CBSOFT

VIII CONGRESSO BRASILEIRO DE SOFTWARE

FORTALEZA 2017

ANAIS

VIII WORKSHOP DE ENGENHARIA DE SOFTWARE BASEADA EM BUSCA

REALIZADO POR

ORGANIZAÇÃO

APOIO

PATROCÍNIO

VIII WORKSHOP DE ENGENHARIA DE SOFTWARE BASEADA EM BUSCA

18 a 22 de setembro de 2017 | *September 18 - 22, 2017*
Fortaleza, CE, *Brazil*

ANAIS | *PROCEEDINGS*

Sociedade Brasileira de Computação – SBC

COORDENADORES DO COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE CHAIRS*
Márcio Barros (UNIRIO)
Sílvia Regina Vergilio (UFPR)

EDITORES | *PROCEEDINGS CHAIRS*
Fernando Antonio Mota Trinta (UFC)

COORDENADORES GERAIS | *GENERAL CHAIRS*
Fernando Antonio Mota Trinta (UFC)
Rossana Andrade (UFC)
Marum Simão Filho (UNI7)

REALIZAÇÃO | *REALIZATION*
Sociedade Brasileira de Computação (SBC)

EXECUÇÃO | *EXECUTION*
Universidade Federal do Ceará (UFC) – Departamento de Computação (DC)
Universidade Estadual do Ceará (UECE)
Centro Universitário Sete de Setembro (UNI7)
Universidade de Fortaleza (UNIFOR)

APOIO | *SUPPORT*
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)
Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq)

PATROCÍNIO | *SPONSORS*
Google
Mob Telecom

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Central do Campus do Pici

8º Workshop de Engenharia de Software Baseada em Busca (8. : 2017 : Fortaleza).

Anais do 8º Workshop de Engenharia de Software Baseada em Busca [recurso eletrônico] / 8º Workshop de Engenharia de Software Baseada em Busca, 18-22 setembro 2017; editado por Fernando Antonio Mota Trinta. – Fortaleza: Universidade Federal do Ceará, 2017.

Tema central: Software: Teoria e Prática.
Disponível em: http://www.sbc.org.br/cbsoft2017/anais.
ISBN: 978-85-7669-394-9
1. Software. 2. Engenharia de software. 3. Software - Desenvolvimento. I. 8º Congresso Brasileiro de Sofware. II. Trinta, Fernando Antonio Mota. III. Barros, Márcio. IV. Vergilio, Sílvia. V. Título.

Apresentação

Sejam bem-vindos ao Workshop de Engenharia de Software Baseada em Busca – WESB. Nesta oitava edição, o *workshop* se consagra como um importante fórum de discussão sobre a aplicação de técnicas de busca para solucionar problemas da Engenharia de Software no cenário nacional. No contexto do WESB, técnicas de busca englobam tanto técnicas tradicionais, como força bruta ou *branch-and-bound*, quanto meta-heurísticas, como algoritmos genéticos e bio-inspirados. O WESB é um *workshop* sobre fundamentos teóricos, de experiências práticas e de automação da Engenharia de Software Baseada em Busca (*SBSE – Search Based Software Engineering*) em projetos acadêmicos e industriais.

O objetivo é a troca de experiências, opiniões e debates entre os participantes, visando a discussão do cenário atual e perspectivas de colaborações e pesquisas futuras na área. Os trabalhos submetidos para esta edição foram cuidadosamente revisados por pelo menos três avaliadores do comitê de programa, que contou com pesquisadores de diferentes regiões do país. Estes anais contêm os trabalhos selecionados dentre as submissões recebidas. Ao todo, seis trabalhos completos foram selecionados e serão apresentados nas duas seções técnicas que integram o evento. Os principais temas abordados incluem: teste de software, requisitos de software, reparo de software e métricas. Além das seções técnicas, a programação também inclui uma palestra convidada e discussões sobre SBSE.

Gostaríamos de parabenizar a todos os autores dos trabalhos selecionados e agradecer aos autores de todas as submissões realizadas. Agradecemos especialmente aos membros do Comitê Técnico de Programa e aos revisores por eles elencados que cumpriram com presteza os prazos estabelecidos. Sabemos que sem esta valiosa colaboração não teríamos conseguido concluir o processo de revisão no tempo previsto. Agradecemos também aos organizadores do CBSoft 2017 pela infraestrutura disponibilizada e pela oportunidade oferecida.

Esperamos que o WESB 2017 contribua para ampliar e consolidar a área de Engenharia de Software Baseada em Busca no Brasil.

Desejamos a todos um excelente evento!

Fortaleza, setembro de 2017.

Márcio de Oliveira Barros (UNIRIO)
Silvia Regina Vergilio (UFPR)
Coordenadores do WESB 2017

Foreword

Welcome to the Search-Based Software Engineering Brazilian Workshop – WESB. In its eighth edition, the workshop is consolidated as an important national forum for the discussion on the application of search-based techniques to address Software Engineering problems. In this sense, search techniques encompass both traditional techniques, such as brute force and branch-and-bound, as well as metaheuristics, such as genetic and bio-inspired algorithms. The areas of interest for WESB include theoretical foundations, practical experiences, and automation of Search-based Software Engineering, both in academic and industrial projects.

We look forward for the exchange of experiences, opinions and debates among the participants, aiming at the discussion of the current scenario and perspectives for collaboration and future researches in the area. The papers submitted for this edition were carefully reviewed by at least three peer reviewers from the Program Committee, which brings forth researchers from different regions of the country. The following pages contain six selected papers from the submissions received. These papers will be presented in the two technical sessions that will be part of the event. Key topics covered by these papers include: software testing, software requirements, software repair, and metrics. In addition to the technical sessions, the schedule also includes an invited talk and discussions on SBSE.

We would like to congratulate the authors of the selected papers and thank the authors for all submissions. We are particularly grateful to the members of the Program Technical Committee and to the reviewers whom they have appointed. We know that without this valuable collaboration we would not have been able to complete the review process in due time. Thanks also to the organizers of CBSoft 2017 for the infrastructure provided and the opportunities offered.

We hope that WESB'2017 will contribute for the expansion and consolidation of the Search-based Software Engineering research area in Brazil.

We wish you all an excellent event!

Fortaleza, September 2017.

Márcio de Oliveira Barros (UNIRIO)
Silvia Regina Vergilio (UFPR)
WESB 2017 PC Chairs

Comitê técnico | *Technical committee*

Coordenadores de comitê de programa | *PC chair*

Márcio de Oliveira Barros (UNIRIO)
Silvia Regina Vergilio (UFPR)

Comitê de programa | *Program committee*

Adriana Cesário de Faria Alvim (UNIRIO)
André Britto (UFS)
Arilo Claudio Dias Neto (UFAM)
Auri Marcelo Rizzo Vincenzi (UFSC)
Celso G. Camilo-Junior (UFG)
Geraldo R. Mateus (UFMG)
Gledson Elias (UFPB)
Gustavo A. L. de Campos (UEC)
Jerffeson T. de Souza (UECE)
Leila M. A. Silva (UFS)
Maria Cláudia F. P. Emer (UFTPR)
Thelma E. Colanzi (UEM)
Wesley K. G. Assunção (UFTPR)

Revisores externos | *External reviewers*

Eduardo N. A. Farias

Programação | *Workshop program*

Dia 1 | *Day 1* **1**

Sessão Técnica #1 | *Technical Session #1*

Métodos Evolucionários e o Next Release Problem
Rodrigo Basniak (UFTPR), Adolfo Gustavo Serra Seca Neto (UFTPR), Laudelino C. Bastos (UFTPR), Maria Claudia F. P. Emer (UFTPR) 1

Software Requirements Prioritization using Fuzzy Logic
Dayvison Lima (UECE), Raphael Saraiva (UECE), Thayse Alencar (UECE), Gustavo Campos (UECE), Jerffeson Souza (UECE) 11

Utilizando Função de Escalarização para Controle da Relevância de Métricas de Software
Italo Yeltsin (UECE), Allysson Allex Araújo (UFC), Raphael Saraiva (UECE), Jerffeson Souza (UECE) 21

Sessão Técnica #2 | *Technical Session #2*

Genetic Programming-based Composition of Fault Localization Heuristics
Diogo M. de Freitas (UFG), Plínio S. Leitão-Júnior (UFG), Celso G. Camilo-Junior (UFG), Altino Dantas (UFG), Rachel Harrison (Oxford Brookes University) 31

Uma Abordagem para a Priorização de Casos de Teste Baseada no Histórico de Detecção de Falhas
Dennis Silva (UFPI), Ricardo Rabelo (UFPI), Pedro Santos Neto (UFPI), Guilherme Lima (UFPI), Ricardo Britto (Blekinge Institute of Technology), Pedro Almir Oliveira (IFMA) 41

Ternarius: um operador de mutação para o reparo de software baseado em busca com representação subpatch
Vinícius P. L. De Oliveira (UFG), Eduardo F.D. Souza (UFG), Altino Dantas (UFG), Lucas Roque (UFG), Celso G. Camilo-Junior (UFG), Jerffeson T. Souza (UECE) 51

Lista de autores | *Authors* **61**

Métodos Evolucionários e o *Next Release Problem*

Rodrigo Basniak [1], Adolfo Gustavo Serra Seca Neto [1], Laudelino C. Bastos [1], Maria Claudia F. P. Emer[1]

[1]Departamento Acadêmico de Informática (DAINF)
Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba – PR – Brazil

rbasniak@gmail.com, {adolfo, bastos, mcemer}@utfpr.edu.br

Abstract. *This paper presents a literature review that helped to understand the state of the art of the researches that tried to solve this problem. It has been discovered that this area still has much to be developed and that it has been given a constant attention from researches all over the world. This review mapped the source of the datasets used in the literature and discovered that most of them are randomly generated. The most used techniques are genetic algorithms and ant colony optimization and the results' validation are done using the fitness curves or Pareto fronts and, only a small number of works cared to do a more detailed analysis on the quality of the solutions generated by the algorithms.*

Resumo. *Esse artigo apresenta uma revisão de literatura com o intuito de entender o estado da arte das pesquisas que buscam encontrar soluções para esse problema. Com a realização da revisão pode-se dizer que essa área ainda tem muito a ser desenvolvida e que tem recebido uma constante atenção de pesquisadores do mundo todo. A revisão buscou os tipos de datasets nos quais os estudos são aplicados e pôde-se determinar que a maioria dos datasets é gerado aleatoriamente e com poucos requisitos. As técnicas mais utilizadas são algoritmos genéticos e otimização por colônia de formigas e a validação dos resultados é apenas teórica, fazendo uso de gráficos de fitness ou frontes de Pareto, apenas uma pequena porcentagem se preocupa em fazer uma análise detalhada em cima das soluções encontradas pelos algoritmos.*

1. Introdução

A Engenharia de Requisitos é parte fundamental do processo de desenvolvimento de software, pois são os requisitos de software que transmitem as necessidades dos clientes para os desenvolvedores. Portanto, requisitos formam a base de todos os produtos de software e, consequentemente, da Engenharia de Requisitos (ER) [1].

O desafio de definir quais funcionalidades deverão ser implementadas em um software é enorme, pois envolve muitos fatores, como a preferência dos *stakeholders*[1], custos de desenvolvimento, prazos, requisitos mutáveis, entre outros fatores [2].

Existe uma solução exata para esse problema, porém ela não é conhecida e o esforço computacional que seria necessário para encontrá-la é inviável, sendo então necessário o

[1] *Stakeholder: pessoa ou grupo que possui algum tipo de interesse no projeto.*

uso de alguma técnica de busca [3]. Entretanto, para a aplicação dessas técnicas se tornar possível, antes o problema deve ser modelado como um problema de busca.

Quem definiu e modelou o Problema do Próximo *Release* (*Next Release Problem*, ou NRP) como um problema de busca e otimização pela primeira vez foi Bagnal [2]. Este também o classificou como um problema do tipo NP-Difícil, ou seja, é um problema combinatorial e inviável de ser tratado por métodos exaustivos mesmo quando o número de requisitos é relativamente pequeno, o que pode ser confirmado na literatura, pois vários trabalhos se dedicam a aplicar as técnicas propostas em cima do conjunto de requisitos proposto em [4], e que possui apenas 20 requisitos e 5 *stakeholders*.

Dada a natureza do problema, e o fato de não ser possível obter uma solução analítica exata [5], de 2001 até a data desse trabalho, muitos métodos foram propostos para resolver esse problema. A maioria deles utilizando algoritmos de computação evolucionária, que é a área de conhecimento que compreende uma família de algoritmos de otimização inspirados na evolução biológica das espécies.

A contribuição deste trabalho é o mapeamento das aplicações de técnicas meta-heurísticas no campo de priorização e seleção de requisitos de software, que são atividades do planejamento de *release*.

O presente trabalho foi estruturado da seguinte maneira: a Seção 2 apresenta uma breve explicação do problema do Próximo *Release* em sua formulação mono e multi-objetivo; a Seção 3 apresenta um breve histórico dos trabalhos relacionados a este; a Seção 4 apresenta a metodologia escolhida para a realização do mapeamento proposto; a Seção 5 apresenta os resultados e discussão, buscando responder as questões de pesquisa e seguir o sistema de classificação propostos na Seção 4; a Seção 6 acrescenta algumas considerações sobre o presente trabalho.

2. Problema do Próximo *Release*

O problema de definir quais requisitos farão parte do próximo *release* de um software foi proposto como um problema de busca e otimização pela primeira vez em 2001 [2], e passou a ser conhecido como "*Next Release Problem*". Nessa seção o modelo original é apresentado de forma sucinta. O modelo tem um conjunto inicial de requisitos $R = \{r_1, r_2, \dots, r_n\}$ (por exemplo, o *backlog* de um projeto), que representam funcionalidades do sistema, propostas ou desejadas por m *stakeholders*, e todas são candidatas a serem incluídas na próxima versão. Os *stakeholders* são representados em um vetor $S = \{s_1, s_2, \dots, s_m\}$.

Alguns *stakeholders* podem ter uma importância maior, por isso, cada *stakeholder* i possui uma importância w_i associada a ele, resultando em um vetor $W = \{w_1, w_2, \dots, w_m\}$ com a importância de cada um para o projeto.

Para incluir um requisito j em uma versão do software, existe um custo e_j de desenvolvimento associado, que representa o esforço necessário para sua implementação. Assim é definido o vetor $E = \{e_1, e_2, \dots, e_n\}$.

É comum um requisito ser importante para diferentes *stakeholders*, porém, o mesmo requisito raramente tem o mesmo grau de importância entre diferentes *stakeholders*. Com isso, a importância que um requisito r_j tem para um *stakeholder* s_i é dada pelo valor v_{ij}.

Quanto maior a importância do requisito r_j tem para um *stakeholder* s_i, maior o valor de v_{ij}. Todos esses valores de importância podem ser arranjados em uma matriz $m \times n$.

O valor agregado ao software, s_j, gerado pela inclusão de um requisito r_j, é medido pela soma ponderada de seus valores de importância para cada *stakeholder* s_i e pode ser representado por $s_j = \sum_i^m w_i \, v_{ij}$.

