaheurística clássica de algoritmos genéticos, e, portanto, não podem ser aplicadas as metaheurísticas multi-objetivas NSGA-II e MOCell. No entanto, como já mencionado, de forma relativamente simples, é possível em um momento futuro reformulá-la como um problema multi-objetivo.

3.4 AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE CORREÇÕES DE BUGS POR HABILIDADE E DISPONIBILIDADE DO DESENVOLVEDOR

Xiao e Afzal (2010) propõem uma abordagem de agendamento e alocação de bugs baseada em uma busca mono-objetiva para encontrar o máximo valor possível de correções de bugs. Para tal, a abordagem considera as atribuições de desenvolvedores apropriadas para um conjunto de tarefas de correção de bugs, acrescentando a importância de políticas eficientes e eficazes para atribuição de bugs. Xiao e Afzal argumentam que estas políticas afetam a satisfação dos clientes e a motivação dos desenvolvedores frente a otimização de recursos para tomada de decisões no âmbito da organização.

A proposta de Xiao e Afzal (2010) tem como diferencial um método de correspondência de capacidade entre atividades de bugs e recursos humanos, através do nível de habilidade e disponibilidade para as atividades de desenvolvimento e testes. Mas, por sua vez, o método proposto também tem suas limitações, pois exige uma sistemática que não é prática nas organizações pela dificuldade de implantação, pois requer não apenas o cadastramento dos relatórios de bugs, mas também das habilidades necessárias para resolvê-los. E, por outro lado,

outra limitação, também indicada pelo próprio autor, é o reconhecimento da necessidade de um nível de maturidade na organização sobre os seus processos para inferir e manter atualizadas as habilidades dos seus recursos.

Assim como esta dissertação, a proposta de Xiao e Afzal também faz uso de algoritmo genético. No entanto, a sua proposta compara os resultados do seu estudo apenas com o algoritmo *Hill Climbing*, ao contrário do que foi realizado na proposta desta dissertação, que foi realizada uma comparação com um algoritmo exaustivo para validar a qualidade das soluções encontradas em problemas menores. Além disso, na abordagem aqui proposta, também foram realizadas comparações com uma projeção de competitividade humana e com resultados de execuções do algoritmo aleatório.

O cálculo para a estimativa do esforço na proposta de Xiao e Afzal foi baseado pela quantidade de homem-hora investida no desenvolvimento e teste do bug corrigido e verificado (calculado pelo resultado do agendamento); o momento em que a correção deve ser iniciada; prazo final para correção e verificação (este atributo é utilizado como restrição para o agendamento); momento em que o bug deve ser corrigido e verificado, dado a sua estimativa de tempo gasto e momento inicial de correção (esse atributo é um resultado do agendamento); e, por fim, dois critérios utilizados na proposta desta dissertação que é a prioridade e a severidade.

Além dos critérios e restrições citados no parágrafo anterior, há ainda duas restrições utilizadas na abordagem de Xiao e Afzal que é a penalidade aplicada ao valor de um bug selecionado quando a tarefa de correção de bugs tem previsão de ultrapassar o prazo limite de correção. E por outro lado, um benefício pode ser incorrido no valor de um bug selecionado quando a tarefa de correção de bugs tem previsão de ser concluída antes do prazo.

Os objetivos da sua proposta então é alocar bugs com maior prioridade e severidade; com base no agendamento e obter o valor total máximo para correção de todos os bugs, mas se caso o bug não puder ser corrigido em um determinado prazo, o seu valor passa a ser penalizado com o valor 0 para importância do modelo, caso contrário o valor será calculado pela sua prioridade, severidade e peso preferencial e por fim, há o benefício para o bug que for corrigido antes do prazo para finalização.

Assim como esta dissertação, a abordagem de Xiao e Afzal faz uso de um estudo de caso real onde o líder de equipe do projeto planeja corrigir todos os bugs em um prazo definido e estima o esforço necessário para correção dos bugs através de julgamento de especialistas.

Porém, nesta abordagem não foi realizada uma comparação com a competitividade humana, ou seja, não houve uma comparação do julgamento de especialistas com o algoritmo utilizado.

Observou-se também que, na realização dos experimentos, foi dada preferência de peso no primeiro momento para prioridade e em um segundo momento para severidade, mas que juntos, os critérios não somavam 100% no peso total. Ao contrário, na abordagem desta dissertação, os experimentos não adotam preferências por prioridade ou severidade, dando uma calibração inicial de peso igual para cada critério, deixando a possibilidade de alteração posterior para atender a uma possível preferência dos clientes.

E por fim, a proposta do Xiao e Afzal (2010) não infere, nem explora dependências entre os bugs. Conforme já dito, a abordagem proposta nesta dissertação adota um modelo mais real, considerando os relacionamentos de precedência e de concorrência definidos por bugs interdependentes.

3.5 AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE CORREÇÃO DE BUGS POR EXPERIÊNCIA DO DESENVOLVEDOR EM ATIVIDADE ANTERIOR

Com um objetivo similar ao presente estudo, Netto et al. (2016) propôs uma abordagem automatizada para o problema de agendar um conjunto de tarefas de correção de bugs com diferentes prioridades para um conjunto de desenvolvedores disponíveis, fornecendo também uma solução que inclui uma alocação de tarefas de correção de bugs para desenvolvedores com experiência anterior em um determinado conjunto de categorias de bugs. Os objetivos da proposta de Netto é: (i) minimizar o tempo necessário para disponibilizar a próxima versão; (ii) garantir que os bugs mais urgentes sejam corrigidos na próxima release; e (iii) maximizar a taxa de ocupação da equipe de testes.

Assim como a presente abordagem, a proposta de Netto considera que os bugs devem ser associados a um componente no relatório de bug, considera a versão que o bug foi identificado, o tempo em que ele deve ser corrigido (que não é calculado pelo modelo) e a prioridade dos bugs que fornece a ordem em que devem ser corrigidos. Por outro lado, ao contrário da abordagem proposta, a abordagem de Netto não considera o conceito de severidade que é bem conhecido e amplamente adotado pelos modelos de informações para relatórios de bugs nos BTSs disponíveis, e também não inclui a restrição de dependências de bugs e orçamento, que por sua vez poderia permitir um preenchimento melhor e mais real das lacunas

de tempo dos desenvolvedores e testadores, criando assim oportunidade de trabalhos futuros nesta área.

Esta solução de Netto et al. (2016), apesar de ser interessante, também deve ser colocada como complemento em abordagem futuras desta proposta, mas ela deixa uma lacuna ao considerar as experiências dos desenvolvedores em tarefas futuras como forma de escolha para as próximas tarefas. A questão é a seguinte: e os novos desenvolvedores que não possuem experiência, como seriam utilizados em sua proposta?

A proposta de Netto adota uma abordagem mono-objetiva, na qual uma função de custo é modelada para medir a aderência do cronograma proposto, aumentando o número de bugs selecionados, priorizando os bugs mais urgentes e aumentando a taxa de ocupação da equipe de desenvolvimento. Como a abordagem aqui proposta, a abordagem de Netto et al. (2016) foi proposta como uma formulação ponderada de um único objetivo. No entanto, a proposta dele não apresenta nenhum estudo de caso para avaliar os seus resultados e muito menos uma comparação com a competitividade humana ou utilização de outros algoritmos.

3.6 CONSIDERAÇÕES FINAIS

Este capítulo abordou diversos aspectos correlacionados a este trabalho, através dos trabalhos relacionados, que estão inclusive, com um escopo acima deste, por exemplo, enquanto esta dissertação trata da seleção de bugs para correção, os trabalhos relacionados envolvem não só a seleção, mas também o agendamento e a alocação de tarefas de correção de bugs. Todavia este trabalho traz como principal contribuição a dependência de bugs baseada em componentes, uma forma automática de rastreamento desta dependência, além dos conceitos de grau de cobertura e grau de interdependência dos bugs.

E a razão pelo uso de componentes é justificado no sentido de que eles estão mais próximos do código, ou seja, decidir a relação de dependência através de componentes no contexto da engenharia de software pode trazer mais eficiência em seu rastreio, por exemplo, do que em requisitos.

Quanto a natureza dos componentes, segundo Spagnoli e Becker (2003), os componentes são tratados como artefatos por poderem assumir uma variedade de diferentes formas como, por exemplo, código fonte, documentação e código executável. Ora, se os bugs ocorrem durante e após a fase de implementação, e os componentes podem ser representados por artefatos derivados do código fonte, então assumir as dependências de bugs baseada em componentes já que é ligada diretamente ao código, seria melhor do que basear por requisitos.

A origem da busca das dependências baseada em componentes foi também assimilada na pesquisa de Burity e Elias (2011), que reforça a ideia da utilização da abordagem de componentes quando a utiliza para escolha na montagem de módulos. Note que a abordagem de dependência de bugs baseada em componentes se dá porque é possível gerar a dependência de bugs pelo mapeamento de dependências de componentes, tendo em vista que os bugs ocorrem nos componentes, que, por sua vez, representam o comportamento funcional do sistema e dos elementos presentes nele.

Portanto, a abordagem apresentada nesta dissertação é a primeira proposta que explora dependências de bugs baseada em componentes, lidando com relações de precedência ou concorrência entre bugs. Além disso, como uma contribuição adicional, em vez de considerar as dependências de bugs simplesmente como um parâmetro de entrada, o que deixaria a proposta muito limitada, a abordagem proposta adota uma nova estratégia de rastreamento em que as dependências de bugs podem ser inferidas diretamente das relações entre os componentes no nível arquitetural de projetos de software baseados em componentes, que, somente posteriormente, é utilizado como variável de entrada calculado de forma automatizada.. De forma resumida, a Tabela 1 apresenta a análise comparativa do presente trabalho (identificado como LIMA 2018) frente aos demais trabalhos relacionados.

Como pode ser observado na Tabela 1, a técnica de busca mais utilizada nos trabalhos relacionados é baseada em algoritmos genéticos, seja em sua concepção mono-objetiva ou multi-objetiva, fato que evidencia a adequabilidade desta classe de algoritmos ao problema de seleção de bugs.

Embora não seja incluída na Tabela 1, a abordagem proposta por Burity e Elias (2011) merece destaque, pois trata de uma abordagem de recomendação de módulos de software em Linhas de Produtos de Software que explora o conceito de dependências de componentes, tendo o objetivo de construir módulos com baixo acoplamento e alta coesão. É importante destacar que a abordagem (BURITY e ELIAS, 2011) foi adotada como inspiração para propor a identificação de dependências de bugs a partir das dependências de componentes.

Tabela 1 - Comparação com trabalhos relacionados

Proposta	Dependências de Bugs		Problema	Algoritmo de Busca	Múltiplos clientes	Parâmetros
	Representação	Identificação				
LIMA 2018	Grafo Direcionado Cíclico ou Acíclico	Método baseado em Dependências de Componentes Arquiteturais	Seleção	Algoritmo Genético	Votação	Prioridade, Severidade, Custo, Orçamento
DREYTON 2015	Grafo Direcionado Acíclico	Variável de entrada do modelo	Seleção Priorização	Algoritmo Genético	Votação	Relevância, Prioridade, Severidade
DREYTON 2016	Grafo Direcionado Acíclico	Variável de entrada do modelo	Seleção Priorização Alocação	NSGA-II MOCcell IBEA	Votação	Relevância, Prioridade, Severidade, Experiência
ANDRADE 2011	Grafo Direcionado Acíclico	Variável de entrada do modelo	Seleção	NSGA-II MOCcell	Votação	Preferência, Severidade, Tempo, Orçamento
XIAO 2010	-	-	Agendamento Alocação	Algoritmo Genético	-	Prioridade, Severidade, Competência, Esforço, Prazo
NETTO 2016	-	-	Agendamento Alocação	Algoritmo Genético	-	Tempo, Custo, Prioridade, Severidade, Classe do bug

4 ABORDAGEM PROPOSTA

Nesta seção apresenta-se a abordagem proposta para automação do processo de seleção de bugs para correção da próxima *release*, denominado aqui como PSB.

É necessário antes de adentrar no contexto das camadas do problema no PSB saber em qual momento do ciclo de vida (estudado na seção 2.1 desta dissertação) do bug o problema está inserido, para contextualizar melhor o leitor.

Com base na Figura 1, que trata do ciclo de vida de um bug, esta abordagem se insere em etapa anterior à fase inicial de correção do bug e também na etapa posterior a sua validação. A seguir tem-se as etapas que ocorrem até a validação do bug.

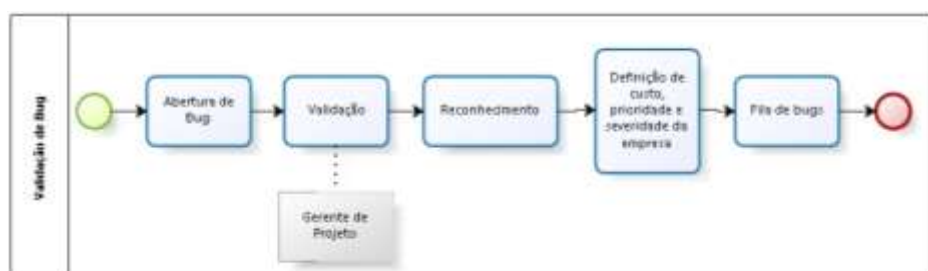
O processo começa com a identificação de uma falha no sistema e a informação referente a esta falha precisa chegar ao gerente de projetos. Para isso, o cadastro do relatório de bug pode ser efetuado ou pelo cliente (equipe de TI da organização que colabora diretamente com o negócio), ou pelo produtor (equipe de desenvolvimento ou de testes).

Como está descrito na fundamentação teórica, o cadastro pode ser realizado em um sistema de rastreamento de bugs (BTS) que pode ser qualquer um disponível no mercado, mas que absorva as informações que serão necessárias para o modelo, ou seja, os critérios podem ser configurados (o modelo para esta proposta foi executado no *REDMINE*).

Ao final do cadastro é gerado um relatório de bug ainda incompleto, porque necessita de validação e de preenchimento de outros dados.

Após o relatório ser cadastrado, os bugs encontrados em produção serão validados e considerados fazendo parte da fase NOVO (Figura 1). No momento da validação, o gerente de projetos pode eliminar duplicações de bugs e alimentar os campos referentes à empresa. Para a abordagem em questão, o bug, após ser validado, é enviado para uma fila ou repositório de bugs, conforme ilustrado na Figura 16.

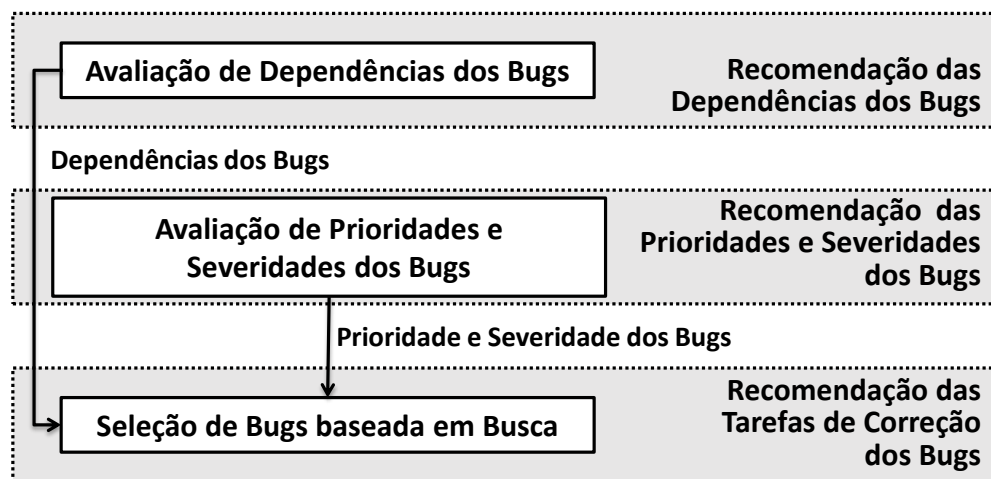
Figura 16 - Validação de Bug



Com os bugs validados no repositório de bugs é possível definir as próximas etapas que constituem a proposta desta abordagem, que são: definir as etapas do problema e os dados de entrada da proposta para depois fazer a seleção dos bugs através de uma busca com o algoritmo genético (seção 2.2.1). E por fim oferecer um conjunto de sugestões de quais bugs serão implementados para a próxima versão do sistema.