Para definir o conteúdo de um *sprint*, é necessário selecionar um subconjunto $\hat{R}$ a partir de R, que maximize o valor agregado ao software e minimize o esforço de desenvolvimento. O valor agregado (satisfação total dos *stakeholders*, ou 'sat') e o esforço de desenvolvimento (ou 'eff') podem ser obtidos, respectivamente, pelas Equações 1 e 2.

$$sat(\hat{R}) = \sum\nolimits_{j \in \hat{R}} s_j \tag{1}$$

$$eff(\hat{R}) = \sum\nolimits_{j \in \hat{R}} e_j \tag{2}$$

Dado um corte de orçamento no projeto, existe um número limitado de recursos B, e o esforço total necessário para desenvolver todas as funcionalidades em $\hat{R}$ não pode exceder esse limite.

Por fim, o problema pode ser formulado como um problema de otimização pelas Equações 3 e 4.

$$maximizar\ sat(\hat{R}) \tag{3}$$

$$sujeito\ a\ eff(\hat{R}) \leq B \tag{4}$$

Esse tipo de problema é conhecido como sendo NP-difícil [5] e pode ser representado como um problema de mochila 0/1.

Em 2007 esse modelo foi estendido para uma abordagem multi-objetivo [6], ou seja, agora mais de um objetivo é considerado. Além de maximizar o valor agregado ao software, é necessário minimizar o esforço de desenvolvimento. O objetivo dessa abordagem é explorar a gama inteira de soluções ótimas variando o peso desses dois objetivos. Com essa abordagem é possível usar os resultados para responder perguntas como: *"O que aconteceria se pudéssemos alocar mais 10% de recursos para esse projeto?"*, ou ainda,*"O que aconteceria se tivéssemos que reduzir o orçamento em 20%?"*

Então para uma formulação multi-objetivo o problema pode ser descrito pelas Equações 5, e 6:

$$maximizar\ sat(\hat{R}) \tag{5}$$

$$minimizar\ eff(\hat{R}) \tag{6}$$

Em um problema multi-objetivo não existe apenas uma solução, mas várias, e, para avaliá-las utiliza-se o gráfico de Eficiência de Pareto [7].

3. Trabalhos Relacionados

Dois trabalhos relativamente recentes e com propostas parecidas com as deste trabalho foram encontrados na literatura. O mais recente [8] foca não só na Engenharia de Software Baseada em Busca, mas também em Preferência. Nesse trabalho, os autores procuram responder em que momento as preferências são fornecidas, quais áreas da Engenharia de Software são investigadas e quais são os algoritmos utilizados. Como resultado do trabalho eles concluíram que: em mais da metade dos artigos as preferências eram fornecidas interativamente; que quatro áreas da Engenharia de Software eram abordadas nos artigos, sendo que a mais abordada foi o planejamento de *releases*; e que mais da metade dos artigos estudados empregaram algoritmos genéticos em sua formulação original ou em alguma variação.

O segundo trabalho [9], também uma revisão de literatura, foca na priorização e seleção de requisitos utilizando métodos de busca. Ao longo do trabalho, os autores não chegaram a uma conclusão quanto às tendências de modelagem, pois existem trabalhos publicados utilizando modelagens mono ou multi-objetivo, e ambos os tipos de modelagem estavam sendo publicadas em trabalhos recentes. Os autores [9] concluíram que a maioria dos trabalhos utilizava algoritmos genéticos ou alguma variação dos mesmos.

Apesar de similar ao trabalho de 2015 [9], o trabalho proposto aborda aspectos diferentes como o tipo de validação utilizado nos trabalhos, os tipos de bases de dados nas quais os métodos foram aplicados e os trabalhos mais recentes na área, posteriores a 2013, visto que apesar de ser de 2015, o trabalho [9] só considerou trabalhos publicados até 2013.

4. Metodologia

O processo de mapeamento escolhido para esse trabalho, foi proposto por Petersen et al [10] e inclui a definição das questões de pesquisa, busca e triagem dos trabalhos, definição do sistema de classificação e extração dos dados.

Inicialmente foram definidas as questões de pesquisa:

- **RQ1:** quais métodos evolucionários já foram aplicados no processo de priorização e seleção de requisitos de software.
- **RQ2:** como os métodos utilizados foram validados?
- **RQ3:** sobre que tipos de bases de dados os métodos foram aplicados?

Na sequência foi definido um conjunto de palavras-chave baseado na proposta do trabalho, que é apresentado na Tabela 2. Os termos do primeiro grupo estão diretamente ligados ao problema principal do trabalho, ou seja, Engenharia de Requisitos. Os termos do segundo grupo, ou seja, Algoritmos Evolucionários, são os diversos nomes e siglas de métodos evolucionários que tem sido aplicado em Engenharia de Software Baseada em Busca, e foram adaptados do trabalho de Nascimento et al [8]. Para as buscas iniciais, os artigos deveriam conter obrigatoriamente uma palavra do primeiro grupo e uma do segundo grupo no título, resumo e/ou palavras-chave.

Área	Palavra-Chave
Engenharia de Requisitos	next release problem
	next release
	release planning
	software release
	requirements
	requirements prioritization
	requirements elicitation

Área	Palavra-Chave
Algoritmos evolucionários	search based
	multi-objective optimization
	genethic algorithm
	genethic programming
	hill climbing
	simulated annealing
	ant colony optimization
	particle swarm optimization
	artificial bee colony
	differential evolution
	evolutionary algorithm

Tabela 1 - Palavras-chave utilizadas na pesquisa

Bases de dados eletrônicas na área de ciência da computação e tecnologia em geral foram consultadas. A quantidade de resultados encontrados em cada base é apresentada na Tabela 2, totalizando 332 resultados que, numa análise inicial, possuíam relação direta com a proposta do trabalho, baseado em uma leitura preliminar do título e do resumo.

Base Acadêmica	Endereço Eletrônico	Resultados
ACM Digital Library	http://dl.acm.org	39
IEEE Xplore	http://ieeexplore.ieee.org	79
SBSE Repository	http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/repository.html	66
Science Direct	http://www.sciencedirect.com	47
Scopus	http://www.scopus.com.br	51
Web of Science	http://www.isiknowledge.com	39

Tabela 2 – Bases acadêmicas utilizadas na pesquisa

Em posse de todos os resultados, o primeiro passo da triagem foi remover os artigos duplicados e que não foi possível obter acesso ao texto completo durante a fase de pesquisa, resultando em 75 itens. Em seguida os títulos e resumos foram analisados individualmente para remoção dos resultados que não estavam de alguma forma ligados à proposta desse trabalho, diminuindo o número para 42. O último passo da triagem correspondeu à leitura completa de todos os artigos, no qual foi detectado que 3 dos artigos eram apenas teóricos (revisão de literatura ou resumos). Com isso o número total de artigos utilizados para gerar os resultados apresentados neste mapeamento foi de 35. A tabela resumida com os artigos utilizados no mapeamento pode ser acessada em *https://goo.gl/mVbmZt*.

O sistema de classificação, apresentado na Tabela 3, foi definido baseado nas questões de pesquisa.

Grupo	Objetivo
Data	mapear os anos em que os artigos foram publicados
Algoritmos	determinar quais tipos de algoritmos tem sido utilizados
Modelagem	determinar como o problema foi abordado, se foi utilizando uma modelagem de objetivo simples ou multi-objetivo.
Fontes	determinar de onde vieram os requisitos utilizados nos trabalhos
Dependências	determinar se os problemas foram modelados considerando as dependências existentes entre os requisitos.
Validação	determinar como os trabalhos tem sido validados e os resultados apresentados

Tabela 3 - Grupos utilizados para classificação dos resultados

5. Resultados e Discussão

Após a formulação inicial do NRP em 2001, houve um período, até 2008, no qual o problema não teve muita atenção, no qual poucos artigos foram publicados. Porém, essa situação se inverteu a partir de 2009, quando passaram a ser publicados de 3 a 5 artigos por ano abordando o NRP. Em 2016 esse número caiu novamente, pois somente um artigo foi publicado. A Figura 1 mostra o número de publicações que abordaram o NRP no período de 2001 a 2017. Como esse trabalho foi elaborado no primeiro semestre de 2017, o número de artigos publicados nesse ano pode não refletir a realidade.

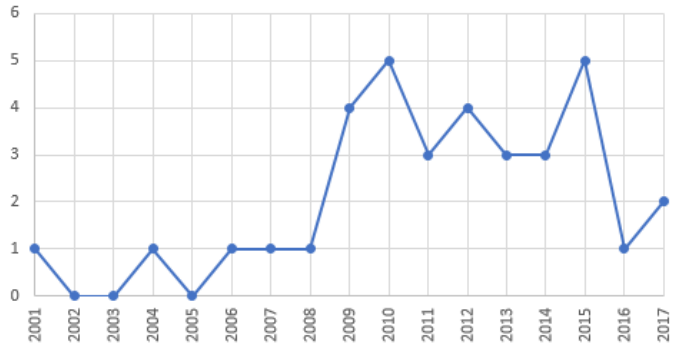


Figura 1 - Publicações por ano

A maioria dos artigos propõe algum tipo de inovação, seja propondo uma metodologia nova ou, adaptando algum algoritmo existente para obter melhores resultados para esse problema em específico. Do total de artigos, 71% se enquadram nessa situação e os 29% restantes, aplicam os métodos em suas definições originais, com o objetivo de comparar os resultados entre eles.

Com relação às famílias de algoritmos utilizados, os algoritmos genéticos dominam essa categoria, sendo utilizado em 36% das aplicações. Em segundo e terceiro lugares estão o ACO (*Ant Colony Optimization*) e SA (*Simulated Annealing*) com 13% e 12% respectivamente. Para esses resultados, os algoritmos foram agrupados com base em sua formulação original. Ou seja, o algoritmo *Ant-Q*, que é uma variação do ACO original, foi contabilizado no grupo ACO, por exemplo. Essa decisão foi tomada de modo a tentar

observar algumas tendências mais gerais sobre as famílias de algoritmos utilizados, pois do contrário, como cada trabalho apresenta um certo grau de adaptação nos algoritmos, quase todos os resultados seriam grupos com apenas uma aplicação nos trabalhos considerados. A Figura 2 mostra o total de utilizações de cada método, lembrando que existem trabalhos que aplicam mais de um método. O grupo "Outros" agrupam algoritmos que foram utilizados um ou duas vezes e não se enquadram em nenhuma das outras famílias.

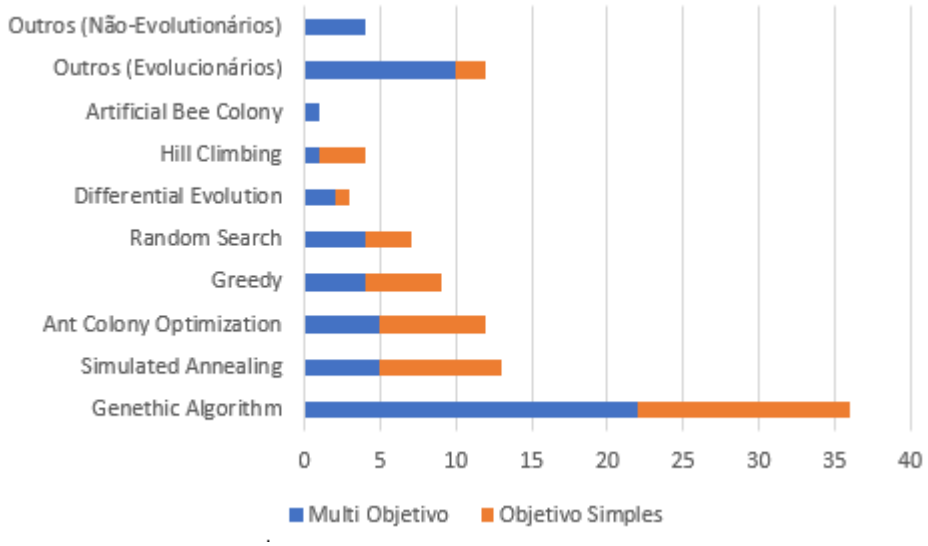


Figura 2 - Número de aplicações de cada família de algoritmos

Quanto ao tipo de modelagem utilizado, o resultado ficou bem distribuído. Uma pequena parcela dos artigos, apenas 11% deles, utilizaram ambas as modelagens, mono e multi-objetivo, no mesmo trabalho; 43% utilizaram uma abordagem mono-objetivo e 46% uma abordagem multi-objetivo.

Os métodos precisam ser aplicados em um dataset de requisitos para produzir as soluções e esse trabalho mapeou três tipos de datasets: 1) gerados aleatoriamente, 2) dados de projetos reais e 3) minerados de repositórios *open source*.

Os requisitos gerados de forma aleatória seguem critérios diferentes, mas de uma forma geral são gerados por meio de dados randomizados e com critérios especificados em seus respectivos artigos. Os dados de projetos reais são provenientes de empresas e projetos que realmente aconteceram. O mais comum é um dataset de um projeto da Motorola, com 35 requisitos e 4 stakeholders [11]. Já os datasets minerados de fontes *online* foram criados a partir de repositórios de bugs de projetos *open source*. Cada *bug* foi considerado como um requisito e cada pessoa que fez algum comentário para esse *bug* foi considerada como um cliente [12]. Os valores de peso de cada *stakeholder* e esforço necessário para implementar o requisito foram gerados aleatoriamente.

A grande maioria, 69% dos projetos, utiliza datasets gerados aleatoriamente, 11% datasets de projetos reais e 14% datasets minerados, os outros 6% não cita a origem dos dados. O restante não informou a origem. Dos 4 projetos que utilizaram datasets reais, 3 utilizaram os dados da Motorola e o outro não especificou a fonte.

O tamanho desses datasets também foi avaliado quanto ao número de requisitos e número de *stakeholders* envolvidos. Dos 68% de trabalhos que citam o tamanho dos datasets utilizados, 73% não excedem a marca de 200 requisitos. Os outros 27% utilizam datasets variando de 412 a 3502 requisitos. O número de *stakeholders* segue uma distribuição parecida, sendo que 58% utilizam datasets com até 20 *stakeholders* e os outros 42% variam de 21 a 1000 *stakeholders*. Os histogramas na Figura 3 mostram a distribuição dos requisitos e *stakeholders* respectivamente.

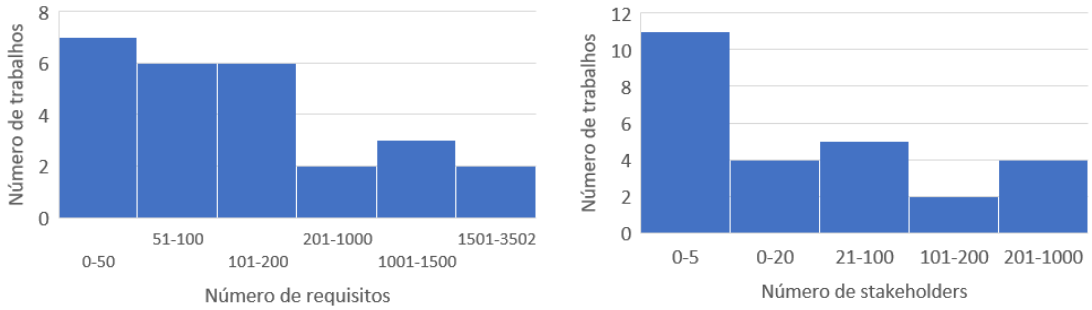


Figura 3 – Histograma de distribuição do número de requisitos (esquerda) e *stakeholders* (direita) nos datasets

Em um projeto real os requisitos possuem interdependências entre si, sendo a mais comum delas a precedência de requisitos, na qual um determinado requisito só pode ser desenvolvido depois que outro já estiver implementado. Esse é um modelo bastante simplista, pois na realidade os requisitos possuem interdependências mais complexas, como mutualidade ou exclusão [13]. Um total de 23% dos artigos não cita como foram tratadas as interdependências entre os requisitos e 20% não as considerou. Dos 57% que restaram, 35% consideraram o conjunto completo de interdependências, e o restante considerou apenas a relação de precedência entre eles.
A última dimensão avaliada foi quanto ao tipo de validação ou apresentação dos resultados utilizada nos trabalhos. De forma geral, foi observado que os artigos que utilizam uma abordagem mono-objetivo apresentam os resultados da função de *fitness* da solução final do algoritmo proposto e de outros que foram escolhidos para comparação. Se a abordagem foi multi-objetivo os resultados são apresentados de duas formas diferentes: 1) comparando média e desvio padrão das soluções encontradas e/ou 2) comparando a forma e completude do fronte de Pareto [7]. De todos os trabalhos analisados, apenas 6% se preocuparam em analisar o conteúdo das soluções encontradas, o que foi feito comparando os resultados obtidos com soluções encontradas por seres humanos a partir dos mesmos datasets.