Assim, no contexto do PSB, este trabalho propõe uma abordagem automatizada para selecionar bugs em projetos de software baseados em componentes. Desta forma, conforme mostrado na Figura 17, a abordagem proposta adota uma arquitetura em camadas, estruturada em três fases.

Figura 17 - Arquitetura em 3 camadas



Fonte: Lima e Elias (2017).

A primeira fase, denominada de *Recomendação das Dependências dos Bugs*, é responsável por avaliar as dependências dos componentes e depois derivar as dependências dos bugs, estabelecendo relações de precedência e de concorrência entre suas respectivas tarefas de correção de bugs. Como resultado principal, esta fase produz uma matriz que representa as dependências dos bugs, extraídas de informações diretamente disponíveis nos relatórios de bugs, mas também explorando a arquitetura baseada em componentes do projeto de software.

A segunda fase, denominada *Recomendação das Prioridades e Severidades dos Bugs*, é responsável por avaliar a prioridade e a severidade de cada bug do ponto de vista do produtor de software e seus clientes. Conforme mencionado, a proposta adota um modelo multicliente para indicar os bugs mais relevantes, refletindo não apenas a perspectiva do produtor de software, mas também o feedback dos clientes sobre a importância e o impacto dos bugs relatados.

Adotando como entradas as dependências dos bugs, as prioridades dos bugs e as severidade dos bugs, a terceira fase, denominada *Recomendação das Tarefas de Correção dos Bugs*, adota um algoritmo baseado em busca para selecionar os bugs mais relevantes, mas com o custo total limitado pelo orçamento do projeto de software disponível. É importante enfatizar que esta fase produz um conjunto de possíveis seleções de bugs, ordenadas de acordo com sua relevância, que podem ser adotadas pelo gerente do projeto para escolher as tarefas de correção de bugs.

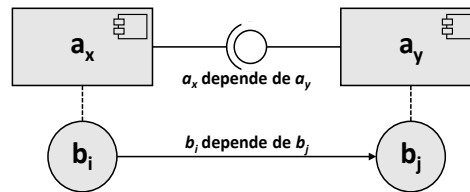
Ao apresentar a arquitetura em camadas, o restante desta seção descreve a proposta com mais profundidade: como identificar as dependências dos bugs; como avaliar a prioridade e severidade dos bugs; e como reformular o PSB como um problema de busca.

4.1 RECOMENDAÇÃO DAS DEPENDÊNCIAS DOS BUGS

Para identificar as dependências dos bugs, a abordagem proposta assume que as dependências entre componentes de software implicam em dependências entre os respectivos bugs associados. Assim, a abordagem explora a arquitetura baseada em componentes do projeto de software para rastrear e identificar as dependências dos bugs, trazendo assim duas contribuições. A primeira contribuição é o próprio conceito de dependência baseada em componentes. A segunda contribuição diz respeito ao método de identificação automática das dependências.

Felizmente, cada relatório de bug geralmente indica o artefato alvo onde ocorre o bug relatado, possibilitando rastrear as dependências dos bugs a partir das dependências dos componentes. A ideia subjacente é que, como ilustrado na Figura 18, um bug b_j ocorrendo em um componente a_y pode causar, impactar ou afetar outro bug b_i que ocorre em outro componente a_x que depende do primeiro. Em outras palavras, considerando que o componente a_x depende do componente a_y , a abordagem proposta tem a compreensão que o bug b_i pode depender do bug b_j . À primeira vista, ambos os bugs podem parecer distintos, mas certamente compartilham sintomas e mensagens de erros semelhantes. Este entendimento converge com o conceito de que a arquitetura de componentes é definida como um conjunto de componentes que representam as relações e suas dependências comportamentais ou funcionais.

Figura 18 - Dependência entre componentes e bugs relacionados



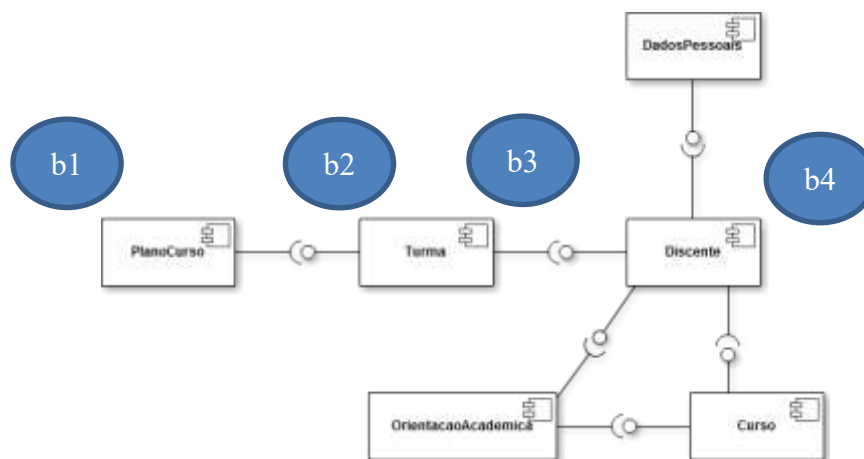
Fonte: Lima e Elias (2017).

A escolha na utilização desta abordagem foi decorrente do entendimento de que componentes estão mais relacionados ao código do que os requisitos. Acredita-se que os resultados desta abordagem na dependência de bug também reflitam uma relação melhor com o código, trazendo assim soluções mais eficientes, no que se refere a relação de um bug a outro.

Com base no exemplo apresentado na Figura 18, pode-se dizer que a seleção de um bug automaticamente implicará na seleção para correção dos bugs que o precedem.

Para uma representação mais detalhada da dependência contida na Figura 18, a representação gráfica de uma arquitetura de componentes na Figura 19 aborda uma exemplificação do que seria a dependência de bug baseada em componentes, com 4 bugs, (b1 até b4), demonstrando a dependência dos bugs em decorrência das dependências entre os componentes.

Figura 19 - Dependência de bugs baseada em componentes



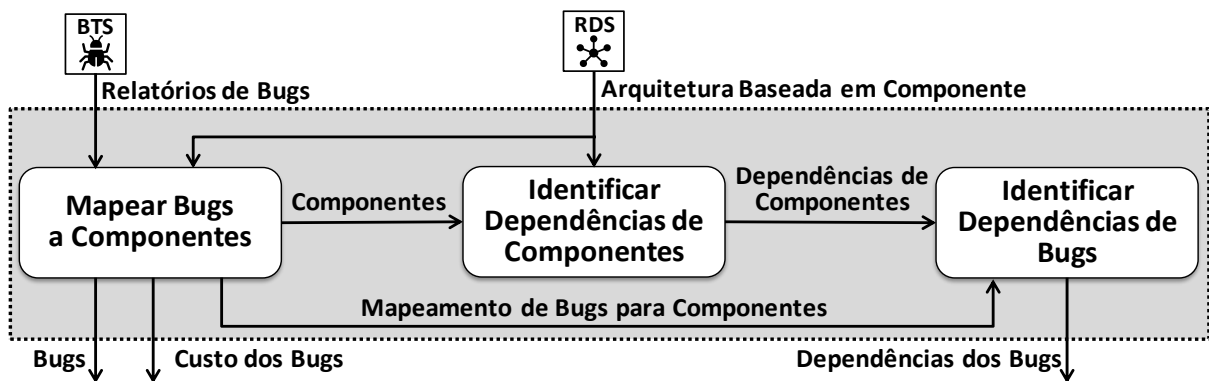
Conforme a Figura 19, sabe-se que o componente *PlanoCurso* utiliza serviços providos pelo componente *Turma*, que por sua vez utiliza serviços providos pelo componente *Discente*. Assim, *PlanoCurso* depende de *Turma* que depende de *Discente*. Consequentemente, um defeito em *Discente* pode provocar um defeito em *PlanoCurso* ou *Turma*, configurando uma dependência funcional. Desta forma, as soluções válidas para o problema relatado, considerando as dependências entre os bugs, seriam:

- resolver apenas b4, pois o componente que ele está associado não possui componentes que ele depende com bugs, como é o caso de *Curso* e *DadosPessoais*;
- resolver b2, b3 e b4, pois b2 e b3 que ocorre em *Turma* depende de *Discente* que possui b4, e, por isso, b4 deve também fazer parte dos bugs selecionados para correção na solução. Além disso, os bugs b2 e b3 que se encontram no mesmo componente são dependentes entre si;
- resolver b1, b2, b3 e b4, pois ao selecionar b1 que ocorre em *PlanoCurso*, deve-se selecionar b2, b3 e b4 porque pertencem a componentes que *PlanoCurso* depende direta e indiretamente.

Este entendimento de dependência deve ocorrer, pois é possível que uma alteração no componente do bug menos dependente afete o uso desse componente por outro que dependa dele. Sendo assim, é importante que se respeite as precedências no momento de corrigir os bugs, corrigindo b4 para depois corrigir b2 ou b3, e, somente em seguida corrigir b1.

Para detalhar como se dá o processo de recomendação destas dependências, a Figura 20 apresenta três etapas:

Figura 20 - Recomendação das Dependências de Bugs



Fonte: Lima e Elias (2017).

A primeira etapa, denominada *Mapear Bugs a Componentes*, tem como objetivo principal identificar o componente de software em que cada bug ocorre. Para fazer isso, esta etapa adota como entrada: (i) os *relatórios de bugs* disponíveis em um BTS; e (ii) a *arquitetura baseada em componentes* disponível em um Repositório de Documentação de Software (RDS) – *Software Documentation Repository*.

Inicialmente, como elemento básico da abordagem proposta, esta etapa produz como saída o conjunto de bugs candidatos relatados no BTS, representados na abordagem proposta pelo conjunto *B*, conforme indicado na equação (1). Além disso, como uma informação

arquitetural básica, esta etapa adota a arquitetura do software como entrada para caracterizar o projeto de software baseado em componentes como um conjunto de componentes ou artefatos de software, representados na abordagem proposta pelo conjunto A , conforme indicado na equação (2).

$$B = \{b_1, b_2, \dots, b_m\} \quad (1)$$

$$A = \{a_1, a_2, \dots, a_k\} \quad (2)$$

Ao identificar o conjunto de bugs B e o conjunto de componentes de software A , a primeira etapa deve extrair o *mapeamento de bugs a componentes*. Conforme já mencionado, cada relatório de bug geralmente indica um componente de software no qual ocorre o bug. Assim, é possível a partir do conjunto de relatórios de bugs em um BTS extrair o mapeamento de bugs a componentes. Aqui, este mapeamento é representado pela matriz binária $BA_{B \times A}$, em que cada termo $ba_{i,j}$ é definido na equação (3), onde linhas e colunas representam bugs e componentes, respectivamente. É importante notar que, conforme definido pela equação, cada bug ocorre em apenas um componente, mas cada componente pode ser relacionado a zero ou mais bugs.

$$BA_{B \times A} = \left\{ ba_{i,j} \mid \exists_{b_i} b_i \in B \wedge \exists_{a_j} a_j \in A \wedge ba_{i,j} \in \{0, 1\} \right\} \quad (3)$$

$$ba_{i,j} = \begin{cases} 1 & \text{se o bug } b_i \text{ ocorre no componente } a_j \\ 0 & \text{caso contrário} \end{cases}$$

$$\sum_{a_j \in A} ba_{i,j} = 1$$

$$\sum_{b_i \in B} ba_{i,j} \geq 0$$

Com estas informações pode-se gerar a matriz ilustrada na Figura 21, ou seja, demonstrar a lista dos componentes com seus respectivos bugs, em que cada célula representa um bug b que varia de 1 a m associado a um componente a que varia de 1 a k .

Figura 21 - Matriz Conceitual BA de associação de bugs com componentes

	a_1	a_2	...	a_k
b_1	$ba_{1,1}$	$ba_{1,2}$	...	$ba_{1,k}$
b_2	$ba_{2,1}$	$ba_{2,2}$	...	$ba_{2,k}$
...	...	...	...	...
b_m	$ba_{m,1}$	$ba_{m,2}$	...	$ba_{m,k}$

Para ilustrar melhor a relação descrita acima, considere como exemplo a célula $ba_{2,1}$ na Figura 22. Se o valor for 1, o bug b_2 ocorre no componente a_1 . Por outro lado, se o valor for 0 indica que este bug não ocorre neste componente.

Figura 22 - Matriz BA de associação de bugs com componentes

	a_1	a_2	a_3
b_1	1	0	0
b_2	0	1	0
b_3	0	1	0
b_4	0	0	1

A matriz de exemplo de associação de bugs a componentes identifica a lista dos bugs com seus respectivos componentes. Neste exemplo, o bug b_1 ocorre no componente a_1 , os bugs b_2 e b_3 ocorrem no componente a_2 , e, por fim, o bug b_4 ocorre no componente a_3 .

Como uma saída adicional, com base em informações em relatórios de bugs, esta etapa também traz como saída o custo ou esforço para correção do bug, representada na abordagem proposta pelo conjunto V , conforme definido na equação (4). Como dito anteriormente, o custo ou esforço para corrigir um bug é uma informação introduzida por desenvolvedores em relatórios de bugs. Observe que o valor para correção do bug b_i é representado pelo termo v_i , cujo valor é maior que zero.

$$V = \{v_i \mid \exists_{b_i} b_i \in B \wedge v_i > 0\} \quad (4)$$

A segunda etapa, denominada *Identificar Dependências de Componentes*, tem o objetivo de identificar as dependências de componentes a partir da arquitetura do sistema em questão. Na abordagem proposta, um componente a_j depende de outro componente a_k se e somente se a_j requer uma operação provida por a_k .

Para extrair as dependências entre os componentes, esta etapa adota como entrada a *arquitetura baseada em componentes* juntamente com o conjunto de componentes de software relacionados. É importante destacar que, para que o trabalho não fique limitado a apenas arquiteturas baseadas em componentes, o conceito de componente pode ser abstraído para outros níveis de menor granularidade, com o objetivo de utilizar outros tipos de artefatos, como diagrama de classe, pacotes do sistema, módulos, etc., de forma que quanto menor for o nível de granularidade, maior o esforço de processamento e maior a precisão dos resultados.

Para derivar as dependências dos componentes, primeiramente, a arquitetura baseada em componentes é representada e transformada em um grafo direcionado, no qual os vértices são definidos por componentes ou artefatos e as arestas direcionadas são definidas por interfaces requeridas e providas, apontando a partir do componente chamador que possui a interface requerida em direção ao componente chamado que possui a interface provida. Por exemplo, na Figura 18, o grafo direcionado resultante tem dois vértices, digamos a_x e a_y , e apenas uma aresta partindo de a_x para a_y .

A abordagem proposta representa dependências de componentes por uma matriz de adjacência, que é uma matriz quadrada binária, chamada $AD_{A \times A}$, cuja definição está na equação (5). As linhas e colunas da matriz representam componentes, de modo que seu elemento ad_{jk} é 1 quando há uma relação de dependência do componente a_j com o componente a_k , e 0 quando não há relação entre eles. É importante notar que, conforme definido, cada componente pode depender de 0 ou mais componentes. Por analogia, cada componente pode ter 0 ou mais componentes dependentes.

$$AD_{A \times A} = \left\{ ad_{j,k} \mid \exists_{a_j} a_j \in A \wedge \exists_{a_k} a_k \in A \wedge ad_{j,k} \in \{0, 1\} \right\} \quad (5)$$

$$ad_{j,k} = \begin{cases} 1 & \text{se o componente } a_j \text{ depende do componente } a_k \\ 0 & \text{caso contrário} \end{cases}$$

$$\sum_{a_k \in A} ad_{j,k} \geq 0$$

Com estas informações, pode-se então trabalhar com um exemplo para demonstração de uma arquitetura de componentes e/ou diagrama de componentes. Na Figura 23, será visualizada a dependência funcional do componente a_1 em relação ao componente a_2 e do componente a_2 para o componente a_3 . Para demonstrar a relação de dependência de um

componente com outro utilizamos aqui a matriz conceitual de dependência direta de componentes na Figura 24, onde cada célula representa a relação de um componente com os demais, por exemplo a célula $ad_{1,2}$ representa a relação de dependência do componente a_1 que depende do componente a_2 .