Assim sendo, após analisar os resultados obtidos, é possível observar quanto ao NRP que:

- a partir de 2009, o interesse por essa área tem se mantido constante e novas abordagens de como resolver esse problema vem surgindo a cada ano. Com esse trabalho foi possível não só determinar quais métodos evolucionários tem sido utilizados para resolver o NRP, mas também que os algoritmos genéticos e otimização por colônia de formigas possuem uma adoção muito superior em relação aos outros métodos (RQ1). Mesmo assim é importante ressaltar que a gama de métodos utilizados é grande, e que novos trabalhos têm sido publicados adaptando algoritmos menos populares e com bons resultados;

[1] *função de fitness: tipo de função objetivo utlizada para medir a qualidade das soluções em uma população em problemas de otimização*

- quanto aos métodos e validação dos resultados (RQ2), apenas 9% envolveram especialistas da área de requisitos para avaliar e comparar a qualidade das soluções encontradas pelos algoritmos. Esse é um número bastante baixo, pois a seleção de requisitos não é algo exato e na prática, tais algoritmos funcionariam como ferramentas de auxílio na tomada final de decisão dos especialistas, principalmente em situações em que existam conflito de interesses entre os *stakeholders* e a empresa. Claro que as comparações estatísticas e de *fitness* das soluções encontradas também são muito importantes e avaliam a performance dos algoritmos, mesmo assim era de se esperar um maior envolvimento de profissionais da área de requisitos nos trabalhos realizados;
- os requisitos utilizados para testar os métodos empregados no NRP são, geralmente, de datasets gerados aleatoriamente (RQ3). Isto porque requisitos de projeto possuem informações sensíveis das empresas, sendo muito difícil conseguir dados reais para as simulações de NRP, dessa forma, 71% dos artigos utilizam datasets gerados aleatoriamente. E como o NRP é complexo de ser resolvido mesmo com poucos requisitos [2], 70% dos projetos se limita a datasets de apenas 200 requisitos.

6. Considerações Finais

Uma das principais lacunas encontradas durante a realização desse trabalho foi a falta de participação de profissionais da área de requisitos para avaliar as soluções encontradas e com isso determinar quão competitivos algoritmos são em relação às decisões tomadas por seres humanos.

Outro ponto que deveria ser mais explorado é a obtenção de datasets reais para aplicação desses métodos, pois assim seria mais fácil verificar a qualidade dos resultados obtidos em relação às decisões humanas.

Por fim, a Engenharia de Software Baseada em Busca é um ramo muito extenso e, dentro dela, a seleção de requisitos é uma área que ainda possui muito potencial a ser explorado.

As conclusões e tendências mencionadas nesse trabalho foram feitas baseadas nos dados disponíveis. É provável que existam mais trabalhos e pesquisas realizados nessa área, que não tenham sido encontrados nas bases de dados pesquisadas nesse trabalho.

7. Referências

[1] J. Thomaschewski, M. J. Escalona e E.-M. Schön, “Agile Requirements Engineering: A systematic literature review,” *Computer Standards & Interfaces*, nº 49, pp. 79-91, 2016.

[2] A. J. Bagnall, V. J. Rayward-Smith e I. M. Whittley, “The Next Release Problem,” *Information and Software Technology*, nº 43, pp. 883-890, 2001.

[3] M. Harman e B. F. Jones, “Search-based software engineering,” *Information and Software Technology*, vol. 43, nº 14, pp. 883-839, 2001.

[4] D. Greer e G. Ruhe, “Software release planning an evolutionary and iterative approach,” *Information and Software Technology*, vol. 46, nº 4, pp. 243-253, 2004.

[5] M. R. Garey e D. S. Johson, Computers and Intractability: a Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.

[6] Y. Zhang, M. Harman e S. A. Mansouri, "The Multi-Objective Next Release Problem," em *9th Annual Conference on Genetic and Evolutionary Computation*, 2007.

[7] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro e A. Alba, "MOCell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, vol. 24, nº 7, pp. 726-746, 2009.

[8] T. N. Ferreira, S. R. Vergilio e J. T. Souza, "Engenharia de Software Baseada em Busca e em Preferência: Uma Visão Geral," em *VII Workshop de Engenharia de Software Baseada em Busca*, Maringá, 2016.

[9] A. M. Pitangueira, R. S. P. Maciel e M. Barros, "Software requirements selection and prioritization using SBSE approaches: A systematic review and mapping of the literature," *Journal of Systems and Software*, vol. 103, nº -, pp. 267-280, 2015.

[10] K. Petersen, R. Feldt, S. Mujtaba e M. Mattson, "Systematic mapping studies in software engineering," em *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, Swinton, UK, 2008.

[11] J. J. Durillo, Y. Zhang, E. Alba, M. Harman e A. J. Nebro, "A study of the bi-objective next release problem," *Empirical Software Engineering*, vol. 16, nº 1, pp. 29-60, 2011.

[12] J. Xuan, H. Jiang, Z. Ren e Z. Luo, "Solving the Large Scale Next Release Problem with a Backbone-Based Multilevel Algorithm," *IEEE Transactions on Software Engineering*, vol. 38, nº 5, pp. 1195-1212, 2012.

[13] C. P., K. Sandahl, M. Lindvall, B. Regnell e J. N. Dag, "An Industrial Survey of Requirements Interdependencies in Software Product Release Planning," em *IEEE Int'l Symp. Requirements Engineering*, 2001.

[14] J. d. Sagrado, I. M. d. Águila e F. J. Orellana, "Ant Colony Optimization for the Next Release Problem," em *2nd International Symposium on Search Based Software Engineering*, Benevento, 2010.

Software Requirements Prioritization using Fuzzy Logic

Dayvison Lima, Raphael Saraiva, Thayse Alencar,
Gustavo Campos, Jerffeson Souza

[1]State University of Ceará (UECE)
Dr. Silas Munguba Avenue, 1700. Fortaleza, Ceará. Brazil
Optimization in Software Engineering Group
http://goes.uece.br
{raphael.saraiva,thayse.alencar}@aluno.uece.br
dayvison@gmail.com
{gustavo,jeff}@larces.uece.br

Abstract. *One of the most critical tasks during a software development project is the requisites prioritization, due to all the imprecise aspects surrounding this task. Its complexity stems from all these aspects and variables of the organization that need to be taken into account, before suggesting the best order for developing the project's requisites. In this context, the application of Fuzzy Logics is a promising strategy to properly deal with such a challenge. This work aims at proposing a framework to support the decision making in the requisite prioritization throughout the software development process. Furthermore, this approach is capable of dealing with data from ambiguous and imprecise sources. The approach is evaluated with a simple example showing the feasibility of a Fuzzy Decision System in the requisites prioritization.*

1. Introduction

Every day, everyone deals with situations where it is necessary to make choices among available alternatives. When the number of possible choices is minimal, this task is solved with little difficulty. However, when the decision involves dozens or hundreds of possibilities, the task becomes intractable for a human.

One way to make a choice more feasible is prioritizing among several alternatives and considering the highest priority. This challenge is also simple when we have fewer aspects to consider in the prioritization process. For example to rent an apartment, only considering the price is a simple decision criterion of choice. However, when other factors are considered, such as the number of rooms, the property condition, the crime rate in the neighborhood, the decision is not as simple as before. During the development of a software product, the choices that must be made follow the same principle, where, for example, the most important functionality for the end users might not be as important to the project sponsor when considering the costs for development. Prioritization helps deal with this complex decision problem [Wohlin et al. 2005].

According to Schulmeyer and Mcmanus (1987), the quality level of a software product is defined by its ability to meet the needs of all the stakeholders. Requirements prioritization is used in the execution of tasks such as helping the stakeholders to decide what are the main requirements of the system, by selecting the requirement sets for implementation in the product's releases and defining the order among the requirements that

meet conflicting goals. Thus, prioritization must be faced as a strategic process, since the implementation order of the product's requirements is capable of defining gain or loss in the market, as described in [Aurum and Wohlin 2003]. Prioritizing requirements can be defined, according to Ruhe et al. (2002), as the challenge of selecting a set of requirements from available sets so that different requirements, such as technical limitation and stakeholders' preferences are met.

Several works in the area of requirements prioritization are divided in several knowledge areas such as Decision Support Systems (DSS), Search Based Software Engineering (SBSE) and Software Engineering itself. In the DSS area, Bellman and Zadeh (1970) propose a quantitative method for prioritization and classification of possible alternatives, considering fuzzy goals and constraints. A fuzzy decision is understood as an intersection between points that represent these alternatives in the defined universe. These fuzzy values are defuzzificated and converted into an integer programming approach. This approach proved to be efficient in the context where it was applied, but it does not take into consideration the evaluation of the alternatives for the stakeholders to improve the final evaluation criterion.

Gaur and Soni (2010) proposed a DSS to help the stakeholders in the analysis of conflicting requirements according to the defined goals and constraints. The requirements that do not conflict with each other are not considered in the evaluation. The goals and constraints are treated as fuzzy sets, represented by triangular functions, while the requirements are represented by numeric weights. This evaluation results in a number, which is used to define the priority level of each requirement. The stakeholders have the same priority level, and hence it is impossible to distinguish the evaluation of someone in the technical area, such as a developer, from the evaluation of someone in the business or strategic area of the organization. Influence of the priority requirements for the goals is another unconsidered factor.

In the SBSE area, Tonella et al. (2010) proposed an interactive GA based method to prioritize the requirements for a software system. The system requires user input when the existing fitness function results in a tie and a precedence dependency is generated in the requirements prioritization process. Precedence relations were represented in a directed acyclic graph and treated as constraints for the ordering of the requirements.

Karlsson (1996) applied the Analytic Hierarchy Process (AHP) methodology, proposed by Saaty (1990), to the requirements prioritization problem [Karlsson 1996]. Its usage simply consists of comparing all the pairs of possible requirements and deciding which one has the higher priority [Saaty 2008]. One of the problems encountered with this methodology is that the time needed to evaluate n requirements is proportional to the number n. Hence, in a large proportion project scenario with many requirements, the methodology may become very inefficient, according to [Karlsson et al. 1998].

So, differently from the search based approach applied to the requirements prioritization problem, common in the context of SBSE, but similar to AHP applications, this work incorporates strategic features pointed by the organization and its stakeholders involved in the decision-making process needed to prioritize software requirements. This work proposes a formal framework and a decision-making process to properly deal with the prioritization task. The framework allows the representation of ambiguous and

imprecise information surrounding the task, using the notions of Fuzzy Sets and Fuzzy Logic.

The paper is organized in five sections. Section 2 presents the proposed approach to the formulated problem. Section 3 presents the proposal evaluation. Finally, Section 4 presents some conclusions and future works.

2. Proposed Approach

Representing the human needs through a language capable of describing all (or most of) the singularities has always been a great challenge. The proposed approach is a framework associated with an algorithm that can solve the software requirements prioritization, and is guided by an information set. The information is described in linguistic terms from the speech of people involved in the development of software and, in other words, in the stage of requirements prioritization. The approaches seen so far try to mathematically model human judgment, by quantifying and attributing it to the formulas of numeric results.

2.1. Fuzzy Sets

Lotfi Asker Zadeh, mathematician and computer scientist of the University of California in Berkeley, thought it would be interesting to model a system by rules described in a natural language. In the 60's, Zadeh proposed a logic different from the classical logic. He introduced fuzzy logic to the world [Zadeh 1965]. This logic is a generalization of Boolean logic. The essential difference between them is the range of values defining the possible degrees of membership. While Boolean logic works with only two symbols (0 and 1), fuzzy logic allows the classification of each attribute through the degrees of membership defined in the set of real numbers.

When using the classical logic, for example, it is easy to classify if a specific element is a member of the set of integer numbers. However, it is more difficult to classify if the element is a member of the set of real numbers near zero. There is an inherent subjectivity in the proposition "near zero". Continuously adding small increments, when will the number no longer be considered near zero? Based on fuzzy logic, it is possible to create fuzzy systems, composed essentially of: input/output fuzzy variables, a knowledge database represented by fuzzy rules, and fuzzy inference mechanisms. Problems, where the inputs and outputs are precise values, also demand fuzzification and defuzzification blocks to transform the absolute values into fuzzy values and vice-versa.

Fundamental to fuzzy logic, according to [Lee 2006], a fuzzy set A in a universe U is the set of ordered pairs of generic elements x and their level of pertinence $\mu_A(x)$. Fuzzy sets can be defined as $\mu_A(x) : U \rightarrow$ [0,1], where the pertinence level is continuous, not just a binary value such as "is in" or "is not in". Pertinence functions may be represented by triangular, trapezoidal, quadratic or Gaussian functions. Linear, triangular and trapezoidal are the most popular, due to the simplicity of implementation, in which the computational cost demanded by other function does not reflect the quality of improvement for the represented values [Yen and Langari 1998].

2.2. Formal Definitions

In the formal framework below, the uppercase letters represent sets, fuzzy sets and fuzzy relationships; $U = 0.0 + 0.1 + ... + ... + 1.0$ is the discrete universe of discourse in which

fuzzy sets are defined; L = [0; 1] is the interval that defines the participation or pertinence values; and $F(U)$ describes a family of fuzzy sets defined in U. A type-2 fuzzy set is a fuzzy set where pertinence values are fuzzy sets defined in U [Nguyen and Walker 2005]. The first part of the framework allows the representation of Fuzzy Goals, Fuzzy Desired Situations and Fuzzy Requirements of a stakeholder. A pair (variable, linguistic aspiration level) defines a Fuzzy Goal. The next definition formalizes the notion of a Fuzzy Goal.

Definition 1: A goal G_m is said to be a Fuzzy Goal if the aspiration level A_m, assigned to an attribute of a desired situation, is described in terms of a fuzzy subset of U:

$$G_m = \left(X_m, A_m = \sum_{i=1}^{Nu} A_m(u_i)/u_i \right) \tag{1}$$

where X_m is the name of the m-th linguistic variable that represents the m-th attribute of a desired situation; Nu is the cardinality of the universe of discourse employed to the definition of the fuzzy sets defining the linguistic values in the set $T(Aspiration)$; A_m is the linguistic value of X_m belonging to the set $T(Aspiration)$ and represents the aspiration level assigned to X_m; $A_m(u_i)$ is the participation level of $u_i \in U$ in the fuzzy set that defines $A_m \in F(U)$.