Figura 23 - Diagrama de Componentes

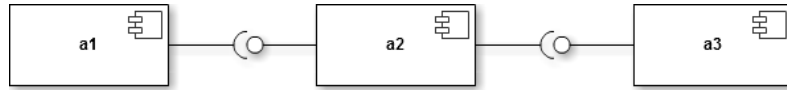


Figura 24 - Matriz Conceitual AD de dependência direta de componentes

	a_1	a_2	...	a_k
a_1	$ad_{1,1}$	$ad_{1,2}$	...	$ad_{1,k}$
a_2	$ad_{2,1}$	$ad_{2,2}$	...	$ad_{2,k}$
...	...	...	...	...
a_j	$ad_{j,1}$	$ad_{j,2}$	...	$ad_{j,k}$

Para ilustrar melhor a relação descrita acima, considere como exemplo a célula $ad_{1,2}$ na Figura 24. Lê-se a dependência da linha para coluna, assim a posição $ad_{1,2}$ é a dependência do componente a_1 em relação ao componente a_2 . Desta forma, a Matriz AD que ilustra a dependência direta de componentes na Figura 25 representa o resultado do mapeamento da matriz quadrada $AD_{A \times A}$.

Figura 25 - Matriz AD de dependência direta de componentes

	a_1	a_2	a_3
a_1	1	1	0
a_2	0	1	1
a_3	0	0	1

A representação na Figura 25 é resultante do entendimento adquirido do trabalho de BURITY (2011). Em sua proposta, os valores de dependência podem variar entre 0 e 1. Porém, na presente proposta, qualquer valor de dependência maior que 0 é considerado como 1. O motivo desta adaptação é que ao gerar a matriz de dependência de bugs, os valores para cada relação são representados apenas para inferir se pode haver ou não relação de um bug com outro. Sendo assim, qualquer relação entre componentes será considerada nesta proposta como verdadeira ou falsa para bugs.

Agora, na terceira etapa, chamada *Identificar Dependências de Bugs*, o mapeamento de bugs a componentes $BA_{B \times A}$, juntamente com as dependências de componentes $AD_{A \times A}$, são manipuladas matematicamente usando operações de matrizes booleanas para identificar as dependências de bugs, representadas na abordagem proposta pela matriz quadrada binária $BD_{B \times B}$. Como já ilustrado na Figura 18, um bug b_i relacionado a um componente a_x depende de outro bug b_j relacionado a um componente a_y se e somente se a_x depender de a_y .

Para extrair as dependências de bugs $BD_{B \times B}$, conforme definido na equação (6), a terceira etapa capta todas as dependências diretas e indiretas que um bug possui com outros bugs, de modo semelhante ao conceito conhecido como dependência de visibilidade, do inglês *visibility-dependency*, proposta por Sharman e Yassine (2004) para avaliar arquiteturas de software complexas.

$$BD_{B \times B} = \left(BA_{B \times A} \cdot \sum_{n=0}^{|A|} AD_{A \times A}^n \right) \cdot BA_{B \times A}^T \quad (6)$$

Para calcular as dependências de todos bugs $BD_{B \times B}$, a abordagem proposta explora o conceito de *fechamento transitivo* do grafo direcionado, estabelecido pela arquitetura baseada em componentes, que define outro grafo direcionado com o mesmo conjunto de vértices, mas com uma aresta do componente a_x para o componente a_y se e somente se a_y é alcançável a partir de a_x no grafo direcionado definido pela arquitetura baseada em componentes.

Para fazer isso, as dependências de componentes $AD_{A \times A}$ são adotadas como uma Matriz de Estrutura de Projeto, do inglês *Design Structure Matrix* (DSM) (PIMMLER, 1994). Em primeiro lugar, cada n -ésima potência da matriz de alcançabilidade (WARFIELD, 1973), representada como $AD_{A \times A}^n$, é calculada para encontrar as dependências diretas e indiretas que cada componente possui com outros componentes para todos os comprimentos de trajetória sucessivos, ou seja, para uma maior quantidade de níveis que possa possuir um grafo. O somatório dessas matrizes de alcançabilidade produz uma matriz de dependências de visibilidade $AD_{A \times A}^{VD}$, que mostra todas as dependências diretas e indiretas entre os componentes para todos os comprimentos de caminho possíveis até o máximo, definido pelo tamanho do conjunto de componentes A . É importante ressaltar que a matriz $AD_{A \times A}^0$ é representada pela matriz identidade, sinalizando que conceitualmente cada componente depende dele mesmo.

Agora, a multiplicação booleana das dependências diretas de bugs para componentes, representada pelo mapeamento $BA_{B \times A}$, e a matriz de dependências de visibilidade resultante

$AD_{A \times A}^{VD}$ gera a matriz de dependências diretas e indiretas de bugs a componentes $BA_{B \times A}^{VD}$, evidenciando todos os componentes afetados por cada bug. Por fim, a multiplicação booleana da matriz de dependências diretas e indiretas de bugs a componentes $BA_{B \times A}^{VD}$ e a matriz transposta do mapeamento de bugs para componentes, representada por $BA_{B \times A}^T$, produz todas as dependências diretas e indiretas entre bugs $BD_{B \times B}$, revelando todos os bugs afetados por cada bug.

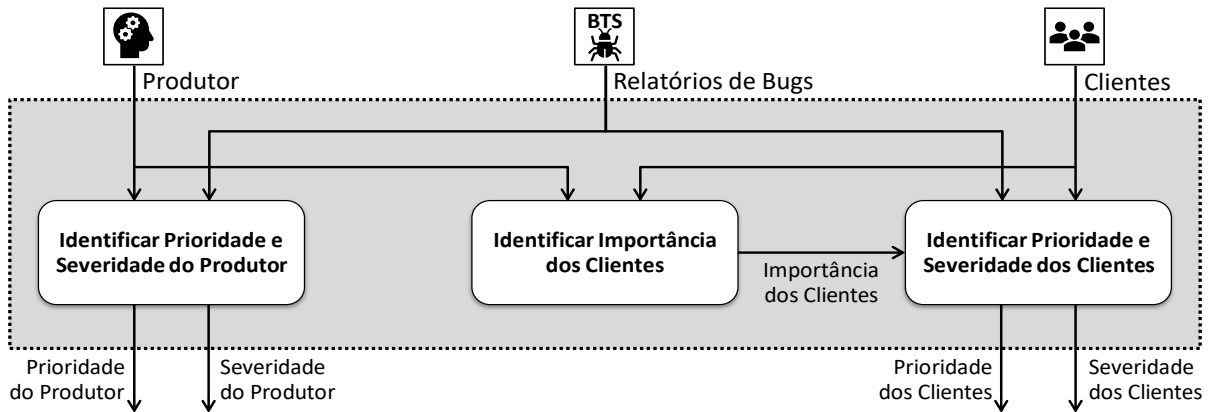
É importante destacar que, na abordagem proposta, as dependências de bugs $BD_{B \times B}$ também são chamadas de Soluções Básicas $BX_{B \times B}$, conforme indicado na equação (7). Essa terminologia é adotada devido ao seguinte raciocínio. Digamos que b_i seja um bug, então, todos os bugs marcados com o valor 1 na linha i das Soluções Básicas $BX_{B \times B}$, definido pela matriz linha $[bx_{i,1}, bx_{i,2}, \dots, bx_{i,m}]$, representam os bugs que devem ser corrigidos juntos com o bug b_i ou antes dele. Na verdade, se o bug b_i for selecionado para ser corrigido, todos os demais bugs que o bug b_i depende também devem ser selecionados para correção.

$$BX_{B \times B} = BD_{B \times B} \quad (7)$$

4.2 RECOMENDAÇÃO DAS PRIORIDADES E SEVERIDADES DOS BUGS

Como meio para estimar a prioridade e a severidade do bug, a abordagem proposta pressupõe que o produtor e cada cliente forneçam informações sobre cada bug relatado. Para tal, como mostrado na Figura 26, a segunda fase, denominada *Recomendação das Prioridades e Severidades dos Bugs*, é estruturada em três etapas.

Figura 26 - Recomendação das Prioridades e Severidades dos Bugs



Fonte: Lima e Elias (2017).

No lado do produtor, a primeira etapa denominada *Identificar Prioridade e Severidade do Produtor* tem o objetivo principal de atribuir valores de prioridade e severidade para cada bug relatado no BTS, gerando como saída a *Prioridade do Produtor* e a *Severidade do*

Produtor, representados na abordagem proposta pelos termos *PP* e *PS*, conforme definido nas equações (8) e (9), respectivamente.

$$PP = \{pp_i \mid \exists_{b_i} b_i \in B \wedge pp_i \in \{P1, \dots, P5\}\} \quad (8)$$

$$PS = \{ps_i \mid \exists_{b_i} b_i \in B \wedge ps_i \in \{blocker, \dots, trivial\}\} \quad (9)$$

Conforme mencionado anteriormente, a abordagem proposta adota os valores de prioridade e severidade definidos no *Bugzilla*. Por um lado, os valores de *prioridade* variam de *P1* a *P5*, onde *P1* é a prioridade mais alta e *P5* é a prioridade mais baixa. Por outro lado, os valores de severidade variam de *blocker* a *trivial*, onde o *blocker* é a maior severidade e *trivial* é a menor severidade. Como os valores de prioridade e severidade devem ser representados numericamente, a abordagem proposta adota a conversão expressa na Tabela 2.

Tabela 2 - Valores de prioridade e severidade convertidos em número

Prioridade	<i>P1</i>	<i>P2</i>	<i>P3</i>	<i>P4</i>	<i>P5</i>
Severidade	<i>blocker</i>	<i>critical</i>	<i>major</i>	<i>minor</i>	<i>trivial</i>
Valor	1.0	0.8	0.6	0.4	0.2

Para facilitar o entendimento, pode-se compreender também a prioridade como imediata para *P1*, alta para *P2*, média para *P3*, baixa para *P4* e sem prioridade para *P5*. Um bom exemplo de uma prioridade alta seria um erro de ortografia no nome da empresa em uma logomarca. Segundo a fundamentação apresentada no Capítulo 2, este mesmo erro representaria uma severidade com impacto *trivial* por fazer referência a problemas de menor impacto em relação ao sistema.

Ainda no lado do produtor, mas considerando seus clientes, a segunda etapa, denominado de *Identificar Importância dos Clientes*, tem como objetivo principal atribuir a importância de cada cliente na perspectiva do produtor de software. Conforme proposto aqui, os clientes são definidos pelo conjunto *C*, representado na equação (10).

$$C = \{c_1, c_2, \dots, c_s\} \quad (10)$$

O fato dos BTSs já possuírem como padrão os campos de prioridade e severidade em seus relatórios de bugs demonstra a importância deles como critérios de avaliação. Inclusive, verifica-se no *Bugzilla* o preenchimento destes critérios pelos usuários. Esta modelagem que permite a avaliação dos critérios dos bugs pelos clientes corrobora para o produtor tomar

decisões quanto a importância deles, pois fornece subsídios para as avaliações por parte do produtor, e isso impacta diretamente na seleção dos bugs para correção, trazendo mais satisfação para os clientes e equilíbrio nas entregas.

Como os recursos são escassos é importante também avaliar a qualidade/confiabilidade/escolhas distintas dos clientes, e, neste cenário, os clientes podem ter interesses diferentes ou mesmo conflitantes sobre os bugs a serem corrigidos na próxima versão. A fim de lidar com esses conflitos de interesse, o produtor de software deve adotar uma estratégia conciliadora para balancear as preferências conflitantes. Aqui, essa estratégia baseia-se no conceito de importância dos clientes, indicando a importância de cada cliente para a estratégia de negócios do produtor.

Conforme definido na equação (11), a *Importância dos Clientes* é representada pelo conjunto I , onde cada termo i_x pode assumir valores no intervalo $(0, 1] \in \mathbb{R}$, indicando a importância associada ao respectivo cliente c_x . Observe que, cada cliente c_x deve ter algum grau de importância, e, portanto, a importância do cliente i_x não pode ser igual a zero.

$$I = \{i_x \mid \exists_{c_x} c_x \in C \wedge i_x \in (0, 1]\} \quad (11)$$

Por exemplo, considerando um sistema acadêmico adotado em uma dada universidade, certamente, as coordenações de cursos, apoiadas pelas respectivas equipes de TI, podem ser considerados os principais clientes, tendo portanto melhores condições de definir as prioridades e severidades dos bugs relatados. Por sua vez, a importância de cada coordenação de curso pode estar relacionada ao seu valor institucional. Neste caso, a importância de cada coordenação de curso poderia ser estimada em função da quantidade de alunos inscritos nos respectivos cursos.

Neste ponto, a terceira etapa, denominada *Identificar Prioridade e Severidade dos Clientes*, pode ser iniciado no lado dos clientes e tem o objetivo principal de estimar os valores de prioridade e severidade de cada bug relatado no BTS, gerando como saída a *Prioridade dos Clientes e a Severidade dos Clientes*, representada na abordagem proposta pelos termos CP e CS , conforme definido nas equações (12) e (13), respectivamente.

$$CP = \{cp_i \mid \exists_{b_i} b_i \in B \wedge cp_i \in [0, 1]\} \quad (12)$$

$$CS = \{cs_i \mid \exists_{b_i} b_i \in B \wedge cs_i \in [0, 1]\} \quad (13)$$

Note que, ao contrário da Prioridade do Produtor PP e da Severidade do Produtor PS , em que o produtor atribui diretamente os valores discretos na Tabela 2, as Prioridades dos

Clientes CP e as Severidade dos Clientes CS não são diretamente atribuídas por eles. Em vez disso, para cada bug b_i , a Prioridade dos Clientes cp_i e a Severidade dos Clientes cs_i são respectivamente estimadas como o valor médio ponderado da prioridade específica do cliente $csp_{x,i}$ e a severidade específica do cliente $css_{x,i}$ para cada cliente específico c_x em relação a cada bug b_i , assumindo dessa forma valores no intervalo $[0, 1] \in \mathbb{R}$, conforme definido nas equações (14) e (15), respectivamente:

$$cp_i = \frac{\sum_{c_x \in C} i_x \cdot csp_{x,i}}{|C|} \quad (14)$$

$$cs_i = \frac{\sum_{c_x \in C} i_x \cdot css_{x,i}}{|C|} \quad (15)$$

Sendo assim a prioridade e a severidade do bug b_i para cada cliente c_x podem ser representadas pelas matrizes da Figura 27, em que cada cliente avalia a prioridade e severidade de cada bug:

Figura 27 - Matriz Conceitual de avaliação dos bugs pelos clientes

	b_1	b_2	...	b_m		b_1	b_2	...	b_m
c_1	$csp_{1,1}$	$csp_{1,2}$	...	$csp_{1,m}$	c_1	$css_{1,1}$	$css_{1,2}$	...	$css_{1,m}$
c_2	$csp_{2,1}$	$csp_{2,2}$	...	$csp_{2,m}$	c_2	$css_{2,1}$	$css_{2,2}$	...	$css_{2,m}$
...	...	...	...	...	...	...	...	...	...
c_s	$csp_{s,1}$	$csp_{s,2}$	...	$csp_{s,m}$	c_s	$css_{s,1}$	$css_{s,2}$	...	$css_{s,m}$

Portanto, da mesma forma que o produtor de software, cada cliente deve atribuir valores de prioridade e severidade para cada bug relatado no BTS, gerando as relações *prioridade específica do cliente* e *severidade específica do cliente*, representada por $CSP_{C \times B}$ e $CSS_{C \times B}$, conforme definido nas equações (16) e (17), respectivamente, que também podem assumir os valores discretos indicados na Tabela 2.

$$CSP_{C \times B} = \left\{ csp_{x,i} \mid \exists_{c_x} c_x \in C \wedge \exists_{b_i} b_i \in B \wedge csp_{x,i} \in \{P1, \dots, P5\} \right\} \quad (16)$$

$$CSS_{C \times B} = \left\{ css_{x,i} \mid \exists_{c_x} c_x \in C \wedge \exists_{b_i} b_i \in B \wedge css_{x,i} \in \{blocker, \dots, trivial\} \right\} \quad (17)$$

4.3 RECOMENDAÇÃO DAS TAREFAS DE CORREÇÃO DOS BUGS

Com base nas informações relativas às dependências, custos, prioridades e severidades dos bugs, estimadas na primeira e segunda fase da abordagem proposta, a terceira fase denominada *Recomendação das Tarefas de Correção dos Bugs* adota conceitos e técnicas de

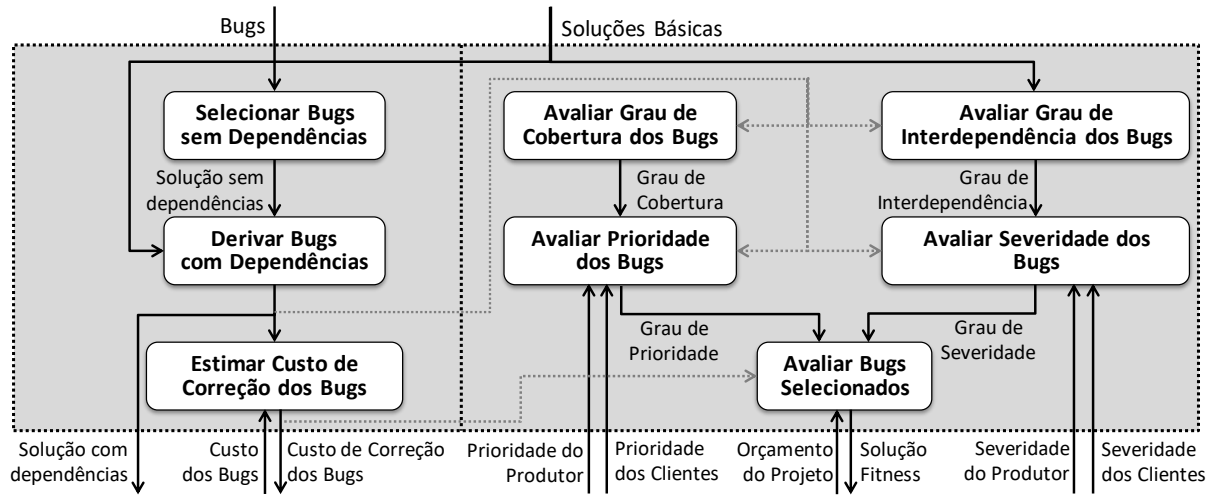
SBSE para representar o Problema de Seleção de Bugs como um problema de otimização para recomendar um conjunto suficientemente bom de soluções ou mesmo, soluções ótimas.

Nesta direção, a terceira fase é estruturada com base na estratégia de busca de algoritmos genéticos (HOLLAND, 1992), que se inspira no processo de evolução biológica e seleção natural. Uma vez que a representação genética e a função de avaliação são definidas, o algoritmo genético prossegue inicializando uma população de soluções candidatas e, em seguida, melhora a população através de um processo geracional, com base na aplicação repetitiva de operadores genéticos clássicos, como seleção, cruzamento, mutação e substituição, que conclui quando uma condição de término é atingida.

Os operadores definidos para utilização desta etapa foram: a seleção por torneio; o cruzamento baseado em máscara com a geração de dois filhos, alternando a escolha do gene do pai quando a máscara for 0 ou 1 para geração do segundo filho; a mutação uniforme com base na probabilidade, de forma que, quanto maior o custo e a dependência de bugs, menor a probabilidade de inversão do gene; e, por fim, a substituição realizada pela técnica de elitismo de 10%, sendo 90% restante realizada utilizando novamente a seleção por torneio para substituir o restante da população.

A discussão desta seção tem como foco dois aspectos da abordagem proposta, que estão ilustrados na Figura 28 nos quadros esquerdo e direito, respectivamente: (i) *geração de uma solução candidata válida*, que considera as restrições impostas pelas dependências de bugs; e (ii) *avaliação de uma solução candidata válida*, com base na função de avaliação proposta e limitada pelo orçamento do projeto de software. É importante destacar que, nem sempre, o orçamento disponível é suficiente para cobrir os custos de correção de todos os bugs, e, por isso, além de avaliar a prioridade e a severidade, é necessário definir um conjunto de bugs que está dentro do orçamento do projeto ao mesmo tempo que respeita as dependências de bugs.

Figura 28 - Recomendação das Tarefas de Correção dos Bugs



Fonte: Lima e Elias (2017).

4.3.1 GERAÇÃO DE UMA SOLUÇÃO CANDIDATA VÁLIDA

Como pode ser observado no quadro esquerdo da Figura 28, a primeira etapa *Selecionar Bugs sem Dependências* é responsável por selecionar aleatoriamente os bugs a serem corrigidos, sem considerar as dependências entre eles. Na abordagem proposta, esta solução inicial é chamada de *Solução sem dependências*, representada pelo cromossomo IX , conforme definido na equação (18), de modo que cada gene ix_i é um ou zero quando o bug b_i é selecionado ou não, respectivamente.

$$IX = \{ix_i \mid \exists_{b_i} b_i \in B \wedge ix_i \in \{0,1\}\} \quad (18)$$

Posteriormente, a segunda etapa *Derivar Bugs com Dependências* tem como objetivo transformar a *Solução sem dependências* IX em uma solução candidata válida que incorpora as restrições das dependências dos bugs. Na abordagem proposta, a solução válida é chamada de *Solução com dependências*, representada pelo cromossomo X , conforme definido na equação (19), de modo que cada gene x_i é um ou zero quando o bug b_i é selecionado ou não, respectivamente. Por exemplo, considere um bug b_i que depende de outro bug b_j , conforme ilustrado na Figura 18. Nesse caso, se o bug b_i for selecionado na *Solução sem dependências* IX , então o bug b_j deve também ser incluído na *Solução com dependências* X .

$$X = \{x_i \mid \exists_{b_i} b_i \in B \wedge x_i \in \{0,1\}\} \quad (19)$$

Para derivar a *Solução com dependências* X , este trabalho trata como matrizes booleanas o conjunto de soluções básicas BX e a *Solução sem dependências* IX , definidas nas equações (7) e (18), respectivamente. Desta forma, a *Solução com dependências* X pode ser derivada diretamente pela multiplicação booleana entre IX e BX , conforme apresentado na equação (20).

$$X = IX \cdot BX_{B \times B} \quad (20)$$

Assim, toda e qualquer solução IX que for multiplicada pelas soluções básicas BX produz como resultado a matriz de soluções válidas X , que contém bugs selecionados com seus respectivos bugs dependentes.

Para concluir a geração de uma solução válida, somando o custo específico v_i para cada bug selecionado x_i , a terceira etapa, denominada *Estimar Custo de Correção dos Bugs*, calcula o valor total para todas as tarefas de correção dos bugs relacionadas aos bugs selecionados. Aqui, esse valor é chamado de *Custo de correção dos bugs*, que é representado pelo termo VS_X , definido na equação (21).

$$VS_X = \sum_{b_i \in B} v_i \cdot x_i \quad (21)$$

4.3.2 AVALIAÇÃO DE UMA SOLUÇÃO CANDIDATA VÁLIDA

Depois de gerar uma solução com dependências válidas, como pode ser percebido no quadro direito da Figura 28, as etapas superiores *Avaliar Grau de Cobertura dos Bugs* e *Avaliar Grau de Interdependência dos Bugs* podem ser realizadas simultaneamente. Por um lado, na primeira etapa, o *Grau de Cobertura*, representado pelo termo CD_X e definido na equação (22), descreve a porcentagem de bugs x_i abrangidos pela *Solução com dependências* X em relação ao conjunto inteiro dos bugs candidatos B , ou seja, mostra a quantidade de bugs selecionados em relação a quantidade total de bugs. Sendo assim, quanto maior for a quantidade de bugs selecionados, maior será o *Grau de Cobertura* da solução. Por outro lado, na segunda etapa, *Avaliar Grau de Interdependência dos Bugs*, definida usando o termo ID_X indicado na equação (23), mostra a proporção de caminhos cobertos pela *Solução com dependências* em relação ao conjunto total de caminhos do grafo direcionado definido pelas dependências diretas e indiretas dos bugs nas *Soluções Básicas* $BX_{B \times B}$.

$$CD_X = \frac{\sum_{b_i \in B} x_i}{|B|} \quad (22)$$

$$ID_X = \frac{\sum_{b_i \in B} \sum_{b_j \in B} bx_{i,j} \cdot x_i}{\sum_{b_i \in B} \sum_{b_j \in B} bx_{i,j}} \quad (23)$$

Neste ponto, como representado no quadro direito da Figura 28, as etapas intermediárias denominadas *Avaliar Prioridade dos Bugs* e *Avaliar Severidade dos Bugs* são responsáveis por estimar os *Graus de Prioridade* e *Severidade* associados a todos os bugs incluídos na *Solução com dependências X*, que são os principais termos que compõem a função de avaliação. Conforme definido na equação (24), o *Grau de Prioridade* P_X é estimado com base no valor médio ponderado da *Prioridade do Produtor* pp_i e da *Prioridade dos Clientes* cp_i para cada bug b_i selecionado, refatorado pelo *Grau de Cobertura* CD_X . Assim, se duas soluções com dependências tiverem a mesma prioridade média ponderada, aquela com maior *Grau de Cobertura* seleciona mais bugs para correção, e, portanto, é favorecida na abordagem proposta.

$$P_X = CD_X \cdot \frac{wp_p \cdot \sum_{b_i \in B} pp_i \cdot x_i + wp_c \cdot \sum_{b_i \in B} cp_i \cdot x_i}{|B|} \quad (24)$$

De forma semelhante, de acordo com a equação (25), o *Grau de Severidade* S_X é estimado com base no valor médio ponderado da *Severidade do Produtor* ps_i e da *Severidade dos Clientes* cs_i para cada bug b_i selecionado, refatorado pelo *Grau de Interdependência* dos bugs ID_X . Assim, por exemplo, se duas soluções com dependências tiverem a mesma severidade média ponderada, aquela com maior *Grau de Interdependência* seleciona mais bugs críticos para correção, e, assim, é preferida na abordagem proposta.

$$S_X = ID_X \cdot \frac{ws_p \cdot \sum_{b_i \in B} ps_i \cdot x_i + ws_c \cdot \sum_{b_i \in B} cs_i \cdot x_i}{|B|} \quad (25)$$

Observe que, segundo as equações (24) e (25), os *Graus de Prioridade e Severidade* podem assumir valores no intervalo $[0,1] \in \mathbb{R}$. Além disso, os termos wp e ws representam pesos normalizados, $(wp_p + wp_c = 1)$ e $(ws_p + ws_c = 1)$, aplicados para representar a contribuição efetiva de produtores e clientes nos *Graus de Prioridade e de Severidade*.

Finalmente, como indicado no quadro direito da Figura 28, a última etapa denominada *Avaliar Bugs Selecionados* tem como objetivo estimar a aptidão ou adequação da solução, com base na função de avaliação proposta e limitada pelo *Orçamento do Projeto PB*. Como pode ser visto na equação (26), a função de avaliação F_X se caracteriza por ser uma função de maximização e estima a adequabilidade de seleção dos bugs de acordo com o valor médio ponderado do *Grau de Prioridade* P_X e do *Grau de Severidade* S_X , assumindo valores no intervalo $[0,1] \in \mathbb{R}$. Além disso, o termo wf representa um peso normalizado, ou seja $(wf_p + wf_s = 1)$, que é aplicado para representar a contribuição efetiva dos *Graus de Prioridade e Severidade* na adequabilidade da solução.

$$F_X = \begin{cases} wf_P \cdot P_X + wf_S \cdot S_X & \text{se } VS_X \leq PB \\ 0 & \text{caso contrário} \end{cases} \quad (26)$$

É importante ressaltar que, se o *Custo de Correção dos Bugs* VS_X for maior que o *Orçamento do Projeto* PB , a abordagem proposta penaliza a solução candidata com dependência (X), atribuindo zero ao valor da função de avaliação. De fato, tal penalidade reduz drasticamente a probabilidade de soluções mais dispendiosas influenciarem a evolução da população, fornecendo boas soluções que não excedam o orçamento do projeto de software.

4.4 CONSIDERAÇÕES FINAIS

O método para encontrar as dependências de bugs baseadas em componentes de forma automática, foi criada para minimizar o esforço do gerente de projetos e com tempo de processamento do algoritmo genético reduzido e aceitável, pelo fato da metodologia eliminar a complexidade inerente nas operações necessárias para considerar as dependências dos bugs. Mas para isso é necessário que os dados acerca dos componentes estejam mapeados nos relatórios de bugs, bem como o sistema possuir uma matriz de dependência direta de componentes para representar o comportamento do diagrama de componentes e a associação do bug ao componente.

Seguem então, algumas vantagens decorrentes da reparação do indivíduo através das dependências dos bugs como forma de preparação de dados: (i) *facilidade no uso dos operadores (cruzamento e mutação)*, ao utilizar os operadores do algoritmo em sua forma original, sem necessidade de adaptações a cada operação para gerar soluções válidas; (ii) *menor custo de processamento*, devido ao uso da matriz de dependências de bugs, que evita a necessidade de realizar correções de funções de reparação, que são complexas e geram um alto custo de processamento, pois até que trouxesse uma solução válida, haveria um alto número de iterações que seriam executadas até corrigir todas as dependências.

Por fim, a formulação do problema PSB na abordagem proposta considera uma função objetivo cujo valor depende da prioridade e severidade dos bugs, atribuídas pelo produtor de software e seu conjunto de clientes, corrigindo os bugs selecionados para a próxima versão do produto de software. Como tal, o objetivo do problema PSB é selecionar o subconjunto de bugs relatados que maximizem o valor da função anteriormente mencionada, mas com custo inferior ao orçamento disponível no projeto.

5 ESTUDO DE CASO

A abordagem proposta foi avaliada usando um conjunto de dados reais extraídos dos relatórios de bugs do *Sistema Integrado de Gestão de Atividades Acadêmicas* (SIGAA-GRADUAÇÃO) adotado na Universidade Federal da Paraíba (UFPB). O SIGAA foi desenvolvido pela Universidade Federal do Rio Grande do Norte (UFRN), mas está presente em várias instituições de ensino, incluindo a UFPB, que realiza a manutenção e customização de acordo com suas necessidades específicas. O produto de software avaliado possui vários módulos com funcionalidades bem encapsuladas. No presente estudo de caso, foi considerado o módulo de graduação, que contempla 227 componentes de software, representados por classes em *java*, dentre um total de mais de 5 mil classes, se fosse considerar todos os módulos, além de graduação. Os relatórios de bugs considerados neste estudo de caso foram registrados pela própria equipe de desenvolvimento e testes.

O conjunto de dados empregado neste estudo de caso vem de dois lotes de bugs relatados no BTS institucional, com 20 bugs para o primeiro lote, que define um espaço de busca de 1.048.576 (2^{20}) soluções candidatas, e com 50 bugs para o segundo lote, definindo um espaço de busca de 1.125.899.906.842.620 (2^{50}). Observe que o espaço de busca de 50 bugs é 1.073.741.824 de vezes maior que o espaço de busca de 20 bugs, ou seja, um pouco mais de 1 bilhão de vezes em comparação ao anterior. Exemplificando a intratabilidade do problema, para um computador selecionar os bugs mais importantes dentro de um conjunto de 50 bugs, para um dado orçamento, o mesmo passaria em torno de 3.834 anos para encontrar a solução ótima ou melhor solução, o que implica que o problema é NP-difícil, sendo operacionalmente inviável realizar cálculos exaustivos para esta quantidade de bugs.