The values of the linguistic variables $T(Aspiration)$ are sentences in the human language formed by terms that represent aspiration levels as, for example $T(Aspiration)$ = ... low + median + ... + high + ... These levels, in turn, are mathematically represented by Fuzzy sets defined in U, as, for example, the sets low = [1.0, 1.0, 0.9, 0.7, 0.5, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0] and high = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.5, 1.0, 1.0, 1.0], when U is discretized in eleven elements, as proposed in the introduction of this section. Usually, more than one goal is used to specify a desired situation. Each goal contributes a different level to the achievement of the situation. The next definition formalizes the notion of the (strategic) desired situation.

Definition 2: A desired situation is said to be fuzzy if it can be represented by a type-2 fuzzy set in terms of fuzzy goals G_m and fuzzy importance levels I_m for the goals in the desired situation $DS(m = 1, ..., M)$:

$$DS = \sum_{m=1}^{M} I_m/G_m \tag{2}$$

where $I_m \in F(U)$ is a linguistic importance value in $T(Importance)$.

The set of terms in $T(Importance)$ and its associated family of fuzzy sets must be adequately defined, in the same manner as the terms in $T(Aspiration)$. In the same way, considering the desired situation, the most important requirements for the stakeholders are known, as well as the achievements of the fuzzy goals for each of these requirements that must be satisfied. The next definition formalizes the concept of the fuzzy requirement for a stakeholder.

Definition 3: A requirement R_{rs} is said to be fuzzy for a stakeholder s if it is represented by a type-2 fuzzy set described in terms of pairs (X_m, R_{mr}) and fuzzy participation levels $I_{ms}(m = 1, ..., M)$:

$$R_{rs} = \sum_{m=1}^{M} I_{mr}/(X_m, R_{mr}) \tag{3}$$

where X_m is the name of the m-*th* linguistic variable employed in the representation of the m-th attribute of a desired fuzzy situation; $R_{mr} \in F(U)$ are the linguistic values of X_m that belong to the set $T(Aspiration)$ and represent the levels achieved in the attribute X_m in a fuzzy goal G_m, given the implementation of the requirement r, and $I_{ms} \in F(U)$ is the linguistic value of importance in $T(Importance)$, displaying the significance of the requirement r for the stakeholder s.

Similarly, a stakeholder describes the importance values of the desired requirements in the formulation, while the organization describes the importance values associated to the set of stakeholders involved in the prioritization process. The next definition formalizes the notion of a fuzzy organization.

Definition 4: An organization Org is said to be fuzzy if it is represented by a type-2 fuzzy set correlating fuzzy importance values $I_s(s = 1, ..., Ns)$ to each stakeholder St_s in the organization's context:

$$Org = \sum_{s=1}^{N_s} I_s / St_s \tag{4}$$

where $I_s \in F(U)$ is a linguistic importance value in $T(Importance)$.

2.3. Evaluation function for stakeholders' requirements

The approach outlined in this section assumes the existence of a function capable of measuring the similarity between a certain fuzzy situation desired by the organization, according to Definition 2, and the achievements of the goals given by the stakeholders' preferred requirements, according to Definition 3. Thus, this function must take into account the linguistic importance values, in $T(Importance)$, and the achievement values, in $T(Aspiration)$ given by a fuzzy requirement of a particular stakeholder.

The use of this function, on each requirement for every stakeholder in the organization, allows us to obtain an evaluation value that indicates how fit a requirement is to the desired situation for the project, and the "synergy" between the desires of the organization and the stakeholder. Equation 5 shows the evaluation function:

$$f_{rs}(R_{rs}, DS) = \sum_{m=1}^{M} \alpha(D1_{mrs}, D2_m)/M \tag{5}$$

where $D1_{mrs}$ and $D2_m$ are fuzzy relationships in $F(U \times U)$ which represent, respectively, the importance of the achievement given by the requirement r in the goal m for the stakeholder s and the importance of the level for the fuzzy goal of the organization m; and α is the similarity measure between both fuzzy relationships.

In this work we used the similarity measure associated to the Euclidean fuzzy distance defined in Equation 6:

$$\alpha(D1_{mrs}, D2_m) = 1 - \frac{\sqrt{\sum_{i=1}^{Nu} \sum_{j=1}^{Nu} |D1_{mrs}(u_i, u_j) - D2_m(u_i, u_j)|^2}}{Nu} \tag{6}$$

where Nu is the cardinality of the universe of discourse employed to the definition of the fuzzy sets representing the linguistic values in the sets $T(Aspiration)$ and $T(Importance)$, e.g., $U = 0.0 + 0.1 + ... + ... + 1.0$; and $D1_{mrs}$ is the fuzzy relationship between the requirement r and the goal m for the stakeholder s, and $D2_m$ is the fuzzy relationship between the importance of the aspiration level in the organization's goal m,

that is, for all $(ui, uj) \in U \times U$, expressed as :

$$D1_{mrs}(u_i, u_j) = \varphi(I_{ms}(u_i), R_{mr}(u_j)) \tag{7}$$

$$D2_{m}(u_i, u_j) = \varphi(I_{m}(u_i), A_{m}(u_j)) \tag{8}$$

where φ is the pseudo-complement operator: $L \times L \rightarrow L$, where L = [0, 1], presents in Equation 9:

$$\varphi(a, b) = \begin{cases} 1 & if\ a < b \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

Therefore, as the similarity measure between $D1_{mrs}$ and $D2_{m}$ increases, f_s also increases. Thus, for each goal $m = 1, ..., M$, if $D1_{mrs} = D2_{m}$ then f_{rs} increases to its maximum value and equals 1. In the same manner, as the similarity values decrease, f_{rs} also decreases, reaching the minimum values of 0. These properties allow us to evaluate and prioritize the requirements desired by each stakeholder, taking into consideration the organization's desired situation.

Moreover, it is important to highlight that the use of the pseudo-complement operator for the computation of the fuzzy relationships is justified by considering that the value computed by this operator prioritizes the importance values associated to the stakeholders' preferred requirements and the organization's strategic goals, regardless of the achievement levels given by the requirements and the goals.

2.4. Decision-making process for requirements priorization

The decision-making process proposed for the requirements prioritization is described, step-by-step, as a means of an algorithm. As seen in Algorithm 1, the inputs necessary for the requirements prioritization are: (1) m linguistic values of aspiration levels in $T(Aspiration)$, and their associated values of importance in $T(Importance)$, which are specified by the fuzzy *goals* of a fuzzy desired situation for the organization in Definitions 2 and 3; (2) the set of ***stakeholders*** and, for each stakeholder in the set, the linguistic values in $T(Aspiration)$, the effectiveness for each fuzzy requirement to attain the aspiration level for each of the fuzzy goals, and the linguistic values in $T(Importance)$ representing the importance of each requirement for each stakeholder, in Definition 4.

Algorithm 1 Requirements Prioritization

1: **Input:** $requirements(r), goals(g), importance(i), stakeholders(s)$
2: **Output:** $requirements\ sorted\ by\ rating$
3: $GoalImportance \leftarrow compRelGoalImportance(g, i)$
4: $ReqStakeholder \leftarrow compRelReqStakeholders(r, s)$
5: **for** $j = 0$ to $size(r)$ **do**
6: $\quad Differences \leftarrow compDifferences(GoalImportance, ReqStakeholder)$
7: $\quad Similarities \leftarrow compSimilarities(Diferences)$
8: $\quad FuzzyEvaluations \leftarrow compFuzzyEvaluations(Similarities)$
9: $\quad r[j].evaluate \leftarrow compMaxMin(FuzzyEvaluations)$
10: **end for**
11: **return** $requirements\ sorted\ by\ their\ evaluations$

The operations performed in Algorithm 1 are $compRelGoalImportance$ and $compRelReqStakeholder$, lines 3 and 4. Despite these functions being different in the nomenclature, they are quite similar in how they work. They apply the pseudo-complement operator φ defined in Equation 9 to the evaluated sets. In the case of the relationship between goals and importance, the operator will be applied to the desired

value in a goal and its respective importance. In the relationship between requirements and stakeholders, the operator will be applied to each achievement value of the requirement in the goals with respect to the importance of the requirement evaluated by the stakeholder.

The next steps, from lines 5-10, perform an individual evaluation for each requirement. The first operation is $compDifferences$. This procedure computes the Euclidean distance between the fuzzy relationships, which represent the stakeholders' requirements, and the fuzzy relationships, which represent the goals and importance of a desired situation for the organization. The operation $compSimilarities$, line 7, computes the similarity between the fuzzy sets defined in Equation 6, employing the values of the differences computed in the previous step. Each similarity value represents an initial ranking position for each evaluated requirement. The higher the similarity value, the higher the fitness of the requirement with respect to the organization's strategic goals and the opinions of a specific stakeholder, considering the effectiveness of the requirement to attain the strategic goals.

The next two steps of the Algorithm 1, lines 8-9, finally rank the set of evaluated requirements considering the set of stakeholders. In other words, the algorithm implements a fuzzy inference. The output is a linguistic value describing the final ranking position of an evaluated requirement and inferred by the conjunction of two linguistic input values: (1) the level of similarity describing the ranking of any requirement computed for any stakeholder; and (2) the importance of any stakeholder in the context of the organization. The fuzzy system requires a set of rules describing the specialized knowledge for the relations between the two input and the output linguistic variables, and thereby guiding the final evaluation.

The result of the fuzzy inference performed on the input values is a fuzzy number that, after defuzzification, is converted into a real value in the interval [0,1] and is returned by the function $compFuzzyEvaluations$. The last operation performed by the Algorithm 1 is an aggregation operation on the values of the fuzzy evaluations for each requirement. In line 9 of Algorithm 1, we have the operation $compMaxMin$, whose result is a real value in the interval [0,1] that determines the final evaluation of the requirement. After the execution of all steps of the algorithm, the requirements are evaluated and ready to be ordered according to their priority value.

3. Proposal Evaluation

3.1. The Experiment Settings

This first evaluation tried to study the relationship mainly between the fuzzy goals of a desired situation and the capacity of the fuzzy requirements to attain the situation. Two instances were used in the experiments. The importance values of the fuzzy goals are treated as real values in the interval [0,1] for the first instance. On the other hand, the second values are treated as fuzzy sets, in the same manner as the fuzzy goals and their importance values, fuzzy requirements and the importance of each stakeholder in the context of the fuzzy organization.

Table 1 presents three stakeholders, their importance values in the context of the organization and the importance values for nine requirements of each stakeholder. The

linguistic values in the table, such as vl, l, m, h and hv, represent respectively, very low, low, medium, high and very high. A linguistic value zero has been added to represent when a requirement has no relevance to the stakeholder.

Table 1. Description of Stakeholders

Stakeholders	Importance	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
St_1	h	h	h	vl	0	m	h	m	m	m
St_2	l	h	m	m	h	0	l	h	h	l
St_3	m	h	h	0	h	0	h	h	h	0

Table 2 shows a scenario with seven fuzzy goals (G_m), associated with seven attributes (X_m). The table presents the aspiration level and the importance value (μ_m) of each goal, a real value in instance-1 and a fuzzy value in instance-2, respectively. Table 2 also illustrates the achieved level in each attribute in the goals for each requirement (R_m).

Table 2. Description of each fuzzy requirement, contribution level for achievement of each goal and desired situation of each goal.

Goals (X_m)	X_1	X_2	X_3	X_4	X_5	X_6	X_7
Importance (μ_m)	0.6	0.1	0.4	0.2	0.4	0.8	0.6
Desired Situation (DS)	m	h	l	h	l	h	m
R1	m	h	l	h	l	h	m
R2	h	m	h	m	h	l	l
R3	h	h	m	h	l	m	h
R4	h	l	m	h	l	l	h
R5	m	l	l	m	h	h	m
R6	h	l	m	h	l	m	h
R7	l	m	m	h	l	h	h
R8	m	m	m	h	l	h	h
R9	h	m	m	h	l	h	h

(a) instance-1

Goals (X_m)	X_1	X_2	X_3	X_4	X_5	X_6	X_7
Importance (μ_m)	l	h	h	m	l	l	h
Desired Situation (DS)	vh	h	l	h	l	h	m
R1	vh	h	l	h	l	h	m
R2	h	m	h	m	vl	b	b
R3	b	m	b	b	b	b	b
R4	h	l	m	h	l	l	h
R5	m	l	vl	m	h	h	m
R6	h	vl	vh	h	l	m	vl
R7	b	m	m	h	vh	h	h
R8	vl	m	m	h	b	h	h
R9	h	vl	vh	vl	vh	vl	vl

(b) instance-2

The example displayed in Tables 1-2 is related to the development of a system for Automatic Teller Machines (ATMs). For example, the attribute X_1 represents "Deliver all high-priority features in the product's first release" and has a median (m) desired or aspiration level in fuzzy goal X_1. The requirements R_4 and R_7 represent respectively "make a withdrawal" and "display welcome message". And the requirement R_4 achieves the goal X_1 in a high (h) level, as indicated in Table 2.

The objective of the evaluation is to check the ability of the framework to perform the prioritization task. The next section evaluates the framework, making changes in the information of the requirements and goals in the tables. More information about the linguistic terms used in the experiments can be found on the paper's webpage[1].

3.2. Results

The results of the two tests for instance-1 and three tests for instance-2 are depicted in Table 3. The normalized real value in each cell of the table describes how suitable a requirement is for a strategic goal in a desired fuzzy situation. After obtaining a list of requirements prioritized according to the information in Tables 2-3, in each of the following tests, the information in certain cells of the tables were modified in order to perceive the sensitivity of the approach with respect to the proper reordering of the list of requirements.

[1] goes.uece.br/raphaelsaraiva/fuzzy4req/

Table 3. Prioritized lists generated from each test.

control		test 1		test 2	
R_1	1.00	R_5	0.88	R_1	1.00
R_5	0.68	R_1	0.80	R_8	0.69
R_8	0.64	R_8	0.67	R_5	0.63
R_3	0.57	R_7	0.58	R_3	0.62
R_7	0.55	R_6	0.57	R_7	0.61
R_9	0.53	R_9	0.56	R_9	0.59
R_6	0.37	R_4	0.56	R_6	0.45
R_4	0.36	R_3	0.37	R_4	0.44
R_2	0.26	R_2	0.29	R_2	0.26

(a) Results for instance-1

control		test 3		test 4		test 5	
R_1	0.72	R_1	0.72	R_1	0.72	R_1	0.92
R_6	0.63	R_5	0.60	R_5	0.72	R_7	0.76
R_5	0.60	R_4	0.57	R_6	0.64	R_8	0.76
R_4	0.56	R_7	0.57	R_4	0.57	R_4	0.75
R_7	0.56	R_8	0.57	R_7	0.57	R_5	0.74
R_8	0.56	R_6	0.52	R_8	0.57	R_6	0.72
R_3	0.44	R_2	0.39	R_3	0.45	R_3	0.64
R_2	0.39	R_9	0.39	R_2	0.39	R_9	0.55
R_9	0.39	R_3	0.38	R_9	0.39	R_2	0.54

(b) Results for instance-2

In Table 3 (a), the control test illustrates the results when the capacity of the fuzzy requirement R_1 was deliberately made to equal the aspiration levels described in the fuzzy desired situation under consideration. In test 1, on the other hand, the aspiration level in the fuzzy goal X_2 of the fuzzy desired situation was changed from high to low. In test 2, there was a change in the importance value of the fuzzy goal G_4, from 0.2 to 0.8.