Portanto, o estudo de caso é dividido em dois cenários: 20 e 50 bugs. Para cada cenário, a abordagem proposta é avaliada em um conjunto de experimentos, que variando o orçamento disponível, avaliam e selecionam um subconjunto de bugs apropriados, cujo objetivo é reduzir o *backlog* dos bugs relatados do BTS. Manualmente, caso não existisse a abordagem proposta, o gerente de projeto deveria planejar a correção dos bugs sem exceder um orçamento de projeto de software ou, para o caso concreto, sem exceder o tempo de desenvolvimento para o *backlog*, e estimar o custo ou o esforço de correção de bugs necessários usando o julgamento de especialistas.

Em resumo, em vez de considerar um determinado orçamento do projeto como meio de avaliar o desempenho da abordagem proposta correspondente aos diferentes orçamentos dos

projetos, a avaliação de cada cenário (20 e 50 bugs) é baseada em 14 experimentos, cada um adotando um nível de orçamento do projeto que representa 5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95 e 100% do custo total do projeto, que é estimado somando o custo de todos os bugs relatados.

Os dados detalhados sobre bugs, componentes e clientes não estão relacionados no trabalho pois entende-se que não é necessário expor dados institucionais relacionados aos bugs do sistema, pois isso poderia de certa forma expor alguma possível vulnerabilidade.

Vale ressaltar que os experimentos realizados neste estudo de caso foram executados em um notebook com processador i7-7700 HQ, com frequência de 2.8 GHz em um sistema Windows 10 de 64 bits e com 8 GB de memória RAM.

5.1 IMPLEMENTAÇÃO E CALIBRAÇÃO

Para cada cenário, os experimentos foram realizados em duas etapas distintas: (i) avaliação automática baseada na abordagem proposta; e (ii) avaliação manual baseada no julgamento de especialistas através de uma equipe de analistas. Na primeira etapa, para avaliar a abordagem proposta¹ foram implementadas três estratégias de busca: (i) busca genética; (ii) busca aleatória; e (iii) busca exaustiva. Em seguida, na segunda etapa, com o intuito de avaliar a competitividade humana da abordagem proposta, foi realizado um experimento baseado no julgamento de especialistas.

No entanto, considerando a complexidade, esforço e variabilidade dos resultados obtidos em cada execução de cada experimento, as diferentes estratégias de busca e o julgamento de especialistas não foram conduzidos igualmente em todos os cenários, experimentos e orçamentos. A Tabela 3 ilustra a organização dos experimentos em cada cenário, execuções e orçamentos.

Tabela 3 - Organização dos experimentos

	Cenário	Execuções	Orçamento
Busca Genética	20 / 50	1000	5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95 e 100%
Busca Aleatória	20 / 50	1000	5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95 e 100%
Busca Exaustiva	20	1	5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95 e 100%
Julgamento de Especialista	20	11	50%

¹ Desenvolvida em Java e IDE Eclipse.

Para as estratégias de busca genética e aleatória, considerando os componentes aleatórios presentes nos respectivos algoritmos de busca, a implementação permite que o experimento seja executado várias vezes, de acordo com um parâmetro configurável, calculando automaticamente o valor médio das melhores soluções recomendadas em todas as execuções. No estudo de caso, para melhorar a confiabilidade dos resultados obtidos, em ambos cenários, cada experimento é executado 1000 vezes para as estratégias de busca genética e aleatória em todos os orçamentos avaliados.

Quanto a estratégia de busca exaustiva, cada experimento é executado uma única vez, pois não existem componentes aleatórios no respectivo algoritmo, e, portanto, não existem diferenças nos resultados obtidos em diferentes execuções. Além disso, considerando o tamanho dos espaços de busca dos dois cenários (20 e 50 bugs), a busca exaustiva foi realizada apenas para os experimentos do primeiro cenário de 20 bugs, porém contemplando todos os orçamentos avaliados.

Já na avaliação baseada no julgamento de especialistas, considerando o enorme esforço requerido para realizar a seleção manual dos bugs relatados, o experimento realizado contempla apenas o cenário de 20 bugs com orçamento de 50%. É importante destacar que cada especialista participante do estudo de caso realiza individualmente a seleção dos bugs. Portanto, considerando que o estudo de caso contempla 11 especialistas, pode-se inferir que a estratégia baseada em julgamento de especialistas foi realizada 11 vezes.

Na estratégia de busca genética, a população inicial é gerada usando um esquema probabilístico baseado no custo e nas dependências dos bugs. No esquema adotado, a probabilidade de um dado bug ser selecionado é inversamente proporcional ao seu custo e grau de dependência. Portanto, quanto maior o custo de correção do bug e a quantidade de dependências em relação a outros bugs, menor a probabilidade de ser incluído em uma solução da população inicial. Além disso, considerando a restrição orçamentária, bugs com custo maior que o orçamento disponível são descartados da seleção, ou seja, a probabilidade de ser incluído em uma solução da população inicial é zero.

Em seguida, a população é melhorada através de um processo geracional, com base na aplicação repetitiva dos operadores genéticos de seleção, cruzamento, mutação e substituição. Na abordagem proposta, o operador de seleção é baseado na *seleção por torneio*, cujo princípio de funcionamento consiste em escolher aleatoriamente duas soluções (pais) da geração atual, e, posteriormente, selecionar aquela com maior valor da função objetivo, conforme já discutido em 2.2.1 a. No entanto, é importante lembrar que a abordagem proposta penaliza soluções

com custo superior ao orçamento, atribuindo o valor zero as suas respectivas funções objetivo. Neste caso, se as duas soluções (pais) escolhidas possuem custo acima do orçamento, a seleção por torneio escolhe aquela com menor custo.

No que se refere ao operador de cruzamento, a abordagem proposta adota o *cruzamento uniforme baseado em máscara*, que consiste na geração de duas novas soluções (filhas) a partir da combinação de características de duas soluções (pais) da população atual, conforme já apresentado em 2.2.1 b.

Após a geração das novas soluções (filhas), o operador de mutação deve ser aplicado para assegurar diversidade populacional. Na abordagem proposta, o operador clássico de *mutação binária uniforme* é adotado, conforme explicado em 2.2.1 b. No entanto, observe que as dependências de bugs representam uma forma indireta de seleção, ou seja, selecionar um bug que depende de outros bugs também implica em incluir na solução estes outros bugs. Consequentemente, na abordagem proposta, a probabilidade de mutação de cada bug é calculada de forma inversamente proporcional ao seu grau de dependência. Ou seja, quanto maior a quantidade de dependências em relação a outros bugs, menor a probabilidade de mutação do bug. Vale destacar que bugs com custo acima do orçamento nunca são selecionados, e, além disso, jamais são considerados pelo operador de mutação.

Neste ponto, considerando a população atual e as novas soluções (filhas) geradas pelos operadores de cruzamento e mutação, o operador de substituição deve ser aplicado para gerar a nova população. Na abordagem proposta, a substituição *com elitismo* é adotada, conforme já discutido em 2.2.1 c. aplicando um índice de elitismo de 10%, inicialmente, a nova população é formada com 10% das melhores soluções que constituem a população atual e as novas soluções (filhas) geradas. Em seguida, o restante da nova população (90%) é escolhido com base na mesma estratégia de seleção por torneio adotada para geração da população inicial.

Considerando a busca genética, além do número de execuções, a implementação permite configurar o tamanho da população e a quantidade de gerações, esta última representando a condição de parada do algoritmo genético. A Tabela 4 apresenta os valores adotados para o tamanho da população, bem como a quantidade de gerações para cada cenário, evidenciando também a quantidade de soluções totais avaliadas em cada execução, que nada mais é que o tamanho da população multiplicado pela quantidade de gerações. Como pode ser observado, o tamanho da população foi calibrado como sendo o dobro do número de bugs do cenário em questão. Já a quantidade de gerações foi calibrada como sendo o dobro do tamanho da população.

Tendo em vista a busca aleatória, foram testados a mesma quantidade de indivíduos considerados no algoritmo genético, contando com 3.200 soluções avaliadas para o primeiro cenário de 20 bugs, e 20.000 soluções avaliadas para o segundo cenário de 50 bugs. Desta forma, tanto a busca genética quanto a busca aleatória cobriram apenas cerca de 0,3% e 0,000000002% do espaço de busca dos cenários de 20 e 50 bugs, respectivamente.

Tabela 4 - Configuração do algoritmo genético

Cenário	Tamanho da população	Quantidade de gerações	Soluções avaliadas
20 bugs	40	80	3200
50 bugs	100	200	20000

Concluindo a calibração, os pesos dos critérios de avaliação para estimar o grau de prioridade (w_p), grau de severidade (w_s) foram normalizados adotando o valor 0,5 tanto para o lado do produtor (w_{p_p} e w_{s_p}) quanto para o lado do cliente (w_{p_c} e w_{s_c}), representando assim uma contribuição igual para ambos no cálculo do grau de prioridade (P_X) e do grau de severidade (S_X), conforme representado nas equações (24) e (25), respectivamente. É importante destacar que, caso seja de interesse do gerente de projetos, é possível eliminar a contribuição dos clientes, bastando para tal adotar o peso 1,0 para o produtor e 0,0 para os clientes. Além disso, os pesos dos critérios de avaliação para estimar a função objetivo (w_f) também foram normalizados com valor 0,5, representando uma contribuição igual para o grau de prioridade (w_{f_p}) e o grau de severidade (w_{f_s}) no cálculo da função de avaliação (F_X), conforme representado na equação (26).

Vale ressaltar que a calibração de todos os parâmetros foi realizada a partir de diversas execuções experimentais, mas, caso seja de interesse, o gerente de projetos pode parametrizar a busca genética com valores mais apropriados a realidade do projeto de software em questão ou mesmo considerar aspectos específicos da organização desenvolvedora. Por exemplo, caso desejado, o gerente de projetos pode desconsiderar a participação dos clientes atribuindo os valores um e zero aos pesos relacionados ao produtor e aos clientes, respectivamente.

A estratégia de busca baseada no julgamento de especialistas foi realizada por 11 analistas de sistemas, que trabalham no desenvolvimento dos módulos do SIGAA, alguns tendo participado do processo de desenvolvimento e correção de bugs do referido módulo, o que se configura em escolhas realizadas com base no conhecimento e experiência de profissionais envolvidos diretamente na manutenção e evolução do sistema sob avaliação.

Para conduzir e analisar a seleção de bugs recomendada por cada especialista, um formulário foi concebido, contemplando algumas perguntas relativas aos seguintes aspectos: (i) *caracterização pessoal* - identifica o perfil profissional do especialista envolvido no experimento; (ii) *seleção de bugs* – enumera o conjunto de bugs selecionados pelo especialista; e (iii) *quadro de justificativas* – elenca os motivos que levaram o especialista a selecionar os referidos bugs.

No que diz respeito ao aspecto de caracterização pessoal, por um lado, conforme mostrado na Figura 29, cerca de 72,7% dos participantes são analistas que possuem mais de 3 anos de experiência em desenvolvimento ou testes de software, 9,1% com 1 a 3 anos de experiência e 18,2% com até 1 ano de experiência. Por outro lado, conforme mostrado na Figura 30, em torno de 81,8% dos analistas participantes já trabalharam em atividades relacionadas diretamente com o módulo de graduação do SIGAA, enquanto apenas cerca de 18,2% não exerceram qualquer atividade com o referido módulo. Consequentemente, como pode ser observado, a grande maioria dos especialistas envolvidos no experimento baseado no julgamento de especialistas está diretamente relacionado com atividades típicas do desenvolvimento e teste de software do módulo de graduação do SIGAA, fato que favorece para avaliar a competitividade humana da abordagem proposta em virtude da maturidade e experiência dos analistas envolvidos.

Figura 29 - Experiência em desenvolvimento ou testes de software

Qual o seu tempo de experiência trabalhando em desenvolvimento ou testes de software?

11 respostas

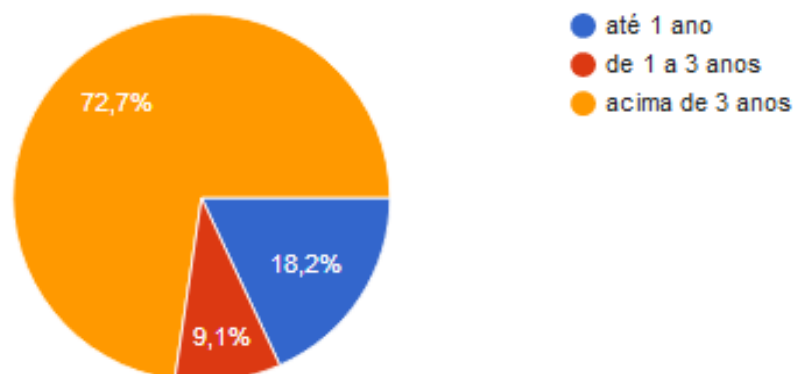
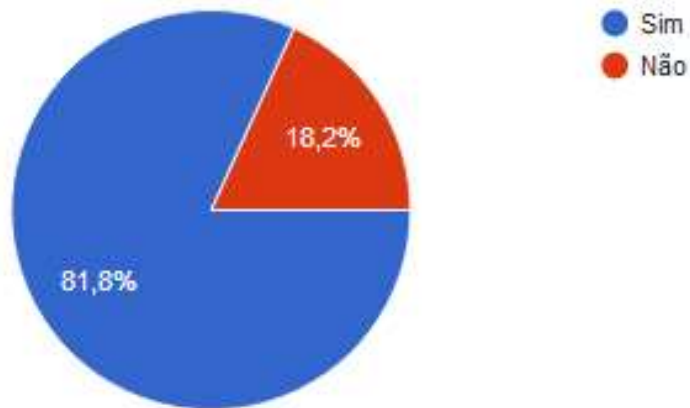


Figura 30 - Experiência no módulo de graduação da UFPB
Já trabalhou no módulo de Graduação da UFPB?
 11 respostas



5.2 RESULTADOS E AVALIAÇÃO COMPARATIVA

Para o primeiro cenário com 20 bugs, a busca genética foi comparada inicialmente com a busca aleatória e a busca exaustiva, ou seja, as abordagens automáticas foram comparadas entre si. No segundo cenário com 50 bugs, a busca genética foi comparada apenas com a busca aleatória, pois tornou-se computacionalmente inviável realizar a busca exaustiva no gigantesco espaço de busca deste cenário. Observe que, em ambos cenários, a busca aleatória foi adotada como uma forma de estimador de referência ou teste de racionalidade, do inglês *sanity check*, conforme sugerido por Harman (2007).

Em seguida, considerando apenas o cenário com 20 bugs, a competitividade humana da abordagem proposta foi avaliada realizando a comparação dos resultados da busca genética com a busca baseada no julgamento de especialistas. Vale ressaltar que, no segundo cenário com 50 bugs, a competitividade humana não foi avaliada pois tornou-se inviável afastar mais uma vez os analistas de suas atividades profissionais para alocá-los em um novo experimento bem mais complexo que demandaria muito mais esforço e tempo.

5.2.1 AGENDAMENTO E ALOCAÇÃO MONO-OBJETIVA DE BUGS

No primeiro cenário com 20 bugs, inicialmente, a busca genética e a busca aleatória foram executadas 1000 vezes para cada restrição de orçamento, conforme já sinalizado na Tabela 3. Após a realização de cada experimento, o valor médio da função de avaliação da melhor solução encontrada em cada execução foi calculado. Em seguida, considerando o

reduzido espaço de busca definido pelos 20 bugs, a busca exaustiva foi executada apenas uma única vez para descobrir a solução ótima ou melhor solução.

Conforme mostrado na Figura 31, a comparação dos resultados da busca genética com a busca aleatória demonstra que, em todos os experimentos, a busca genética é sempre melhor do que a busca aleatória. Observe na Tabela 5 que, considerando as restrições orçamentárias até 25%, a solução recomendada pela busca genética é pelo menos 129% melhor do que a solução da busca aleatória. No caso específico de 5% do orçamento, esse ganho ultrapassa 5000%. A partir da restrição orçamentária de 30%, o ganho da busca genética começa a ser reduzido para patamares abaixo de 100%, mas nunca se tornando inferior a 28%. Neste cenário com 20 bugs, os resultados obtidos evidenciam a eficiência da busca genética, certamente justificada pela eficácia dos operadores genéticos em guiar a busca em direção a boas soluções.

Figura 31 - Comparação da Busca Genética e Busca Aleatória para 20 bugs

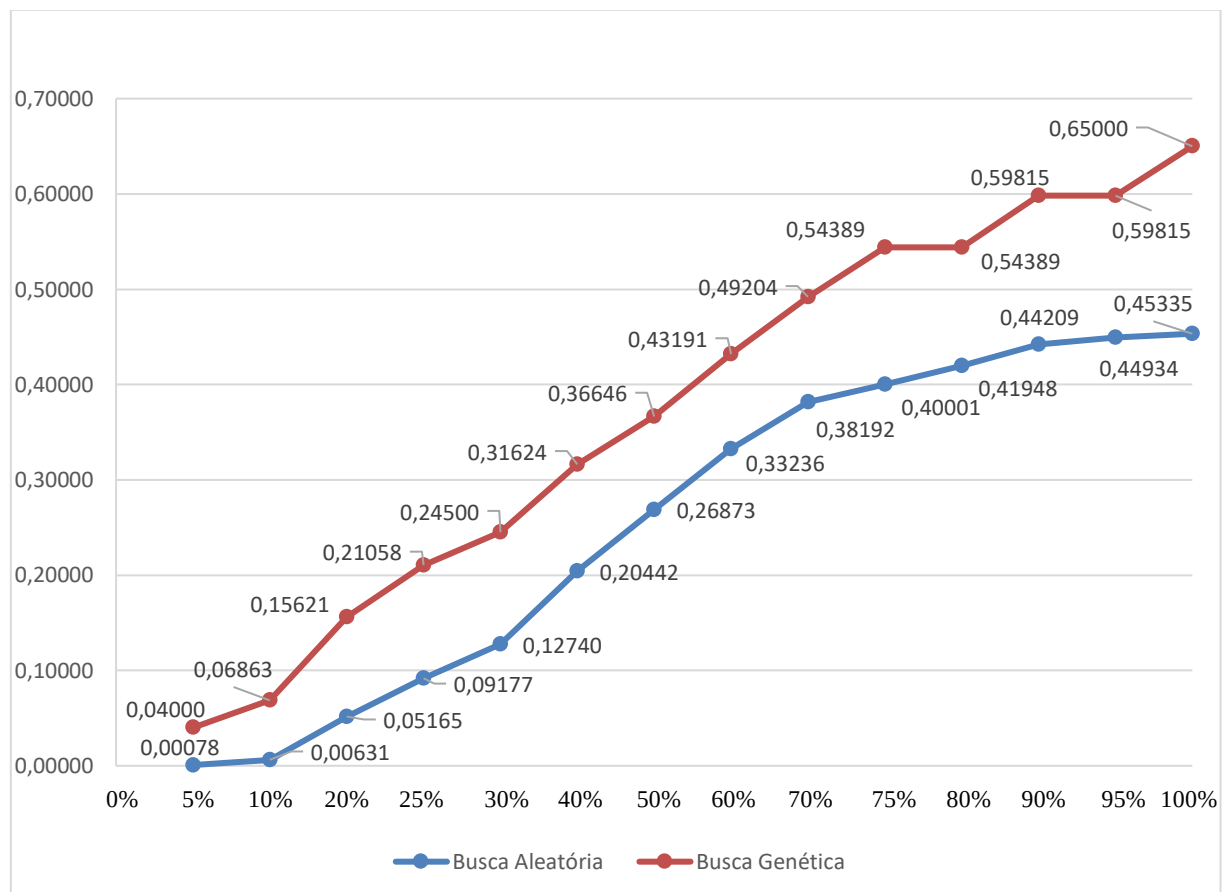


Tabela 5 – Resultados da Busca Genética e Busca Aleatória para 20 bugs

Orçamento	Busca Genética	Busca Aleatória	Ganho
5%	0,04000	0,00078	5028,21%
10%	0,06863	0,00631	987,64%
20%	0,15621	0,05165	202,44%
25%	0,21058	0,09177	129,46%
30%	0,24500	0,12740	92,31%
40%	0,31624	0,20442	54,70%
50%	0,36646	0,26873	36,37%
60%	0,43191	0,33236	29,95%
70%	0,49204	0,38192	28,83%
75%	0,54389	0,40001	35,97%
80%	0,54389	0,41948	29,66%
90%	0,59815	0,44209	35,30%
95%	0,59815	0,44934	33,12%
100%	0,65000	0,45335	43,38%

Considerando que o cenário com 20 bugs possui um espaço de busca relativamente pequeno, neste cenário também foi possível realizar a comparação das estratégias de busca genética e busca exaustiva. Os resultados obtidos revelam que, para todas os valores de restrições orçamentárias avaliadas, para todos os experimentos e na grande maioria das 1000 execuções, a busca genética foi capaz de encontrar a solução ótima, descobertas e confirmadas pela busca exaustiva.

De forma mais detalhada, a análise comparativa da busca genética em relação a busca exaustiva pode ser vista na Tabela 6. Como pode ser observado, exceto para as restrições orçamentárias de 20%, 40% e 60%, todas as execuções da busca genética encontraram a solução ótima sinalizada pela busca exaustiva. Ou seja, o valor médio obtido nas 1000 execuções da busca genética de cada experimento é exatamente igual ao valor ótimo da busca exaustiva.

No entanto, apenas para os orçamentos de 20%, 40% e 60%, a busca genética teve médias abaixo do valor ótimo da busca exaustiva. Os valores médios para estes 3 casos na busca genética foram 0,15621, 0,31624, e 0,43191, ao passo que a busca exaustiva obteve os valores ótimos 0,15639, 0,31633, e 0,43200, respectivamente. Ou seja, em relação à busca genética, o ganho da busca exaustiva foi de apenas 0,12%, 0,03% e 0,02%, respectivamente. Dito de outra forma, em relação à busca exaustiva, a eficiência da busca genética foi de 99,88%, 99,97%, e 99,98%, respectivamente. Vale ressaltar que em todos os demais percentuais orçamentários,

tanto a busca genética quanto a busca exaustiva obtiveram os mesmos valores, representando assim uma eficiência de 100%.

Tabela 6 - Resultados da Busca Genética e Busca Exaustiva para 20 bugs

Orçamento	Busca Genética	Busca Exaustiva	Ganho	Eficiência
5%	0,04000	0,04000	0,00%	100,00%
10%	0,06863	0,06863	0,00%	100,00%
20%	0,15621	0,15639	-0,12%	99,88%
25%	0,21058	0,21058	0,00%	100,00%
30%	0,24500	0,24500	0,00%	100,00%
40%	0,31624	0,31633	-0,03%	99,97%
50%	0,36646	0,36646	0,00%	100,00%
60%	0,43191	0,43200	-0,02%	99,98%
70%	0,49204	0,49204	0,00%	100,00%
75%	0,54389	0,54389	0,00%	100,00%
80%	0,54389	0,54389	0,00%	100,00%
90%	0,59815	0,59815	0,00%	100,00%
95%	0,59815	0,59815	0,00%	100,00%
100%	0,65000	0,65000	0,00%	100,00%

Novamente, os resultados obtidos evidenciam a eficiência da busca genética, que, conforme já citado anteriormente, é justificada pela eficácia dos operadores genéticos em guiar a busca em direção a boas soluções, tanto que, neste cenário com 20 bugs que apresenta um espaço de busca relativamente pequeno, a busca genética obteve eficiência praticamente igual a busca exaustiva.

Como outro resultado interessante, deve-se destacar o baixo custo de processamento da abordagem proposta, que pode ser percebida por sua rápida convergência. Para o conjunto de 20 bugs, como mostrado na Tabela 7, a busca genética alcançou os resultados apresentados com apenas 80 gerações, que convertido em tempo de execução chega a aproximadamente 0,56 segundos. Já o algoritmo aleatório, na mesma situação, tem um tempo de execução de cerca de 0,44 segundos, pois simplesmente gera soluções aleatórias sem o custo de processamento dos operadores genéticos. Em contrapartida, embora a busca aleatória apresente um tempo de execução cerca de 21,16% melhor em relação à busca genética, o ganho da busca genética variou do melhor para o pior caso de 5028% a 28,83%, alcançando um ganho médio geral de 483%, e, considerando os 10 piores casos da busca genética, o ganho médio obtido foi de 36,36%, sendo portando plenamente aceitável a permuta de maior consumo de processamento por melhor qualidade dos resultados.

É importante destacar que o consumo de processamento da busca genética torna-se totalmente irrelevante quando comparado com o consumo da busca exaustiva. Como também pode ser observado na Tabela 7, enquanto a busca genética foi realizada em apenas 0,56 segundos, a busca exaustiva requereu cerca de 113 segundos (aproximadamente 2 minutos), fato que torna a busca genética cerca de 200 vezes mais rápida que a busca exaustiva, e, como ilustrado na Tabela 6, com eficiência praticamente idêntica.

Tabela 7 - Tempo de execução do cenário de 20 bugs

Tempo de execução	Busca Genética	Busca Aleatória	Busca Exaustiva
Tempo de Execução	0,56 s	0,44 s	113 s

5.2.2 ABORDAGENS AUTOMÁTICAS NO CENÁRIO COM 50 BUGS

Considerando os excelentes resultados obtidos no cenário com 20 bugs, a próxima etapa foi avaliar a abordagem proposta em um cenário com espaço de busca gigantesco, onde seria inviável a busca exaustiva. Nesse sentido, o segundo cenário foi construído para realizar a seleção dentre 50 bugs relatados, comparando a abordagem proposta baseada na busca genética apenas com a busca aleatória, adotada como forma de obter uma estimativa de referência para avaliar a qualidade das soluções encontradas pela abordagem genética proposta.

Em todos os experimentos realizados no cenário com 50 bugs, tanto a busca genética quanto a busca aleatória também foram repetidas em 1000 execuções, conforme já sinalizado na Tabela 3. Os resultados obtidos revelam que, para todos os valores de restrições orçamentárias avaliadas, exceto para a restrição orçamentária de 30%, em todas as 1000 execuções, a busca genética foi sempre capaz de encontrar a mesma solução. Ou seja, nestes casos, o valor médio calculado é exatamente igual ao valor máximo encontrado em cada execução. Mesmo no caso do orçamento de 30%, a busca genética encontrou a mesma solução máxima em 983 das 1000 execuções, alcançando um valor médio de 0,10391 e um valor máximo de 0,10396, ou seja, o valor médio representa 99,95% do valor máximo encontrado.

Vale ressaltar que os valores máximos encontrados não necessariamente são as soluções ótimas, pois, como já sinalizado, neste cenário com 50 bugs é inviável executar o algoritmo exaustivo para descobrir as soluções ótimas de cada restrição orçamentária. Para estimar o tempo de execução da busca exaustiva, considere o cenário com 20 bugs, espaço de busca com 2^{20} soluções e tempo de execução de 113 segundos. De forma aproximada, pode-se concluir

que, no cenário com 50 bugs e espaço de busca com 2^{50} soluções, o tempo de execução da busca exaustiva seria em torno de 3.834 anos.

Tabela 8 – Resultados da Busca Genética e Busca Aleatória para 50 bugs

Orçamento	Busca Genética	Busca Aleatória
5%	0,00931	0,00000
10%	0,02661	0,00000
20%	0,06189	0,00000
25%	0,08234	0,00000
30%	0,10391	0,00000
40%	0,18344	0,00000
50%	0,24057	0,00000
60%	0,37100	0,00000
70%	0,47578	0,00000
75%	0,51295	0,00000
80%	0,54458	0,00000
90%	0,58116	0,00000
95%	0,59651	0,00000
100%	0,61500	0,00000

Outra informação muito importante, revelada na Tabela 8, diz respeito aos resultados obtidos nas execuções da busca aleatória. Como pode ser observado, para todas as restrições orçamentárias e para todas as execuções, a busca aleatória não foi capaz de encontrar nenhuma solução com custo abaixo do orçamento em questão ou que respeitasse as restrições das dependências dos bugs. Novamente, estes resultados evidenciam de forma mais intensa a eficiência da busca genética, certamente justificada pela eficácia dos operadores genéticos em guiar a busca em direção a boas soluções.

Como outro resultado interessante, deve-se destacar o baixo custo de processamento da abordagem proposta, que pode ser percebida por sua rápida convergência. Para o conjunto de 50 bugs, como mostrado na Tabela 8, a busca genética alcançou os resultados apresentados com apenas 200 gerações, que convertido em tempo de execução chega a 5,97 segundos. Já o algoritmo aleatório, na mesma situação, tem um tempo de execução de cerca de 4,49 segundos, pois simplesmente gera soluções aleatórias sem o custo de processamento dos operadores genéticos. Em contrapartida, embora a busca aleatória apresente um tempo de execução cerca de 24,82% melhor em relação à busca genética, o ganho da busca genética foi superior em todos os casos trazendo soluções quanto a busca aleatória que não trouxe nenhuma solução válida,

sendo portando plenamente aceitável a permuta de maior consumo de processamento por possibilitar o alcance dos resultados.

5.2.3 ABORDAGEM BASEADA NO JULGAMENTO DE ESPECIALISTAS

Considerando os excelentes resultados obtidos pela abordagem genética proposta nesta dissertação, em um segundo momento, foi realizado um estudo comparativo entre a busca genética e a busca baseada no julgamento de especialista, desta vez com o propósito de avaliar a competitividade humana da abordagem proposta. Como já explicado, em virtude da complexidade e consequente esforço requerido para o cenário com 50 bugs, a busca baseada no julgamento de especialistas foi realizada apenas para o cenário com 20 bugs.

Neste cenário com 20 bugs, apesar de mais simples, nenhuma das soluções recomendadas pelos 11 analistas participantes era válida. Em resumo, dentre as 11 soluções recomendadas, três analistas recomendaram soluções com custo acima do orçamento, enquanto os demais não conseguiram identificar soluções que atendessem as restrições relativas às dependências de bugs.

Consequentemente, os resultados mostram claramente a complexidade de apontar soluções baseadas no julgamento de especialistas para o problema da seleção de bugs, reforçando ainda mais a necessidade de abordagens automáticas para resolver o problema. E, nesta direção, evidencia a competitividade humana da abordagem proposta.

Em adição, dada a qualidade das soluções recomendadas pela abordagem proposta, podemos concluir que a busca genética não só recomenda soluções dentro do orçamento como também indica soluções que atendem plenamente as dependências dos bugs. E, mais importante, conforme mostram os valores resultantes das funções de avaliação, as soluções da busca genética superam eficientemente aquelas da busca aleatória e até mesmo igualam-se aquelas da busca exaustiva, onde foi possível fazer a comparação.

A partir dos formulários preenchidos pelos analistas, foi possível identificar alguns pontos que os mesmos consideram relevante no problema de seleção de bugs. Em relação às dependências de bugs, 5 dos 11 analistas consideram difícil identificar estas dependências. Claramente, considerando os resultados obtidos no experimento baseado no julgamento de especialistas, fica evidente que a dificuldade de identificar as dependências é realmente muito presente tendo em vista que nenhum dos analistas foi capaz de apontar uma solução que atendesse plenamente as restrições ligadas às dependências dos bugs.

Já considerando a prioridade, a severidade, o custo e as dependências de bugs, o critério indicado como de maior importância na análise dos bugs foi a prioridade, sendo escolhido por 5 dos 11 analistas, seguido dos outros critérios com 3 votos para as dependências de bugs, 2 votos para a severidade e apenas 1 voto para o custo.