As expected, in the control test, R_1 achieved 100% similarity with the goals, since this requirement exactly achieves the desired fuzzy situation [m, h, l, h, l, h, m]. This result was also seen in Test 2. There was a reordering among the requirements in Tests 1 and 2, which was affected by the changes in the aspiration level and importance value. In test 2, for example, no requirement reached a maximum similarity level, since with the change in the aspiration in goal G_2, the requirement R_1, which achieves [m, h, l, h, l, h, m], does not achieve the new desired fuzzy situation [m, l, l, h, l, h, m]. In test 2, the requirement R_1 stayed on the top of the list, and the increase in the importance level of G_4 induced the new reordering. If all requirements were exactly equal to the desired situation, such as R_1, there would be 100% similarity in all cases.

In Test 3, detailed in Table 3(b), the importance of desired values was changed to only one value of [l, h, h, m, **l**, l, h] to [l, h, h, m, **vh**, l , h], thus indicating that the fuzzy goal G_5 has gone from being "low" to "very high". Again, this modification changed the evaluation values, leading to a reordering in most of the requirements involved. These results are expected, given that all requirements contribute in some value to the achievement of the goal in question G_5.

Test 4 shows the result when the importance values of the requirements for stakeholders in Table 1 are changed. By modifying the importance value of the requirement R_5 by stakeholder St_2 from "Zero" to "High", [h, m, m, h, **0**, l, h, h, h] to [h, m, m, h, **h**, l, h, h, l], the final evaluation value of the requirement increased from 0.60 to 0.72. This result is expected, since the evaluation of one requirement does not interfere with the evaluation of the others. It is important to note that the evaluation values of the other requirements remained the same.

Finally, Test 5 demonstrates how the importance values associated to the stakeholders, in the context of the organization, influence the requirements prioritization. In this test, the importance value of the stakeholder St_2 was changed from "Low" [l] to "High" [h] in Table 1. This operation modified the evaluation of all requirements, creating a new order in the prioritization list, according to the results shown in Table 3 (b).

4. Conclusion

Requirements prioritization is a hard task. Previous works have dealt with the problem in the software development process without considering the uncertainty level present in the

discourse of stakeholders in an organization. This work proposes a framework guided by fuzzy goals and fuzzy requirements that is intended to address the prioritization problem.

Besides presenting the framework, examples for evaluating its feasibility were described. The results show that the approach is very flexible and suitable for prioritization. As seen in the tests, different configurations for the information of the domain problem lead to different and new suitable solutions. Thus, the formalized approach is able to represent the needs of those involved in the software development process, in a manner which is similar to the human language, in other words, forming the evaluation on the basis of the organization's strategic goals, the known capacity requirements needed to achieve the goals, the requirements evaluation of the stakeholder and their importance for the software in development.

After this initial testing, the approach will still need to be adapted to solve the prioritization problem in a real project. From this basis, the formal framework will be extended aiming at addressing models of interdependence between the levels of aspiration and satisfaction in the fuzzy goals of desired situations, and dealing with different types of interdependence between the fuzzy requirements involved.

References

Aurum, A. and Wohlin, C. (2003). The fundamental nature of requirements engineering activities as a decision-making process. *Information and Software Technology*, 45(14):945–954.

Bellman, R. E. and Zadeh, L. A. (1970). Decision-making in a fuzzy environment. *Management science*, 17(4):B–141.

Gaur, V. and Soni, A. (2010). An integrated approach to prioritize requirements using fuzzy decision making. *International Journal of Engineering and Technology*, 2(4):320.

Karlsson, J. (1996). Software requirements prioritizing. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pages 110–116. IEEE.

Karlsson, J., Wohlin, C., and Regnell, B. (1998). An evaluation of methods for prioritizing software requirements. *Information and software technology*, 39(14-15):939–947.

Lee, K. H. (2006). *First course on fuzzy theory and applications*, volume 27. Springer Science & Business Media.

Nguyen, H. T. and Walker, E. A. (2005). *A first course in fuzzy logic*. CRC press.

Ruhe, G., Eberlein, A., and Pfahl, D. (2002). Quantitative winwin: a new method for decision support in requirements negotiation. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 159–166. ACM.

Saaty, T. L. (1990). How to make a decision: the analytic hierarchy process. *European journal of operational research*, 48(1):9–26.

Saaty, T. L. (2008). Relative measurement and its generalization in decision making why pairwise comparisons are central in mathematics for the measurement of intangible factors the analytic hierarchy/network process. *Revista de la Real Academia de Ciencias Exactas, Fisicas y Naturales. Serie A. Matematicas*, 102(2):251–318.

Schulmeyer, G. and McManus, J. I. (1987). *Handbook of software quality assurance*. Van Nostrand Reinhold Co.

Tonella, P., Susi, A., and Palma, F. (2010). Using interactive ga for requirements prioritization. In *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pages 57–66. IEEE.

Wohlin, C. et al. (2005). *Engineering and managing software requirements*. Springer Science & Business Media.

Yen, J. and Langari, R. (1998). *Fuzzy logic: intelligence, control, and information*. Prentice-Hall, Inc.

Zadeh, L. A. (1965). Fuzzy sets. *Information and control*, 8(3):338–353.

Utilizando Função de Escalarização para Controle da Relevância de Métricas de Software

Italo Yeltsin[1], Allysson Allex Araújo[2], Raphael Saraiva[1], Jerffeson Souza[1]

[1]Universidade Estadual do Ceará (UECE)
Fortaleza, Ceará - Brasil.
Grupo de Otimização em Engenharia de Software da UECE

[2]Universidade Federal do Ceará (UFC)
Cratéus, Ceará - Brasil.
{italo.medeiros,raphael.saraiva}@aluno.uece.br
{allysson.araujo}@crateus.ufc.br
{jerffeson.souza}@uece.br

Abstract. *SBSE research field aims to formulate Software Engineering Problems (SE) as search problems. Through the definition of a fitness function, the search for an optimal or sub-optimal solution can be guided. When there is one or more metrics associated with a SE problem, constructing a fitness function capable of assessing solutions quality for such a problem is possible. However, it is recurrent assumptions about fitness functions that provide justice relation between metrics, i.e., when every single metric has exactly the same relevance for the search process. Several properties related to metric value domain might induce the search process to offer privileges to a certain metric bringing out worser outcomes to another ones. Thus, this work aims to proposes a generic mathematical model that uses scalarizing function in order to obtain control over the relevance of each metric in the search process. Regarding the empirical study, it was noticed that the proposed approach is capable of providing a fair relation between metrics better than the original fitness functions proposed in the investigated problems.*

Resumo. *SBSE é a área de pesquisa que objetiva formular problemas de Engenharia de Software (ES) como problemas de busca. Através da definição de uma função de avaliação, a busca por uma solução ótima ou sub-ótima pode ser guiada. Quando há a existência de uma ou mais métricas associadas a um problema de ES, é possível compor uma função de avaliação capaz de mensurar a qualidade das soluções para tal problema. Todavia, é recorrente nas formulações o pressuposto de que uma função de avaliação estabelece uma relação justa entre as métricas, ou seja, todas as métricas têm as mesma relevância no processo de busca. As diversas características quanto ao domínio dos valores das métricas podem influenciar o processo de busca a privilegiar uma métrica em detrimento de outras. Assim, o presente trabalho objetiva propor um modelo matemático genérico que utiliza função de escalarização como forma de obter o controle da relevância das métricas no processo de busca. Em termos de estudo empírico, averiguou-se que a abordagem proposta é capaz de estabelecer uma relação mais justa entre as métricas do que as funções de avaliação originalmente propostas para os problemas investigados.*

1. Introdução

A Engenharia de Software Baseada em Busca (em inglês *Search Based Software Engineering* - SBSE) objetiva reformular problemas da Engenharia de Software como problemas de busca. Um problema de busca é caracterizado por apresentar um espaço de busca onde são procuradas soluções ótimas ou sub-ótimas as quais são qualificadas através de uma função de avaliação ou função *fitness*. Portanto, a função *fitness* reflete os objetivos a serem otimizados e, consequentemente, a qualidade da solução [Harman 2007].

Um passo inicial para definição da função *fitness* é averiguar a existência de alguma métrica que mensure a qualidade do que se pretende otimizar. Uma métrica de software é um padrão de medida de um grau para o qual um sistema ou processo de software possui alguma propriedade. Se tal métrica já existe previamente, torna-se factível utilizá-la como forma de avaliar os resultados de uma abordagem e, inclusive, comparar os resultados obtidos através de outras técnicas [Harman et al. 2012].

Nesse contexto, muitos problemas na engenharia de software possuem métricas associadas as quais apresentam-se como candidatas naturais para composição da função *fitness* [Harman and Clark 2004]. Tal característica permite que as métricas sejam tratadas como um componente crucial para mudança, deixando de ser apenas uma avaliação passiva do sistema, processo ou produto [Harman et al. 2009]. Dentre os exemplos de métricas frequentemente adotadas em SBSE, pode-se mencionar *Percentage of Faults Detected* (APFD) e Cobertura para Testes de Software, Coesão e Acoplamento para Design de Software e risco do requisito para o Planejamento de *Releases*.

Entretanto, uma problemática recorrente na análise dos resultados de SBSE é o pressuposto de uma função *fitness* que estabeleça uma relação justa entre todas as métricas. Isto é, uma função que permita que as métricas tenham a mesma relevância no processo de busca, evitando assim, que uma métrica tenha privilégio (sem ter sido explicitado) indevido em detrimento da outra. Para estabelecimento de tal relação, torna-se necessário uma análise consciente, por exemplo, da natureza e o intervalo de valores que cada métrica pode assumir, visando assim, evitar discrepâncias quanto à avaliação de cada métrica que compõe a função *fitness*. Naturalmente, tais discrepâncias podem prejudicar a interpretação dos resultados.

É nesse contexto que a presente pesquisa se insere. Este trabalho objetiva (i) propor um modelo matemático genérico baseado no conceito de função de escalarização e (ii) avaliar o controle do referido modelo sob o impacto das métricas no processo de busca em constraste às funções de avaliação originais empregadas em problemas mono-objetivos de SBSE. O conceito de função escalarização é comumente utilizado na resolução de problemas multicritérios ou multi-objetivos, onde através da utilização dessa função pode-se otimizar um problema multi-objetivo de forma mono-objetiva [Miettinen and Mäkelä 2002].

O restante do trabalho é organizado da seguinte forma: na Seção 2 apresenta-se a abordagem proposta; na Seção 3 define-se os detalhes referentes ao estudo empírico, incluindo as modelagens matemáticas dos problemas investigados; na Seção 4 discute-se os resultados alcançados. Finalmente, na Seção 5 destacam-se as considerações finais e trabalhos futuros.

2. Abordagem Proposta

Seja A um determinado problema de otimização tendo a ele associado um conjunto de métricas $M = \{m_1(X), m_2(X), ..., m_n(X)\}$, onde X é uma solução qualquer de A. M pode ser decomposto em dois conjuntos, M_{max} e M_{min}, tal que ambos conjuntos tenham elementos distintos, $M_{max} \cap M_{min} = \emptyset$, e a união de ambos resulte em M, isto é, $M_{max} \cup M_{min} = M$. O primeiro conjunto M_{max} representa o conjunto de métricas de A a serem maximizadas, enquanto M_{min} representa o conjunto de métricas a serem minimizadas. Assim, inspirado no conceito de função de escalarização, o modelo genérico proposto neste trabalho consiste em:

$$\min \sum_{m_i(X) \in M_{min}} \alpha_i \times \frac{m_i(X) - menor_{ij}}{maior_i - menor_{ij}} + \sum_{m_i(X) \in M_{max}} \alpha_i \times \frac{m_i(X) - maior_{ij}}{menor_{ij} - maior_{ij}}$$

sujeito a:

$$g_i(X) \qquad i = 1, 2, 3, ..., m,$$

onde α_i representa a relevância que a métrica m_i terá no processo de busca. As variáveis $menor_{ij}$ e $maior_{ij}$ representam respectivamente o menor e o maior valor da métrica m_i conhecidos na iteração j da técnica de busca, ou, por exemplo, da população na geração j de um Algoritmo Genético. A função $g_i(X)$ representa uma restrição do conjunto de m restrições do problema A. Assim, para uma dada solução, os valores de cada métrica estarão normalizados no intervalo [0,1] de acordo com seu $menor_{ij}$ e $maior_{ij}$, os quais são atualizados a cada iteração. Como pode-se notar na função a ser minimizada (primeiro fator), para as métricas a serem minimizadas, a parcela referente a uma métrica $m_i(X)$ tende a 0 quando o valor dessa métrica é mais próximo ao $menor_{ij}$, para métrica m_i na iteração j. A mesma parcela tende a 1, quando m_i tem valor próximo do maior valor para métrica m_i na iteração j, ou seja, $maior_{ij}$.

Para as métricas a serem maximizadas (segundo fator), inverte-se. O valor da parcela tende a 0, quando a métrica referente tem valor próximo do $maior_{ij}$ ou a 1 quando o valor é próximo do $menor_{ij}$. No que se refere a relevância de cada métrica como guia no processo de busca, tem-se o peso α_i como ajuste proporcional. Isto é, quão maior for o peso para uma métrica m_i, maior a relevância da métrica.

Tratando-se da relevância que cada métrica exerce no processo de busca, essa é proporcional ao peso α_i, quanto maior em relação ao outros pesos, mais relevância a métrica m_i terá. Portanto, uma opção que garante relevância equilibrada entre as métricas seria é $\alpha_1 = \alpha_2 = ... = \alpha_n$.

3. Estudo Empírico

Este estudo empírico foi realizado a partir de dois problemas de SBSE, sendo eles: (i) Priorização de *Bugs* baseado na proposta de Dreyton et al. (2015) e (ii) Planejamento de Releases inspirado na modelagem desenvolvida por Dantas et al. (2015) . Em termos de questões de pesquisa, definiu-se:

QP1: A abordagem proposta apresenta melhor controle da relevância de cada métrica no processo de busca em relação às funções *fitness* originalmente utilizadas?

QP2: A abordagem proposta é capaz de melhorar a média dos valores das métricas e de *fitness* em relação às funções de avaliação originais?

Para responder a **QP1** averiguou-se a capacidade da abordagem de manter todas as métricas com a mesma relevância. Nesse sentido, este trabalho propõe a métrica *Média da Diferença Absoluta* (MDA) a qual representa a média de diferença absoluta das métricas normalizadas, conforme demonstrado a seguir:

$$MDA(X) = \frac{2 \times \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} |(\|m_i(X)\| - \|m_j(X)\|)|}{n(n-1)}, \quad (1)$$

onde $\|m_i(X)\|$ é o valor normalizado da métrica m_i em relação ao seu pior e melhor valor, caso ela fosse a própria função *fitness*. Por exemplo, para conseguir o pior e o melhor valor de uma métrica, bastaria executar duas vezes um algoritmo de busca empregando apenas essa determinada métrica como função objetivo: uma execução referente à maximização da métrica e outra referente à minimização. Ao final desse passo, o valor normalizado de $m_i(X)$ é dado por $\|m_i(X)\| = \frac{m_i(X) - pior_i}{melhor_i - pior_i}$. Quanto mais próximo de zero for o valor de MDA de uma solução, mais similares são as relevâncias de cada métrica entre si no processo de otimização. Por exemplo, um MDA igual a 0,5 indica que, em média, as métricas diferem em relevância em 50%.