Ainda sobre dependência de bugs, foram vistas algumas analogias dela com a dependência de requisitos, bem como sua importância na engenharia de software para o problema da próxima *release*. Mas aqui pode-se ter uma evidência da importância da dependência de bugs em termos práticos com um exemplo vivenciado no sistema SIGAA, objeto deste estudo.

Diversas vezes que o componente “dados pessoais” foi atualizado, no trecho de código que trata a forma de alimentar a nacionalidade do discente, provocava um efeito negativo e danoso no componente “impressão de diplomas”, pois este último depende do primeiro e em momentos que o bug da nacionalidade era corrigido na impressão do diploma, sempre era quebrado (provocava o retorno do erro) quando havia alguma alteração em dados pessoais no mesmo sentido e com isso havia um retrabalho para correção de algo que já estava corrigido, trazendo uma ineficiência a equipe, quando verificava-se que as correções em diploma primeiramente não eram eficazes, pois o erro principal estava na forma que os dados pessoais eram alimentados. Desta forma, quando o problema foi corrigido em dados pessoais e o diploma logo após, o problema foi solucionado de uma só vez.

Voltando à análise do estudo de caso de 20 bugs, com o julgamento de especialistas para a seleção de bugs com orçamento de 50%, pelos analistas do STI, esperava-se um tempo maior para as escolhas dos bugs, pois 4 analistas terminaram a seleção em menos de 30 minutos e os demais entre 30 a 60 minutos, sendo o mais rápido em apenas 11 minutos. É provável que, por isso, os analistas não tenham conseguido encontrar soluções que considerassem as dependências. Para efeito de comparação, a busca genética conseguiu fazer a mesma busca em 0,56 segundos, o que já seria um ganho de tempo em um pequeno problema de busca, além de ter encontrado as dependências que não foram descobertas pelos analistas.

Seguindo a forma de avaliação das soluções apresentadas na abordagem proposta neste trabalho, todas as seleções dos bugs para correção realizadas pelos analistas, no orçamento de 50%, tiveram a função objetivo fixada com valor zero pois estavam acima do orçamento ou não atendiam as dependências de bugs, que é um dos fatores necessários para uma solução ser válida. Porém, mesmo que o requisito de dependência seja desconsiderado, o maior e o menor valor da função objetivo encontrado nas seleções seria 0,28547 e 0,08628, respectivamente. Por

outro lado, na busca exaustiva encontrou-se a solução ótima com valor de 0,36646 (esse também encontrado pela busca genética em todas as execuções). Já na busca aleatória, obteve-se o valor médio de 0,26873, porém respeitando as restrições de orçamento e dependências de bugs. Logo, mesmo flexibilizando e não considerando as dependências de bugs, ainda assim, os resultados apontados pelos analistas são numericamente próximos da busca aleatória, mas qualitativamente muito inferiores.

Quanto a cobertura de bugs, a busca genética conseguiu selecionar 15 bugs com um custo de R\$7.640,00, contra o valor de R\$7.700,00 para uma cobertura de 13 bugs da melhor seleção por especialistas. Diante destas informações conclui-se que os analistas, em todos os casos, alcançaram uma seleção com a função objetivo sempre abaixo do valor encontrado pela busca genética, isso sem considerar as dependências de bugs, ou seja, mesmo trabalhando de forma livre, sem considerar as dependências de bugs, não conseguiram encontrar boas soluções em comparação com a busca genética.

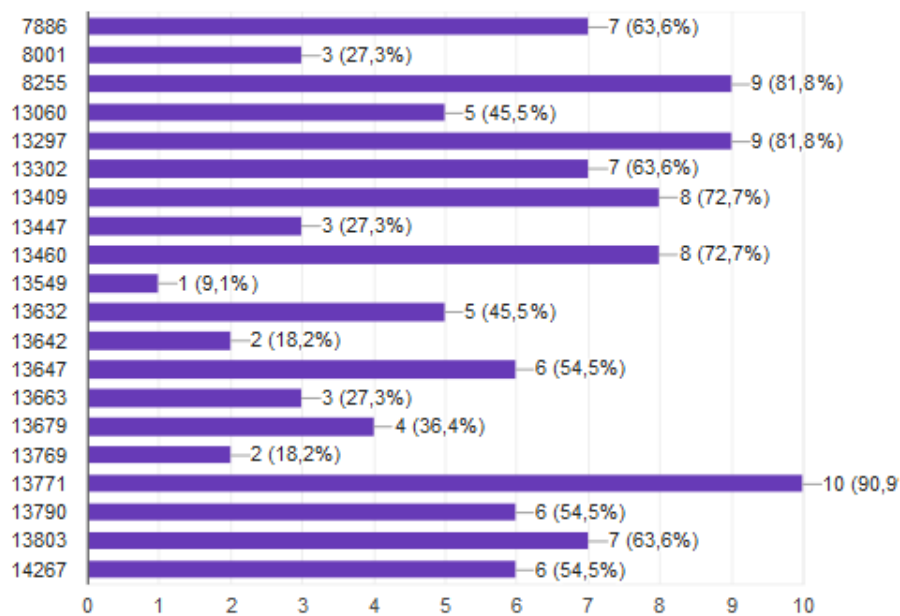
Realizando uma análise dos bugs escolhidos, como mostrado na Figura 32, percebe-se que 10 entre 11 analistas escolheram o bug 13771. Curiosamente, este bug não foi incluído na solução ótima da busca exaustiva e da busca genética. Uma investigação posterior sobre este bug verificou que o mesmo possui alto custo e máxima severidade, que são critérios não indicados pelos analistas como de maior importância na avaliação. No entanto, o bug pode ter sido escolhido pelo fato de possuir uma alta prioridade, critério este indicado como sendo de maior importância. Por outro lado, a busca genética não selecionou este bug uma vez que o mesmo possui uma alta dependência com outros bugs, consumindo praticamente metade do orçamento e prejudicando no grau de cobertura de bugs (maior quantidade de bugs escolhidos).

Na outra extremidade, ou seja, dentre os bugs menos indicados pelos analistas, o bug 13549 foi escolhido apenas uma única vez. Ao analisar este bug na perspectiva das dependências, foi observado que outros bugs escolhidos dependiam do bug 13549 para a solução ser válida, mas, por darem pouca importância às dependências, os analistas acabaram não incluindo este bug que tem impacto direto em diversos outros.

Figura 32 - Quantidade de votos para cada bug

Selecione os bugs para o orçamento de R\$7.770,00, lembre-se de marcar o tempo gasto para escolha. Pode ser necessário abrir o redmine para visualizar e analisar a descrição completa de cada bug, bem como o código do sistema ou até mesmo o sistema/sigaa.

11 respostas



Em relação a forma como foi conduzido o experimento, a grande maioria dos analistas (90,9%) não verificaram o código fonte do sistema em questão, mas apenas verificaram o conteúdo dos relatórios de bugs (81,8%). Vale ressaltar que nenhum chegou a verificar o sistema com testes para melhor caracterizar os bugs. Nesse sentido, percebe-se quão importante são os relatórios de bugs para os analistas de sistemas tomarem suas decisões. Outro ponto que merece destaque é o fato que nenhum analista delineou qualquer arquitetura de dependência de componentes ou de bugs, fato que gerou enormes dificuldades em perceber que as soluções eram inválidas por não atenderem as dependências de bugs.

5.3 CONSIDERAÇÕES FINAIS

Nos experimentos do cenário com 20 bugs, a busca genética se mostrou tão eficiente quanto a busca exaustiva, sendo melhor que as escolhas dos analisas em todos casos e também superando a busca aleatória. Para os experimentos do cenário com 50 bugs, a busca aleatória, nos 14 experimentos, nem mesmo encontrou alguma solução válida. Vale ressaltar que não foi possível realizar a busca exaustiva neste cenário com 50 bugs, pois a quantidade de soluções

candidatas é gigantesca, levando a uma estimativa de tempo de execução em torno de 3.834 mil anos. Conclui-se, então, que à medida que a quantidade de bugs aumenta, a diferença qualitativa da abordagem genética proposta em relação a busca aleatória aumenta, até chegar ao ponto no qual a busca aleatória não consegue nenhuma solução válida, como foi no cenário com 50 bugs.

Para proporcionar maior confiabilidade nos resultados, poderiam ser calculadas outras medidas estatísticas, como por exemplo o desvio padrão. No entanto, considerando a pouca variabilidade dos resultados obtidos, com valores máximos e mínimos quase sempre idênticos, não foram apresentadas tais medidas pois são sempre zero ou muito próximo de zero. Isto posto, verificou-se nas buscas que a estabilidade e qualidade nos resultados alcançados elimina a necessidade de calcular tais medidas estatísticas.

6 CONCLUSÃO

O presente trabalho está contextualizado nas fases de manutenção e testes dos processos de software, empregando conceitos e técnicas da Engenharia de Software Baseada em Buscas, para conceber a estratégia de otimização fundamentada em algoritmos genéticos, e da Engenharia de Software Baseada em Componentes, para identificar as dependências de componentes e, por conseguinte as dependências de bugs. Como principal objetivo, esta dissertação propõe uma abordagem automática para seleção de bugs para correção, inserida no ciclo de vida do *software*. Para avaliação da qualidade das soluções recomendadas pela abordagem proposta, um estudo de caso foi realizado em dois cenários com 20 e 50 bugs, chegando inclusive a avaliar a competitividade humana da proposta a partir da avaliação de um experimento baseado no julgamento de especialista.

Embora a abordagem proposta não tenha focado na seleção de desenvolvedores qualificados para corrigir os bugs selecionados, ela tem como propósito minimizar o esforço do gerente de projetos na definição de quais bugs devem ser selecionados, tendo algumas restrições que se baseiam no impacto para o negócio, e recomenda soluções mais adequada ao produtor de software para equilibrar as diferentes escolhas dos clientes. Porém, com futuros ajustes simples, a proposta pode ser enriquecida com a adoção da priorização dos bugs selecionados, como complemento ao problema de seleção de bugs. Além disso, a abordagem proposta pode ser adotada como etapa inicial para o problema de agendamento, fornecendo como entrada o conjunto de bugs a serem corrigidos, que, combinado com informações relativas a disponibilidade e experiências da equipe de desenvolvimento, pode derivar a recomendação do cronograma de correção.

Em resumo, a abordagem proposta considera: (i) a prioridade e severidade dos bugs, informadas tanto pelo produtor como pelos seus clientes; (ii) as dependências de bugs, inferidas das interfaces providas e requeridas em projetos de software baseados em componentes; (iii) a importância dos múltiplos clientes do ponto de vista do produtor; (iv) o grau de cobertura e o grau de interdependência dos bugs, deduzidos do fechamento transitivo do grafo direcionado correspondente definido pelos componentes de software; e (v) a restrição orçamentária do projeto de software. Como contribuição principal, a abordagem proposta introduz o conceito de dependências de bugs baseadas em componentes, inclusive propondo uma forma automática de rastreamento destas dependências. Além disso, também é uma importante contribuição a

proposição dos conceitos de grau de cobertura e grau de interdependência dos bugs, que complementam o conceito da prioridade e severidade, respectivamente.

Ao explorar um conjunto de dados do mundo real, foram realizados 14 experimentos relacionados a diferentes restrições orçamentárias em cada um dos dois cenários avaliados (com 20 e 50 bugs), com avaliação da busca genética, busca aleatória e busca exaustiva, sendo esta última apenas no cenário com 20 bugs. Além disso, ainda no cenário com 20 bugs e restrição orçamentária de 50%, também foi realizado um único experimento para avaliar a competitividade humana da abordagem proposta, comparando-a com a busca baseada no julgamento de especialistas.

Os resultados mostraram que: (i) a busca genética supera a busca aleatória em relação à qualidade das seleções recomendadas de bugs em ambos os cenários com 20 e 50 bugs, sendo que no cenário de 50 bugs a busca aleatória não obteve nenhuma solução válida; (ii) a busca genética é mais eficiente que a busca exaustiva quanto ao tempo e custo de processamento; (iii) a abordagem proposta pode ser considerada competitiva para o ser humano, em termos de qualidade e tempo, uma vez que foca nas atividades de correção de bugs sobre o que é essencial, diminuindo o esforço da equipe de desenvolvimento ao selecionar, entre vários bugs reportados, um conjunto específico de bugs altamente importantes e impactantes, que são adequados para o orçamento do projeto de software e que atende às restrições das dependência de bugs; e (iv) as recomendações são sensíveis a diferentes calibrações de pesos, permitindo definir as opções que melhor se adaptem aos cenários enfrentados.

Quanto às ameaças à validade do estudo de caso apresentado neste trabalho, pode haver quatro tipos de ameaças (WOHLIN et al. 2000): *ameaças internas*, *ameaças externas*, *ameaças de construção* e *ameaças de conclusão*. No que diz respeito às ameaças internas, a importância dos clientes foi baseada na quantidade de estudantes matriculados nas coordenações de cursos, uma vez que este tipo de informação não se encontra registrada no BTS institucional. Consequentemente, essa decisão pode afetar os bugs recomendados nos experimentos realizados. Apesar da calibração utilizada no estudo de caso ser empiricamente obtida, uma parametrização de afinação pode levar a melhores recomendações em cenários mais complexos (ARCURI et al. 2011). A aplicação de outras metaheurísticas é atraente para outras validações no que se refere ao desempenho em termos de adequação da formulação, qualidade das recomendações, custo e tempo de processamento, e facilidade de adequação da proposta.

Em relação às ameaças externas, o estudo de caso considerou 3 clientes (representados por analistas do STI), 1 produtor e dois conjuntos de dados do mundo real com 20 e 50 bugs.

Réplicas e adaptações em uma gama mais ampla de conjuntos de dados com um grande número de clientes e relatórios de bugs são desejáveis para obter resultados mais generalizados, possibilitando identificar fontes de viés que possam afetar os estudos de caso. Na verdade, os resultados obtidos devem ser aplicáveis às situações em que pressupostos similares sejam mantidos. Caso contrário, a formulação e a representação do problema PSB precisam ser adaptadas ao novo contexto em questão.

Em relação às ameaças de construção, a abordagem proposta adota conceitos e modelos de informação que foram aplicados com sucesso em trabalhos relacionados e nos *Sistemas de Rastreamento de Bugs*. Os pressupostos e os fundamentos desta proposta fazem sentido para o tipo de estudo de caso discutido, no entanto, como já foi mencionado, a formulação e a representação podem mudar caso diferentes abstrações de tarefas de correção de bugs sejam percebidas e adotadas.

Referindo-se a ameaças de conclusão, como um meio para lidar com a natureza estocástica das técnicas de busca e garantir uma comparação justa, as estratégias de busca genética e aleatória foram realizadas várias vezes (1000 execuções) durante cada experimento, superando a aleatoriedade inerente a tais estratégias. Em complemento, é necessário realizar uma análise analítica mais criteriosa para medir semelhanças estatísticas e diferenças entre as estratégias de busca.

É bem verdade que, segundo Runeson et al. (2009), um estudo de caso nunca dará conclusões com significância estatística, mas diferentes evidências, validações juntas e vinculadas passam a ter mais relevância para uma forte conclusão diante de um experimento realizado, como é o caso que se propõe no presente estudo, ao definir múltiplas fontes de forma planejada e consistente, como: (i) os critérios para seleção de bugs com base na literatura (critérios utilizados em outros trabalhos); (ii) utilizou uma metaheurística bastante aplicada em outros trabalhos similares; e (iii) compararam-se as soluções definidas das escolhas manuais dos analistas do STI e das soluções alcançadas pelo algoritmo aleatório e exaustivo com o algoritmo proposto.

Agora, em relação aos trabalhos futuros, como um avanço chave, parece interessante a formulação do problema PSB como um problema multi-objetivo, no qual as soluções podem ser escolhidas a partir de uma Frente de Pareto. Além disso, como outra melhoria, a intenção é incluir a adoção da priorização dos bugs selecionados, os conceitos de agendamento de bugs e alocação de desenvolvedores para automatizar o planejamento e gerenciamento de projetos.