Em relação a investigação da **QP2**, o presente estudo desenvolveu outra métrica denominada *Média das Métricas Normalizadas* (MMN). A intenção é justamente verificar se a abordagem proposta atinge um MMN superior aos valores obtidos pelas funções *fitness* originais referentes aos problemas investigados. A métrica MMN é dada por:

$$\overline{MMN(X)} = \sum_{i=1}^{n} \frac{\|m_i(X)\|}{n} \quad (2)$$

3.1. Problema de Priorização de Bugs

Conforme mencionado anteriormente, este trabalho investiga como parte de seu estudo empírico o Problema de Priorização de *Bugs* através da modelagem proposta por Dreyton et al. (2015) . Tal problema consiste em encontrar a melhor ordenação de *bugs* a serem corrigidos dentro de repositórios de software livre e código aberto baseando-se em um conjunto de métricas a serem otimizadas.

Considere $B = \{b_i \mid i = 1, 2, 3, ..., N\}$ o conjunto de *bugs* reportados, no qual N representa o número de *bugs* disponíveis para priorização presentes no repositório. Como representação da solução, utiliza-se um vetor de elementos do conjunto B, com ordenação específica, $P = \{p_j \mid j = 1, 2, 3, ..., M\}$, no qual M é um parâmetro previamente definido que representa o número de *bugs* a serem apresentados em uma solução.

A função $relevância(P)$ busca atender ao desejo dos membros da comunidade mantenedora pela resolução de um determinado conjunto de *bugs*. O valor de relevância referente a uma solução é definido por:

$$relevância(P) = \sum_{i=1}^{N} votes_i \times isIn(P, b_i); \quad (3)$$

onde $votos_i$ representa a relevância que o *bug* b_i tem para os membros da comunidade denotado pelo valor normalizado de votos obtidos. A função $isIn(P, bi)$ indica quando b_i está em P, retornando 1 se $b_i \in P$, e 0 caso contrário.

A função $importância(P)$ estimula a resolução antecipada de *bugs* considerados

de maior prioridade pelo responsável pela triagem dos *bugs* e pode ser representada por:

$$importância(P) = \sum_{i=1}^{N} prioridade_i \times (M - pos(P, b_i) + 1) \times isIn(P, b_i); \quad (4)$$

onde $prioridade_i$ indica a prioridade dada ao *bug* b_i e $pos(P, bi)$ retorna a posição do *bug* b_i no vetor P se $b_i \in P$, e ∞ caso contrário. A posição que um *bug* b_i ocupa em P reflete diretamente no valor de $importância(P)$, sendo que a função aumenta conforme os *bugs* com maior prioridade são antecipados em P.

A função $severidade(P)$ estimula a correção antecipada de *bugs* considerados mais severos, reduzindo o perigo para o projeto causado pelo possível adiamento da resolução de *bugs* críticos. Essa função é dada por:

$$severidade(P) = \sum_{i=1}^{N} severidade_i \times pos(P, b_i) \times isIn(P, b_i); \quad (5)$$

onde $severidade_i$ é o valor de gravidade atribuído pelos usuários para um *bug* b_i. Um menor valor de $severidade_i$ é alcançado quando os *bugs* com maior gravidade são alocados em posições iniciais da solução.

Assim, a modelagem mono-objetiva utilizada consiste em:

$$\begin{array}{ll} \text{maximizar} & (\alpha \times relevância(P) + \beta \times importância(P) - \gamma \times severidade(P)), \\ \text{sujeito a:} & pos(P, bi) < pos(P, b_j), \text{ if } b_i \prec b_j \text{ and } b_j \in P. \end{array} \quad (6)$$

onde α, β e γ são pesos utilizados para ponderar cada função de acordo com o cenário enfrentado. A restrição definida nesse problema é referente à precedência técnica entre *bugs*, onde a correção de um bug depende diretamente da resolução de outro.

3.2. Problema do Planejamento de Releases

Como segundo e último problema avaliado no presente estudo empírico realizado por este trabalho, adaptou-se a modelagem proposta por Dantas et al. (2015) para o problema do Planejamento de *Releases* (PR). O PR consiste na definição de quais requisitos serão implementados em quais *releases* de um software a ser desenvolvido em um processo incremental [Ruhe et al. 2004].

Considere o conjunto de requisitos $R = \{r_i \mid i = 1, 2, 3, ..., N\}$ disponíveis para serem selecionados para o conjunto de *releases* $K = \{k_j \mid j = 1, 2, 3, ..., P\}$, onde N e P são o número total de requisitos e *releases*, respectivamente. Como representação da solução, utiliza-se o vetor $S = \{x_1, x_2, ..., x_N\}$ onde $x_i \in \{0, 1, ..., P\}$, de modo que se $x_i = 0$ o requisito r_i não está alocado para alguma *release*, caso contrário, o requisito está alocado para uma *release* k_p, sendo $p = x_i$. Assim a função *fitness* é definida como:

$$Fitness(S) = \sum_{i=1]}^{N} y_i \times (valor_i \times (P - x_i + 1) - risco_i \times x_i); \quad (7)$$

onde $y_i \in \{0, 1\}$ é 1 se o requisito r_i foi alocado em alguma *release*, isto é, $x_i \neq 0$, e 0 caso contrário. O valor $risco_i$ é definido como o risco associado ao requisito r_i causado pelo possível adiamento de sua implementação. Por fim, $valor_i$ contém a soma ponderada da importância especificada por cada cliente c_j para um requisito r_i a qual é calculada como:

$$valor_i = \sum_{j=1}^{M} w_j \times importância(c_j, r_i); \quad (8)$$

Portanto, a valor de $fitness(S)$ é maior a medida que os requisitos com valores elevados de $valor$ e $risco$ são implementados nas *releases* iniciais.

Em relação às restrições do problema, primordialmente cada requisito r_i possui um custo $custo_i$ e cada *release* k_p tem um valor de orçamento s_q a ser obedecido. Tal restrição é formalizada abaixo:

$$\sum_{i=1}^{n} custo_i \times f_{i,q} \leq s_q, \forall q \in 1, 2, ..., P; \quad (9)$$

Assim, a proposta mono-objetiva empregada consiste em:

$$\begin{aligned} &\text{maximizar} \quad Fitness(S), \\ &\text{sujeito a:} \quad \sum_{i=1}^{n} custo_i \times f_{i,q} \leq s_q, \forall q \in 1, 2, ..., P. \end{aligned} \quad (10)$$

Por fim, ressalta-se que na abordagem proposta por Dantas *et al.* (2015) o $Fitness(S)$ é penalizado de acordo com o nível de importância de preferências subjetivas não satisfeitas estabelecidas por um tomador de decisão. Porém, para efeito de avaliar apenas apenas as métricas próprias do PR, este trabalho optou por adaptar a modelagem original e excluir tal aspecto relacionado às preferências subjetivas, focando assim, somente nas métricas *valor* e *risco*.

3.3. Configuração do Estudo Empírico

O algoritmo de busca escolhido para avaliar a abordagem proposta foi o Algoritmo Genético (AG). Devido ao caráter estocástico do referido algoritmo, para cada instância e cada abordagem, foram realizadas 30 execuções, coletando-se ao final de cada execução o valor que a solução atingiu para cada métrica do problema avaliado. Além disso, visando coletar o melhor e pior valor das métricas, foram realizadas 30 execuções tendo cada métrica como objetivo a ser maximizado e minimizado, coletando-se o pior e o melhor de todas as execuções. As configurações adotadas para AG são expostas na Tabela 1.

Tabela 1. Configurações do AG para os problemas de Priorização de *Bugs* e Planejamento de Releases

Parâmetro	Problema	
	Priorização de Bugs	**Planejamento de Releases**
Operador de Crossover	Order One	Single Point
Operador de Mutação	Rank Swap Mutation	Random Resetting
Taxa de Crossover	90%	90%
Taxa de Mutação	20%	1%
Tamanho da População	100	100
Número de Gerações	1000	1000

Em relação aos operadores utilizados, o operadores *Order One*, *Single Point* e *Random Resetting* são operadores amplamente reconhecidos na literatura. No entanto, o operador *Rank Swap Mutation* foi desenvolvido na presente pesquisa para o Problema

de Priorização de *Bugs* e consiste em uma variação do *Swap Mutation*. Por restrições de espaço a explicação desse operador foi omitida e o código em Java referente ao mesmo está disponível no seguinte repositório de código[1].

Conforme verificou-se nas modelagens empregadas para cada problema, existem diferentes parâmetros a serem configurados a priori. Para o problema de Priorização de *Bugs*, há os pesos α, β, γ os quais ponderam respectivamente as métricas $relevância$, $importância$ e $severidade$. Neste trabalho considerou uma configuração em que as métricas supostamente apresentam relevâncias iguais, ou seja, $\alpha = \beta = \gamma = 1$.

Em relação as instâncias, foram utilizadas para o Planejamento de *Release* o *dataset-1* referente a um processador de textos contendo 50 requisitos e o *dataset-2* com 25 requisitos o qual representa um sistema de planejamento de releases. Tais instâncias tambems foram avaliadas por Dantas et al. (2015) . Para o Problema de Priorização de *Bugs*, avaliou-se o *dataset_inst100* composto por 100 *bugs* reportados no repositório do software Kate Editor o qual foi disponibilizada por Dreyton et al. (2015) . Destaca-se assim que todas as instâncias utilizada no presente trabalho são baseadas em dados reais.

Com o intuito de mostrar se há diferença estatística referente à métrica MDA entre utilizar a função de escalarização e as funções *fitness* originais dos trabalhos a serem avaliados, foi realizado o teste estatístico Wilcoxon (WC) [Mann and Whitney 1947] considerando um nível de confiança de 99%. Além disso, com o intuito de calcular o tamanho de efeito de uma amostra em relação a outra foi utilizado o teste Vargha-Delaney $\hat{A}_{12}$ [Vargha and Delaney 2000]. O tamanho de efeito $\hat{A}_{12}$ representa a quantidade relativa de vezes que um elemento de uma amostra (1) supera elementos em outra amostra (2). Por exemplo, $\hat{A}_{12} = 0.2$ implica que em 20% da vezes um valor da amostra 1 supera valores da amostra 2. No presente trabalho, (1) representa a Métrica MDA utilizando a função de escalarização e (2) a função de avaliação original.

3.4. Resultados e Análises

A Tabela 2 mostra os valores para cada métrica utilizada para o PR utilizando a abordagem proposta (Escalarização) e a função de avaliação original (*Fitness*), bem como os melhores e piores valores alcançados no caso que cada métrica fosse a própria função *fitness*. Como pode-se observar, utilizando o modelo matemático proposto (Escalarização), para a instância *dataset-1* a métrica *Risco* tem seu valor melhorado em 47,57% para uma perda de *Valor* de 7,05%. Em relação ao *dataset-2*, houve um ganho de 50,34% na métrica *Risco* para uma perda de *Valor* de 8,19%.

Na Tabela 3 são mostrados os resultados em relação ao *MDA*, assim como os valores normalizados de cada métrica. Além disso, também é mostrado o valor de tamanho de efeito referente à métrica *MDA* ($\hat{A}_{12}$) com um dos possíveis símbolos $\triangle$, ▲, $\triangledown$ e ▼, dos quais os dois primeiros indicam respectivamente que, a abordagem proposta em relação à abordagem original, obteve média maior sem e com diferença estatística e os dois últimos indicam média menor sem e com diferença estatística. Conforme pode-se verificar, os valores de MDA para o *dataset-1* decaem, em média, 65,80% com diferença estatística e alta magnitude. Considerando o *dataset-2*, há um decréscimo de 67,40% e, igualmente à instância anterior, os resultados indicam que há diferença estatística com alta magnitude utilizando a abordagem proposta.

[1] https://github.com/ItaloYeltsin/SelectionFactor/

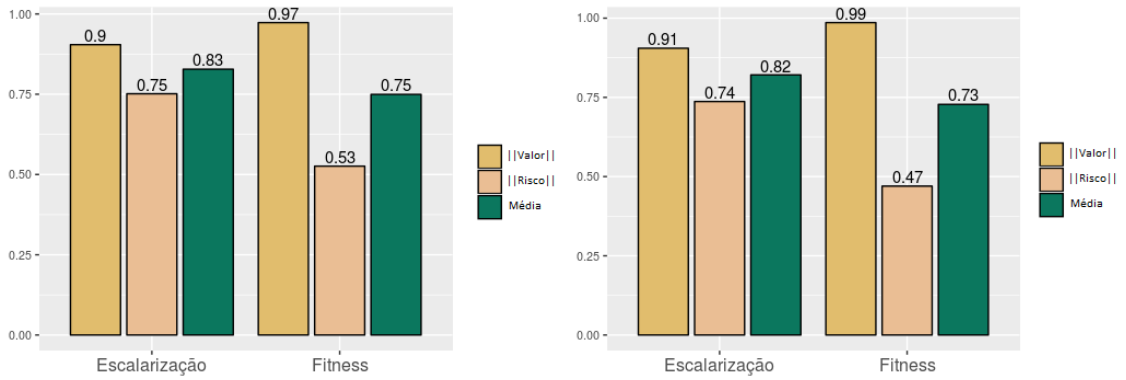
Tabela 2. Média aritmética e desvio-padrão dos valores alcançados para cada métrica por cada abordagem para o Planejamento de *Release*

Instance	Função	Valor	Risco
dataset-1	**Escalarização**	22062.03±743.59	271.13±24.73
	Fitness	23736.57±463.19	517.13±36.96
	Melhor	24394	0
	Pior	0	1091
dataset-2	**Escalarização**	34773.43±1173.72	183.97±22.95
	Fitness	37878.93±409.28	370.53±21.67
	Melhor	38422	0
	Pior	0	699

Tabela 3. Média aritmética e desvio-padrão dos valores normalizados para cada métrica e MDA para o Planejamento de *Release*, incluindo o tamanho do efeito entre as abordagens avaliadas

Instância	Função	|Valor|	|Risco|	MDA	$\hat{A}_{12}$
dataset-1	**Escalarização**	0.9044±0.0305	0.7515±0.0227	0.1529±0.0491	0 ▼
	Fitness	0.973±0.019	0.526±0.0339	0.447±0.0323	
dataset-2	**Escalarização**	0.905±0.0305	0.7368±0.0328	0.1682±0.0604	0 ▼
	Fitness	0.9859±0.0107	0.4699±0.031	0.516±0.0258	

Objetivando-se obter uma melhor visualização dos dados, a Figura 1 expõe os valores normalizados de cada métrica e a média desses valores referentes a cada instância do Planejamento de *Release*. Como pode-se observar, para as duas instâncias há um favorecimento para métrica *Valor* utilizando a função *fitness*. Em contraste, a abordagem proposta consegue melhorar o valor da métrica *Risco* com uma pequena perda de *Valor*. Como resultado, a média dos valores das métricas aumentam de 0,75 para 0,83, considerando o *dataset-1* e 0,73 para 0,82 para o *dataset-2*. Tais resultados implicam que em média, para duas instâncias, as métricas conseguem 83,5% de seus melhores valores utilizando a abordagem proposta e 74% utilizando a função de avaliação original.