REFERÊNCIAS

- ANDRADE, T. C.; FREITAS, F. G.; BRASIL, M. M. A.; SILVA, T. G. N.; COUTINHO, D. P.. SOUZA, J. T. **XLIII Simpósio Brasileiro de Pesquisa Operacional**. Ubatuba/SP, ago 2011, pp. 15-18.
- ANVIK, J.; HIEW, L.; MURPHY, G. C. **Who should fix this bug**. In: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 361-370.
- ARAKI, L. Y.; VERGILIO, S. R.. **Um framework de geração de dados de teste para critérios estruturais baseados em código objeto java**. XI Workshop de Testes de Tolerância e Falhas, SBRC, 2010, pp. 91-104.
- ARCURI, A.; FRASER, G. **On parameter tuning in search based software engineering**. In: COHEN, M. B., Ó CINNÉIDE, M. (eds.) SSBSE 2011, Heidelberg: LNCS, vol. 6956, Springer, 2011, pp. 33-47.
- Associação Brasileira de Normas Técnicas (ABNT)**. Disponível em: <<http://www.abnt.org.br/>>. Acesso em 01 jun 2017, 7 p.
- BÄCK, T.; FOGEL, D. B.; MICHALEWICZ, T. **Evolutionary computation 1: Basic Algorithms and operators**. New York: Taylor & Francis Group, 2000, p. 237.
- BANOV, J. T. M. **Abordagem Evolutiva para o Planejamento Multi-Período da Expansão da Rede de Acesso aos Serviços de Telecomunicações**. Departamento de Engenharia de Sistemas, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, 2005.
- BAGNALL, A. J.; RAYWARD-SMITH, V. J.; WHITTLEY, I. M. **The next release problem**. Information and Software Technology, 43 (14), 2001, pp. 883-890.
- BARROS, R. S; CORTES, O. A. C.; FILHO, J. D. P.; LOPES, R. F. **Um Algoritmo Híbrido Baseado na Mutação Creep para Resolução de Problemas de Otimização com Restrições**. X CONGRESSO BRASILEIRO DE INTELIGÊNCIA COMPUTACIONAL (CBIC 2011). Fortaleza, Ceará: 08 a 11 nov 2011.
- BEAN, C. J. **Genetic algorithms and random keys for sequencing and optimization**. ORSA Journal of Computing, vol. 6, 1994, pp. 154-160.
- BOEHM, B.; BASILI, V. **Software Defect Reduction Top 10 List**, IEEE Computer Society, vol. 34(1) , jan 2001, pp. 135-137.
- BUGZILLA. Disponível em: < <https://www.bugzilla.org/docs/>>. Acesso em 09 fev 2017.
- BURITY, T.; ELIAS, G. **A Quantitative, Evidence-based Approach for Recommending Software Modules**. Federal University of Paraíba. Informatics Department. João Pessoa – Paraíba – Brazil, 2011.
- CAETANO, C. **Gestão de Defeitos**. Engenharia de Software, Magazine, v. 1, n. 1, 2007, pp. 60-67.

- COHN, Mike. **Agile Estimating and Planning**. Robert C. Martin Series. Prentice Hall. 2006.
- CHRISTEL, M.; KANG, K. **Issues in Requirements Elicitation**. Carnegie Mellon University, Pittsburgh TR.CMU/SEI-92-TR-12. sep 1992.
- DEB, K. et al. **A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II**. IEEE Transactions on Evolutionary Computation, V. 6, 2000, pp. 182-197.
- DEV, H.; AWASTHI, R. **A Systematic Study of Requirement Volatility during Software Development Process**. IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 1, mar 2012.
- D'AMBROS, M.; LANZA, M.; PINZGER, M. **A bug's life: visualizing a bug database**. 4th Int. Work. on Visualizing Software for Understanding and Analysis, 2007, pp. 114-120.
- DE JONG, K. A. **An analysis of the behavior of a class of genetic adaptive systems**. Tese (Ph. D. in Computer and Communication Sciences) - University of Michigan, Ann Arbor, 1975, p. 256.
- DREO, J.; PETROWSKI, A.; SIARRY, P.; TAILLARD, E. **Metaheuristics for Hard Optimization: Methods and Case Studies**. 1 ed., Springer, 2005.
- DREYTON, D.; ARAÚJO, A.A.; DANTAS, A.; FREITAS, Á.; SOUZA, J. **Search-based bug report prioritization for kate editor bugs repository**. In: Barros, M., Labiche, Y. (eds.) SSBSE 2015, Heidelberg: LNCS, vol. 9275, Springer, 2015, pp. 295-300.
- DREYTON, D.; ARAÚJO, A.A.; DANTAS, A.; SARAIVA, R.; SOUZA, J. **A multi-objective approach to prioritize and recommend bugs in open source repositories**. In: Sarro, F., Deb, K. (eds) SSBSE 2016, Heidelberg: LNCS, vol 9962, Springer, 2016, pp. 143-158.
- EIBEN, A. E.; SMITH, J. E. **Introduction to evolutionary computing**. Berlin Heideberg New York: Natural computing Series. Springer-Verlag, 2003, 299 p.
- FERREIRA, T. N.; VERGILIO, S. R.; SOUZA, J. T.. **Engenharia de Software Baseada em Busca e em Preferência: Uma Visão Geral**. CBSOFT 2016, VII Congresso Brasileiro de Software – Teoria e Prática. Maringá: 2016, pp. 1-11.
- FILITTO, D. **Algoritmos Genéticos: Uma Visão Explanatória**. Revista Multidisciplinar da UNIESP. Saber Acadêmico, nº 6, dez 2008, pp. 136-143.
- GABRIEL, P. H. R. **Algoritmos evolutivos e modelos simplificados de proteínas para predição de estruturas terciárias**. Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo: 2010, 68 p.
- GABRIEL, P. H. R.; DELBEM, A. C. B. **Fundamentos de algoritmos evolutivos**. Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo: 2008, p. 19.
- GAREY, M. R.; JOHNSON, D. S. **Computers and intractability: A guide to the theory of NP-completeness**. W. H. Freeman, 1979.

GONÇALVES, V. P.; BARROS, M. O. **Modelando o Problema da Próxima Release sob a Perspectiva da Análise de Pontos de Função**. Programa de Pós-Graduação em Informática – Universidade Federal do Estado do Rio de Janeiro (UNIRIO) – Rio de Janeiro: 2013.

HARMAN, M.; JONES, F. B. **The current state and future of search based software engineering**”. *Future of Software Engineering*, 2007, pp. 342–357.

HARMAN, M.; JONES, F. B. **Software engineering: an ideal set of challenges for evolutionary computation**”. 15th Conf. on Genetic and Evolutionary Computation. 2016, pp. 1759-1760.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. **Search based software engineering: a comprehensive analysis and review of trends techniques and applications**. Tech. Rep. TR-09-03, Department of Computer Science, King’s College London: 2009.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. **Search Based Software Engineering: Trends, Techniques and Applications**. *ACM Computy Survey (CSUR)*, vol. 45, n.1, nov 2012, pp. 1-61.

HOANG, D. T. **Metaheuristics for NP-Hard Combinatorial Optimization Problems**. Departament of Eletrical and Computer Engineering, National University of Singapore: 2008, 212 p.

HOLLAND, J. H. **Adaptation in natural and artificial systems: an introducy analysis with applications to biology, control, and artificial intelligence**. MIT Press, Cambridge, MA, USA, 1992.

JAVED T. **A Study to Investigate the Impact of Requirements Instability on Software Defects**. *ACM Software Engineering Notes*, Volume 29, Number 4, may 2004.

JIRA. Disponível em: <<https://confluence.atlassian.com/jira/jira-documentation-1556.html>>. Acesso em 09 fev 2017.

JONES, C. **Software Quality: Analysis and guidelines for Success**. International Thomson Computer Press, 1997.

KOSCIANSKI, A.; SOARES, M. S. **Qualidade de Software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. São Paulo: editora Novatec, 2 ed., 2007, p. 21.

LANKANFI, A.; DEMEYER, S.; GIGER, E.; GOETHALS, B. **Predicting the severity of a reported bug**. In: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010, pp. 1-10.

LEAN TESTING. Disponível em: < <https://leantesting.com/faq/>>. Acesso em 22 mar 2018.

LEDERER, A. L.; PRASAD, J. **Causes of inaccurate software development cost estimates**. *Journal of Systems and Software*, v. 31 n. 2, nov 1995, pp. 125-134.

LEE, A. et al. **Software development cost estimation integrating neural network with cluster analysis**. *Jounal Information and Management*. v. 34. n. 1, aug 1998, pp. 1-9.

LEHMAN M. M.; RAMIL, J. F.; WERNICK, P. D.; PERRY D. E.; TURSKI W. M. **Metrics and Laws of Software Evolution - The Nineties View**. Proc. Fourth Intl. Software Metrics Symp., Metrics'97, 1997, pp. 20-32.

LIGHTHOUSE. Disponível em: < <http://help.lighthouseapp.com/>>. Acesso em 22 mar 2018.

LIMA, G., ELIAS, G. **An Automated Approach for Selecting Bugs in Component-Based Software Projects**. Proceedings of the 2017 International Conference on Software Engineering Research and Practice, Hamid R. Arabnia, Leonidas Deligiannidis, Fernando G. Tinetti, United States of America: ISBN: 1-60132-468-5, CSREA Press, jul 2017, pp. 38-44.

LINDEN, R. **Algoritmos Genéticos: Uma importante ferramenta da Inteligência Computacional**, Rio de Janeiro: Brasport, 2008, 428 p.

LOPES, H. S.; RODRIGUES, L. C. A.; STEINER, M. T. A. **Metaheurísticas em pesquisa operacional**. Curitiba, PR: Omnipax, capítulo 2, 2013, 472 p.

LOUZADA, V. B. F. **Redução e Controle de Erros para as Atividades de Teste de Softwares**. XI SIMPEP – Simpósio de Engenharia de Produção. Baurú – SP: nov 2004.

MAMDOUH ALENEZI; SHADI BANITAAN; **Bug Reports Prioritization: Which Features and Classifier to Use?** 12th International Conference on Machine Learning and Applications. 2013.

MANTIS BUG TRACKER. Disponível em: <<https://http://www.mantisbt.org/documentation.php>>. Acesso em 22 mar 2018.

MELIÁN, B; PÉREZ, J. A. M.; VEGA, J. M. M. **Metaheurísticas: una visión global**. Revista Iberoamericana de Inteligência Artificial. Valencia - España: vol. 7, n. 19, 2003.

MYERS, G. J. **The Art of Software Testing**. United States of America: 2nd ed., 2004, p. 10.

MORALES, J. D. V. **Deteção de dano em estruturas utilizando algoritmos genéticos e parâmetros dinâmicos**. Departamento de Engenharia de Estruturas da EESC-USP, São Carlos: 2009, 191 p.

NEBRO, A. J.; DURILLO, J. J.; LUNA, F.; DORRONSORO, B.; ALBA, E. **MOCeII: A Cellular Genetic Algorithm for Multiobjective Optimization**. International Journal of Intelligent Systems, vol 24, jul 2009, pp. 726 -746.

NETTO, F. C.; BARROS, M. O.; ALVIM, A. C. F. **Uma Abordagem Automatizada para Geração de Cronogramas de Tarefas de Correção de bugs**. Brazilian Symposium on Software Engineering (SBES), Universidade Federal do Estado do Rio de Janeiro (UNIRIO), Salvador: 27 set a 01 out 2010, pp. 80-89.

NETTO, F. C.; BARROS, M. O.; ALVIM, A. C. F. **An automated approach for scheduling bug fix tasks**. International Journal of Software Engineering and Knowledge Engineering, 26(2), pp. 239-271, 2016.

NÓBREGA, S.; SOUZA, S. R.; SOUZA, M. J. F. **Uma Abordagem MultiObjetivo para o Problema de Desenvolvimento de Cronogramas de Projetos de Software**. WESB 2013. Centro Federal de Educação Tecnológica de Minas Gerais (CEFET/MG). Brasil: 2013.

PAGNO, R. T. **Uma ferramenta de estimativa de custos para o desenvolvimento distribuído de software**. Universidade Estadual de Maringá, Maringá: 2010, 172 p..

PENG HE; BING LI; YUTAO MA; LULU HE. **Using Software Dependency to Bug Prediction**. Mathematical Problems in Engineering. Hindawi Publishing Corporation, vol. 2013, 12 p.

PIMMLER, T. U.; EPPINGER, S. D. **Integration analysis of product decompositions**. ASME Design Theory and Methodology Conference, Minneapolis, MN: set 1994, 10 p.

PRESSMAN, R. S. Engenharia de Software, McGrawHill, 6ª edição, 2006.

REDMINE. Disponível em: <<https://www.redmine.org/guide/>>. Acesso em 09 fev 2017.

RIBEIRO, B. L. **Uma Abordagem para Alocação de Equipes em Projetos de Desenvolvimento Distribuído de Software com base em Atributos Não-Técnicos**, João Pessoa/PB: abr 2013.

RUNESON, P.; HÖST, M. **Guidelines for conducting and reporting case study research in software engineering**. Empirical Software Engineering, volume 14 issue 2, apr 2009, pp. 131-164.

SERRANO, N.; CIORDIA, I. **Bugzilla, ITracker, and Other Bug Trackers**. IEEE Software Society, vol 22(2), mar-abr 2005, pp. 11-13.

SHARMAN, D. M.; YASSINE, A. A. **Characterizing complex product architectures**. *Systems Engineering Journal*. 7(1), 2004, pp. 35-60.

SOMMERVILLE, I. **Software Engineering**. Pearson, 9 th ed., 2011.

SOUZA, D. J. **Uma abordagem sobre custo de software**. WebArtigos Link: <http://www.webartigos.com>. Acessado em 15 dez 2014, em 12 mai 2011.

SOUZA, R. G. D. **Introdução ao Bugzilla – Guia Prático de Instalação**, Developer Works - IBM. dez 2012.

SPAGNOLI, L. A.; BECKER, K. **Um estudo sobre o Desenvolvimento Baseado em Componente**. PPGCC/PUCRS, Porto Alegre/RS: mai 2003, 40 p.

THUNG, F. **Automatic prediction of bug fixing effort measured by code churn size**. 5th Int. Work. On Software Mining, 2016, pp. 18-23.

XIAO, J.; AFZAL, W. **Search-based resource scheduling for bug fixing tasks**. In: Proceedings of the 2nd International Symposium on Search Based Software Engineering, Benevento: 07-09 set 2010, pp. 133–142.

YELTSIN, I.; PAIXÃO, M.; SOUZA, J. **Uma Adaptação de Algoritmo Genético para o Problema do Próximo Release com Interdependências entre Requisitos**. WESB-2013. Grupo de Otimização em Engenharia de Software da UECE. Universidade Estadual do Ceará (UECE). Fortaleza/CE: 2013.

YUAN TIAN; Lo, D; CHENGNIAN SUAN. **DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis**. 29th IEEE International Conference on Software Maintenance (ICMS), Eindhoven: 22-28 sep 2013, pp. 200-209.

WARFIELD, J. N. **Binary matrices in system modeling**. IEEE Transactions on Systems, Management, and Cybernetics 3(5), 1973, pp. 441-449.

WEISS, C.; PREMRAJ, R.; ZIMMERMANN, T.; ZELLER, A. **Predicting effort to fix software bugs**. In Proceedings of the 9th Workshop Software Reengineering, may 2007.

WOHLIN, C.; RUNESON, P.; HÖST, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. Kluwer Academic Publishers. Norwell, 2000.

ZHANG, Yang; YIN, Gang; WANG, Tao; YU, Yue; WANG, H. **Evaluating Bug Severity Using Crowd-based Knowledge: An Exploratory Study**. Proceedings of the 7th Asia-Pacific Symposium on Internetware, Wuhan, China: 06 nov 2015, pp. 70-73.