(a) Dataset-1 **(b)** Dataset-2

Figura 1. Valores normalizados e médias das métricas para cada instância do Planejamento de *Release*

No que se refere ao problema de Priorização de *Bugs*, a Tabela 4 mostra os resultados de forma análoga à Tabela 2. A partir da observação dos resultados, observa-se que a métrica *Relevância* tem um ganho de 30,1% para um perda de 7,68% de *Importância* e 18,67% de *Severidade*. Em relação à métrica *MDA*, a Tabela 5 mostra seus valores para cada abordagem, bem como os valores normalizados de cada métrica. Assim, constata-se que há uma redução de 57,84% de *MDA* com diferença estatística e magnitude alta quando comparada a abordagem proposta e a função *fitness* originalmente estabelecida.

Tabela 4. Valores alcançados para cada métrica por cada abordagem para o problema de Priorização de *Bugs*

Instância	Função	Importância	Relevância	Severidade
dataset_inst100	**Escalarização**	351.28±8.27	14.09±0.29	111.45±5.69
	Fitness	380.51±8.76	10.83±0.44	93.91±3.98
	Melhor	441.7	16.08957529	70.15
	Pior	93.5	2.954054054	408.05

Tabela 5. Valores normalizados alcançados para cada métrica e *MDA* para o problema de Priorização de *Bugs Release*

Instância	Função	||Importância||	||Relevância||	||Severidade||	MDA	$\hat{A}_{12}$
dataset_inst100	**Escalarização**	0.7403±0.0237	0.8475±0.0219	0.8778±0.0168	0.0927±0.0228	0 ▼
	Fitness	0.8243±0.0252	0.5999±0.0337	0.9297±0.0118	0.2199±0.0223	

Por fim, a Figura 2 ilustra os resultados obtidos para o problema de Priorização de *Bugs*, demonstrando assim, um comportamento similar ao discutido sobre a Figura 1. Em média as métricas atingem 82,18% dos seus melhores valores em contraste com a função de avaliação original que obteve 78,46%.

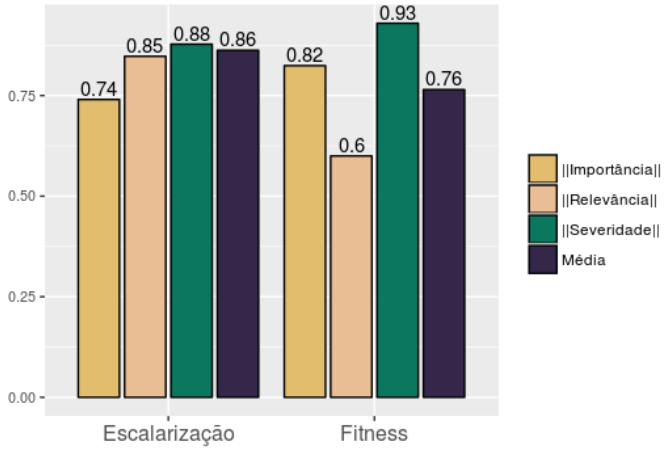


Figura 2. Valores normalizados e média aritmética das métricas para o problema de Priorização de *Bugs*, incluindo o tamanho do efeito

Diante da análises dos resultados, pode-se concluir que, respondendo a **QP1**, a abordagem proposta baseada no conceito de escalarização consegue manter um controle melhor da relevância de cada métrica no processo de otimização, visto que os valores de MDA são menores, para todas as instâncias de todos os problemas. Tal conclusão é reforçada devido os valores de *p-value* e tamanho de efeito obtidos, os quais demonstram que há diferença estatística com alta magnitude. Respondendo a **QP2**, constatou-se que

apesar da perda de valor, o ganho atingido nas demais métricas torna compensador utilizar a abordagem proposta do que as funções de avaliação originais. Tal comportamento é constatado através da média superior dos valores das métricas denotados pelo MMN.

4. Considerações Finais

Diversos trabalhos em SBSE propõem funções de avaliação compostas por métricas de software associadas aos problemas que se deseja resolver. Naturalmente, tais métricas variam em diversas propriedades quanto ao domínio ao qual seus valores pertencem, por exemplo. Essa diversidade entre os valores das métricas podem induzir o processo de busca a privilegiar uma métrica em detrimento de outras. Para lidar com tal impasse o presente trabalho propõe um modelo matemático genérico que faz uso de uma função de escalarização com o intuito de obter um maior controle sobre a relevância que cada métrica em relação ao processo de busca.

Através do estudo empírico realizado, buscou-se averiguar se a abordagem proposta é apta a obter resultados satisfatórios em relação às funções originais propostas para os problemas de Priorização de Requisitos e Planejamento de *Releases*. Para a avaliação dos experimentos foram propostas duas métrica denominadas *Média da diferença Absoluta* (MDA) e *Média das Métricas Normalizadas* (MMN). Os resultados obtidos demonstraram que no geral a abordagem proposta consegue fornecer uma relação mais justa entre as métricas e aumentar em média seus valores.

Em termos de trabalhos futuros, pretende-se avaliar a abordagem em problemas multi-objetivos e realizar um estudo empírico envolvendo outras técnicas de busca, assim como outros problemas da Engenharia de Software.

Referências

Dantas, A., Yeltsin, I., Araújo, A. A., and Souza, J. (2015). Interactive software release planning with preferences base. In *International Symposium on Search Based Software Engineering*, pages 341–346. Springer.

Dreyton, D., Araújo, A. A., Dantas, A., Freitas, Á., and Souza, J. (2015). Search-based bug report prioritization for kate editor bugs repository. In *International Symposium on Search Based Software Engineering*, pages 295–300. Springer.

Harman, M. (2007). The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society.

Harman, M. and Clark, J. (2004). Metrics are fitness functions too. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 58–69. IEEE.

Harman, M., Mansouri, S. A., and Zhang, Y. (2009). Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*.

Harman, M., McMinn, P., De Souza, J. T., and Yoo, S. (2012). Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*, pages 1–59. Springer.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

Miettinen, K. and Mäkelä, M. M. (2002). On scalarizing functions in multiobjective optimization. *OR Spectrum*, 24(2):193–213.

Ruhe, G. et al. (2004). Hybrid intelligence in software release planning. *International Journal of Hybrid Intelligent Systems*, 1(1-2):99–110.

Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

Genetic Programming-based Composition of Fault Localization Heuristics

Diogo M. de Freitas[1], **Plínio S. Leitão-Júnior**[1],
Celso G. Camilo-Junior[1], **Altino Dantas**[1], **Rachel Harrison**[2]

[1]Universidade Federal de Goiás (UFG)
Instituto de Informática (INF)
Alameda Palmeiras, Quadra D, Câmpus Samambaia. Goiânia - Goiás - Brasil

[2]Oxford Brookes University
Department of Computing and Communication Technologies
Wheatley Campus, OX33 1HX. Wheatley - Oxford – United Kingdom

Grupo Intelligence for Software (i4Soft)
http://ic1.inf.ufg.br/i4soft
{diogofreitas,plinio,celso,altinoneto}@inf.ufg.br
rachel.harrison@brookes.ac.uk

Resumo. *Localização de defeitos baseada em espectro avalia suspeita de elementos de código através de heurísticas inspiradas em modelos probabilísticos e estatísticos. O presente trabalho propõe uma abordagem baseada em Programação Genética (PG) para combinar 18 heurísticas existentes em uma fórmula especializada para um projeto. A abordagem foi avaliada em versões de 7 programas do conjunto da Siemens e comparada a 19 outras abordagens – 18 heurísticas genéricas e uma técnica de combinação. Os resultados mostram que a proposta supera as demais. Portanto, conclui-se que PG é uma metaheurística promissora para composição de fórmulas de localização de defeitos.*

Abstract. *Spectrum-based fault localization techniques assess code elements suspiciousness with heuristics inspired by probabilistic and statistic models. This paper proposes a Genetic Programming (GP) approach to combine 18 existing spectrum-based fault localization heuristics into a project oriented fault localization formula. We evaluated the proposal with the 7 programs from the Siemens Suite with single-bugs and compared it against 19 approaches – 18 coverage-based heuristics and one combination approach. The results show that the proposal outperformed the others. Thus, we conclude the GP is a promising metaheuristic to formulate customized fault localization heuristics.*

1. Introduction

Fault localization (FL) is one of the most important tasks in the software development and maintenance life cycle, since faults are constantly introduced and repaired and they impact software quality and cost directly. FL is the process that identifies the code location of the fault associated to a failure observed during testing. As the complexity of software projects grew, FL has become increasingly onerous and time consuming [Hailpern and Santhanam 2002].

A recent survey on FL by Wong *et al.* (2016) covered 331 papers published between 1977 and 2014 and highlighted the increase of publications after 2001. It indicated that there has been a major interest in FL in the last 10 years with the establishment of the theme in major events and journals. Also, publications where divided in eight categories: slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques.

This paper falls under the Spectrum-based category, which are the techniques inspired by probabilistic and statistical models. A program spectrum is a set of information related to its execution behaviour, *i.e.* it is possible to relate software elements to the failure occurrence (software elements commonly refer to a specific granularity, such as line of code or block of statements). These data are collected at run-time, and may consist of a number of counters or flags for the different parts of a program [Abreu et al. 2006].

The objective of a *Spectrum-based Fault Localization (SFL)* heuristic is to identify which elements are responsible for the existing failures based on the program spectra of both successful and failed executions [Lucia et al. 2010]. For each element n, the SFL heuristic calculates a suspiciousness score $S(n)$, that represents the strength of association between an element's executions and failure occurrences. Once every $S(n)$ is calculated, a list can be organized in descending order so that the developer can analyze the elements from top (greater suspiciousness score) to bottom until all faults are located.

As mentioned earlier, many heuristics were proposed based on coverage variables, such as Tarantula [Jones et al. 2002] and Ochiai [Abreu et al. 2006] but their relative performance to locate faults can vary between different software tools. Wang *et al.* (2011) presented a novel method for the linear combination of known SFL heuristics for a customized heuristic. The authors called this approach "*Search-based Fault Localization*" since the heuristic composition problem was treated as a search problem.

Since Wang *et al.* (2011) restricted their heuristic combination to a linear form and we expect that non-linear combination fits the problem's complexity better, this paper proposes the use of Genetic Programming metaheuristic to compose SFL heuristics.

Similarly to Wang *et al.* (2011) , the Siemens suite was used, as obtained in the Software-artifact Infrastructure Repository [Do et al. 2005]. The algorithm was applied to a training set of faulty versions of programs and then evaluated by a test set, using the same methodology as Wang *et al.* (2011) . Thus, the main contributions of this paper are: A method to compose SFL heuristics based on non-linear combination of others heuristics; An extensive experiment with 107 programs and 19 comparable methods.

The paper is structured as follows. Section 2 presents the related works; Section 3 describes the proposed method; Section 4 presents the evaluation methods applied; Section 5 presents the results of the proposal and of many others approaches [Wang et al. 2011]; and Section 6 concludes and presents future works.

2. Related Work

As software projects complexity and size grew, the use of manual techniques for FL became impractical. This motivated the development of techniques to allow automating the FL process in a way that the human intervention was not necessary [Wong et al. 2016].

A tool called *Tarantula* was proposed for visualization of suspicious code by Jones

et al. (2002) . A suspiciousness value was calculated with Equation (1) to each element of code e in order to indicate which elements may contain a fault. The number of successful and failed tests is represented by n_s and n_f respectively, The number of successful and failed executions of element e is represented by $n_s(e)$ and $n_f(e)$ respectively.

$$S(e) = \frac{n_f(e)/n_f}{n_s(e)/n_s + n_f(e)/n_f} \tag{1}$$

The *Ochiai* coefficient was adapted from molecular biology into a SFL heuristic (see Equation (2)) by Abreu *et al.* (2006) . The study indicated that Ochiai outperformed other three heuristics, including Tarantula, in the Siemens suite of programs.

$$S(e) = \frac{n_f(e)}{\sqrt{n_f \times (n_f(e) + n_s(e))}} \tag{2}$$

Another important piece of research in SFL heuristics is an investigation of twenty association measures based on dichotomy matrices by Lucia *et al.* (2010) . Association measures are a tool used in statistics to measure strength of association between two variables; for instance, in our context the association is between the execution (or not) of a software element and the test result (pass or fail). Lucia *et al.* compared the association measures to Tarantula and Ochiai, two well established SFL heuristics. They have indicated that ten of the measures are statistically comparable with Tarantula and Ochiai.

2.1. Search-based Fault Localization

Wang *et al.* (2011) proposed a search-based model to build composite linear heuristics ($H_{Composite}(e)$) based on metaheuristics. The composite heuristic is a linear combination of two SFL heuristics (Tarantula and Ochiai) and the association measures from Lucia *et al.* , *i.e.* each heuristic H_n is multiplied by a corresponding weight w_n and the sum is the composite heuristic, as noted in Equation 3.

$$H_{Composite}(e) = \sum_{i=1}^{n} w_i \times H_i(e) \tag{3}$$

Wang *et al.* (2011) applied three different metaheuristics (Simulated Annealing and two variants of Genetic Algorithm) to figure out the right weights w_n for the Equation (3). Thus the search space was defined by all combinations of w_n in a 7 bit representation. The fitness function was the average proportion of software elements that need to be investigated to locate all faults in a set of programs; a software element is ranked based on suspiciousness scores obtained by the heuristic combination (individual). The methodology was structured in two phases: *training* and *test*. During training, a set of programs variants with faults in known locations are used to search a composite heuristic with GP. In the test phase, the best composite heuristic found by the training phase is used to locate the faults within the test set of programs.

In addition to the original Search-based Fault Localization method, there are other evolutionary initiatives for building and analyzing suspiciousness metrics. Yoo (2012)

introduced a Genetic Programming (GP) approach for evolving risk assessment formulae using only coverage variables, which were empirically evaluated by using 92 faults distributed in faulty versions of four programs. However, the heuristics were built in a non-project oriented fashion, *i.e.* they are software-independent, that is, the heuristics are applied to find faults in any type of software.

3. Approach

Genetic Programming (GP) was proposed by Koza (1992) as a way to search for the fittest individual computer program (or mathematical formulae) in the space of all possible appropriate computer programs (or mathematical formulae) structured as a tree where the internal nodes are functions and the leafs (external nodes) are terminals [Koza 1992].

In a GP algorithm, a population of computer programs (each one is an individual) is bred over many generations using the Darwinian principle of natural selection – *survival and reproduction of the fittest* – in addition to several natural operations such as: crossover, mutation, gene duplication and gene deletion. The initial population is comprised of randomly generated individuals formed by the appropriate functions and terminals. The functions may be standard arithmetic operations, programming operations, mathematical functions, logical functions or domain-specific functions [Koza 1992].

On the FL problem each individual is coded in a tree structure that represents a candidate suspiciousness formula. Hence the population is a set of solutions which evolve according to the GP algorithm aiming to achieve better equations to calculate how suspicious is each program element to be faulty. Thus the search space is the whole set of all valid formulae composed by the functions and the terminals selected to initiate the GP algorithm. Note that non-linear equations are commonly obtained from this process.

Considering the problem of composition of SFL heuristics, Genetic Programming can be applied to search for the fittest composite heuristic, which is also represented as a mathematical formula. The search space is defined by all possible formulae composed by the individual heuristics (terminals) discussed in Section 2 and basic mathematical operations (functions). Due to the non-linear nature of the obtained formulae, this search space includes the one defined by Wang *et al.* (2011) , as it is composed by linear formulae in the form of Equation 3 (the latter is a subset of the first). Hence it is expected that better compositions will be found.

This paper presents a development over the "Search-based Fault Localization" method by Wang *et al.* (2011) through application of Genetic Programming. Hence the composition is not limited by linear equations and more complex formulae are allowed.

The heuristics used as terminals by the GP algorithm are listed in Table 1. They consist of sixteen of the association measures studied by Lucia *et al.* (2010) along with the Tarantula and Ochiai heuristics. In order to avoid normalization, four association measures that have the maximum suspiciousness score as infinite were excluded.

The association measures in Table 1 (H_1 to H_{16}) [Lucia et al. 2010] are written in a different notation from the standard SFL heuristics, with $P(A)$ the probability of event A occuring. In the context of FL, the relevant events are E (execution of element e), $\overline{E}$ (non execution of element e), F (observation of a failure) and $\overline{F}$ (successful runs of the program). Events are combined as $P(E, F)$ and $P(E|F)$ corresponding respectively to

the probabilities of joint probability of E and F and the conditional probability of E if F occurs. Also, probabilities can be replaced by the calculation of observed frequencies of the events E, $\overline{E}$, F and $\overline{F}$.

Table 1. Association measures studied used by the composition algorithms.

ID	Name	Formula
H_1	ϕ-Coefficient	$\frac{P(E,F)-P(E)\times P(F)}{\sqrt{P(E)\times P(F)\times(1-P(E))\times(1-P(F))}}$
H_2	Yule's Q	$\frac{P(E,F)\times P(\overline{E},\overline{F})-P(E,\overline{F})\times P(\overline{E},F)}{P(E,F)\times P(\overline{E},\overline{F})+P(E,\overline{F})\times P(\overline{E},F)}$
H_3	Yule's Y	$\frac{\sqrt{P(E,F)\times P(\overline{E},\overline{F})}-\sqrt{P(E,\overline{F})\times P(\overline{E},F)}}{\sqrt{P(E,F)\times P(\overline{E},\overline{F})}+\sqrt{P(E,\overline{F})\times P(\overline{E},F)}}$
H_4	Kappa	$\frac{P(E,F)+P(\overline{E},\overline{F})-P(E)\times P(F)-P(\overline{E})\times P(\overline{F})}{1-P(E)\times P(F)-P(\overline{E})\times P(\overline{F})}$
H_5	J-Measure	$max\Big(P(E,F)\times\log\Big(P(F\|E)/P(F)\Big)+P(E,\overline{F})\times\log\Big(P(\overline{F}\|E)/P(\overline{F})\Big),$ $P(E,F)\times\log\Big(P(E\|F)/P(E)\Big)+P(\overline{E},F)\times\log\Big(P(\overline{E}\|F)/P(\overline{E})\Big)\Big)$
H_6	Gini Index	$max(P(E)\times[P(F\|E)^2+P(\overline{F}\|E)^2]+P(\overline{E})\times[P(F\|\overline{E})^2+P(\overline{F}\|\overline{E})^2]-P(F)^2-P(\overline{F})^2,$ $P(F)\times[P(E\|F)^2+P(\overline{E}\|F)^2]+P(\overline{F})\times[P(E\|\overline{F})^2+P(\overline{E}\|\overline{F})^2]-P(E)^2-P(\overline{E})^2)$
H_7	Support	$P(E\|F)$
H_8	Confidence	$max(P(F\|E),P(E\|F))$
H_9	Laplace	$max\Big(\frac{P(E,F)+1}{P(E)+2},\frac{P(E,F)+1}{P(F)+2}\Big)$
H_{10}	Cosine	$\frac{P(E,F)}{\sqrt{P(E)\times P(F)}}$
H_{11}	Piatetsky-Shapiro's	$P(E,F)-P(E)\times P(F)$
H_{12}	Certainty Factor	$max\Big(\frac{P(F\|E)-P(F)}{1-P(F)},\frac{P(E\|F)-P(E)}{1-P(E)}\Big)$
H_{13}	Added Value	$max(P(F\|E)-P(F),P(E\|F)-P(E))$
H_{14}	Jaccard	$\frac{P(E,F)}{P(E)+P(F)-P(E,F)}$
H_{15}	Klosgen	$\sqrt{P(E,F)}\times max(P(F\|E)-P(F),P(E\|F)-P(E))$
H_{16}	Information Gain	$(-P(F)\times\log P(F)-P(\overline{F})\times\log P(\overline{F}))-$ $(P(E)\times(-P(F\|E)\times\log P(F\|E))-P(\overline{F}\|E)\times\log P(\overline{F}\|E)-$ $P(\overline{E})\times(-P(F\|\overline{E})\times\log P(F\|\overline{E}))-P(\overline{F}\|\overline{E})\times\log P(\overline{F}\|\overline{E}))$
H_{17}	Tarantula	Equation 1 (Section 2)
H_{18}	Ochiai	Equation 2 (Section 2)

The set of functions allowed in the GP's solutions are the ones observed in the heuristics of Table 1: sum, subtraction, multiplication, division, exponentiation, logarithm, square root and maximum function. The terminal set is: numerical constant 1 (one) and independent variables for each of the heuristics in Table 1.

The fitness function employed by the GP algorithm is the following: given the rank of program elements generated from the suspiciousness scores obtained by the composite heuristic under evaluation, the fitness function is the average proportion of program elements that need to be investigated to locate all faults in programs.

The proposed method is also structured in a training phase and a test phase. The training starts from a set of faulty programs spectra and the actual faults' location. The Genetic Programming algorithm then iterates 1000 generations with a population of 100 individuals and returns the best composition formulae found. In the test phase the suspiciousness of each element is calculated by the GP composite heuristic for every program

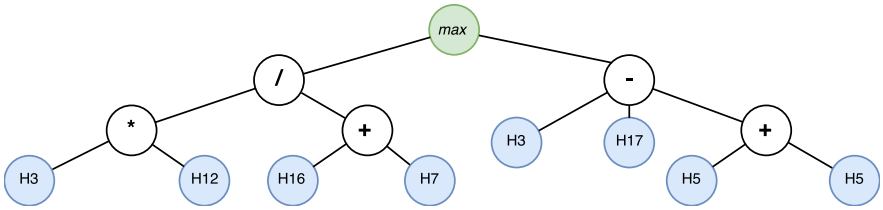


Figure 1. Tree representation of a best solution found by the Genetic Programming method: *max((H$_3$ * H$_{12}$) / (H$_{16}$ + H$_7$), H$_3$ - H$_{17}$ - (H$_5$ + H$_5$))*.

in the test set separately with the elements of higher suspiciousness investigated first.

Figure 1 shows a solution trained with three versions of the program *printtokens* from the Siemens Suite and tested with one buggy version. Only one line of code was investigated to locate the fault (0.51% of the total). The solution is represented as a tree with the functions in the inner nodes and the terminals in the leaf nodes.

4. Experiments

The programs used in the experiments were obtained from the Software-artifact Infrastructure Repository (SIR), a project created to distribute software-related artifacts for experimentation with program analysis, software testing, and education [Do et al. 2005]. The programs obtained from the SIR were the seven programs from the Siemens Suite and are listed in Table 2 [Hutchins et al. 1994]. These programs were selected because they are also used by Wang *et al.* (2011) and Lucia *et al.* (2010) .

Table 2. Program set used in the experiments.

Program	printtokens	printtokens2	replace	schedule	schedule2	tcas	tot_info
LOC	472	399	512	292	301	141	440
Faulty Versions	7 (4 used)	10	32 (28 used)	9 (7 used)	10 (9 used)	41 (30 used)	23 (19 used)
Test Set	4030	4115	5542	2650	2710	1608	1051

For each program, there are several faulty versions with a known single fault. In alignment with the aforementioned publications, some versions were not used in the experiments because the fault was in a non-instrumented line (e.g. variable declaration bug), they are: 4 and 5 from *printtokens*; 12 from *replace*; 13, 14, 15, 36 and 38 from *tcas*; and 6, 10, 19 and 21 from *tot_info*. In addition, some versions were not used because either the test set did not reveal the fault or the fault was not located in a single line of code, they are: 1 from *printtokens*; 6, 21 and 32 from *replace*; 2 and 7 from *schedule*; 9 from *schedule2*; and 10, 11, 31 ,32, 33 and 40 from *tcas*.

Line of code was used as the granularity to locating faults as it is more precise than coarse groupings (e.g. block of statements and method). The spectra for the remaining 107 programs were generated with the tools *lcov*[1], *gcov* (a GNU standard test coverage tool that comes with GCC compiler) and a custom program. Finally, the Java framework JGAP[2] was used to construct the evolutionary algorithms.

The parameters applied in both the GA and the GP algorithms were obtained from preliminary tests, they as follows: generations: 1000; population: 100 individuals; cros-

[1] `http://ltp.sourceforge.net/coverage/lcov.php`
[2] `http://jgap.sourceforge.net/`

sover probability: 1.0; mutation probability: 0.08; new chromosomes percentage: 0.3; reproduction probability: 0.1; maximum crossover depth: 20; maximum initial depth: 10; minimum initial depth: 2; probability that a function instead of a terminal is chosen in crossing over: 0.9.

4.1. Method Evaluation

To better compare and assess the performance of the composite heuristics produced by GP against the ones produced by other researcher's method and the standard FL and association measures, three evaluation metrics were used:

- **Average Score:** Quantifies the average effort to locate all existent faults according to the suspicious rank [Gong et al. 2015]. The metric is calculated, as follows:

$$\textit{Avg. Score} = \frac{\textit{Rank}}{\textit{Number of executed statements}} \times 100\%$$

- **Accuracy (acc@n):** Counts the number of faults within the top n positions of the suspiciousness rank [B. Le et al. 2016]. We use 1, 3. 5 and 10 for n.
- **Wasted Effort (wef@n):** Counts the number of non-faulty program elements that had to be investigated before an actual fault was located in the top n positions of the suspicious rank [B. Le et al. 2016]. We again use 1, 3, 5 and 10 for n.

We performed a three-fold cross validation. The program set is randomly divided into three groups with two groups used for training (training phase) and the remaining one used for testing (test phase). For each training set 30 iterations of experiments are performed so that each group of programs is used equally in training.

5. Results

Table 3 summarizes the results of the Genetic Programming method (GP) and compares them against the original "Search-based Fault Localization" method implemented with the Genetic Algorithm (GA) and the best performing standard heuristic. The Table shows results for each of the seven Siemens programs and an overall mean result, the number of faulty versions (one fault per version), Average Score (lower values are better), acc@n (higher values are better) and wef@n. Also, the GP and GA results represent an average of 30 runs for the three-fold cross validation.

The GP composed formulae had better mean results for all evaluation metrics. The Average Score (16.43%) was better than the GA's (21.79%) and than the best standard heuristic (27.05%). Also, the GP obtained better acc@n and wef@n, with 11.24% of all faults in the first position of the ranking and 53.80% of all faults in the first ten positions.

In programs *schedule*, *schedule2* and *tcas* the GP composed heuristics were also better in Average Score and either better or very close behind in acc@n and wef@n. In the program *tot_info* GP and GA are closely matched in Average Score, acc@n and wef@n, with both superior to the best standard heuristic, as we can see in Table 3.

The GP-trained heuristics lost to the others in *printtokens*, *printtokens2* and *replace*. The poor results in *printtokens* are understandable because of the small training set, since it was observed that when a good fitness function was found in the training set, its value during testing phase would decline.

Table 3. Comparison of the best performing standard heuristic and composite heuristics by GA and GP.

Program	Method	N faults	Avg. Score	acc@1	acc@3	acc@5	acc@10	wef@1	wef@3	wef@5	wef@10
	GP	4	9.44%	0.70	1.93	2.47	3.23	3.30	8.10	11.43	16.07
printtokens	GA	4	3.21%	0.00	2.00	3.00	3.00	4.00	9.00	11.00	16.00
	H_{18}	4	3.46%	0.00	3.00	3.00	4.00	4.00	8.00	10.00	10.00
	GP	10	21.84%	2.23	3.87	4.33	4.60	7.77	20.10	31.60	59.13
printtokens2	GA	10	11.08%	2.10	4.37	6.00	6.00	7.90	19.53	28.53	48.53
	H_{10}	10	18.42%	2.00	4.00	4.00	4.00	8.00	20.00	32.00	62.00
	GP	28	10.38%	5.67	9.90	11.37	18.57	22.33	60.47	94.07	152.00
replace	GA	28	8.45%	7.07	10.63	11.57	20.00	20.93	58.07	90.93	143.30
	H_{18}	28	10.78%	5.00	10.00	10.00	17.00	23.00	61.00	97.00	156.00
	GP	7	5.77%	0.87	3.73	5.53	5.93	6.13	14.63	18.23	24.13
schedule	GA	7	9.24%	0.90	3.10	3.13	3.13	6.10	15.23	22.97	42.30
	H_1	7	10.54%	0.00	1.00	2.00	2.00	7.00	19.00	30.00	55.00
	GP	9	30.69%	1.40	1.73	1.83	3.17	7.60	22.20	36.63	68.33
schedule2	GA	9	51.23%	0.00	0.00	0.00	1.00	9.00	27.00	45.00	87.00
	H_1	9	54.09%	0.00	0.00	0.00	1.00	9.00	27.00	45.00	88.00
	GP	30	21.77%	0.00	3.07	5.80	11.87	30.00	85.87	136.10	239.67
tcas	GA	30	40.34%	0.00	5.00	5.07	10.27	30.00	80.03	129.90	237.77
	H_{18}	30	44.26%	0.00	5.00	5.00	11.00	30.00	81.00	131.00	229.00
	GP	19	12.69%	1.17	3.47	6.60	10.20	17.83	50.63	77.57	128.20
tot_info	GA	19	12.44%	1.00	3.97	6.00	8.77	18.00	50.03	77.03	130.27
	H_5	19	18.51%	2.00	3.00	5.00	7.00	17.00	50.00	79.00	145.00
	GP	107	16.43%	12.03	27.70	37.93	57.57	94.97	262.00	405.63	687.53
Mean	GA	107	21.79%	11.07	29.07	34.77	52.17	95.93	258.90	405.37	705.17
	H_{10}	107	27.05%	7.00	24.00	27.00	36.00	100.00	272.00	434.00	795.00

5.1. Statistical Analysis

Considering the stochastic behavior of Genetic Algorithms and Genetic Programming techniques, we performed statistical tests over the metric Score Average to define when and how large the difference is between the algorithms.

Table 4 presents the information of the statistical tests Wilcoxon and Vargha and Delaney's Â [Arcuri and Briand 2014]. The column "Test Set" shows three distinct test set from each program is shown. All values of *p-value* are reported and all values in boldface represent that significant statistical difference was achieved (at 0.05 significance level). The last columns presents the effect-size results comparing GA versus GP and the magnitude of the difference between them.

Looking at the Table, we notice that the statistical difference was not verified in just two cases of all 21 comparisons. In such cases, the value of magnitude was "Negligible", confirming the Wilcoxon test. These results supply more confidence for the previous analysis based on the mean values of Score Average.

Specifically analyzing *printtokens*, the magnitude was "Large" in all three comparisons, with values close to 1 in the two first cases and close to 0 in the third one. Knowing that low values of Score Average are preferred and V-D $\hat{A}_{12}$ values near 1 indicate that the results from GA are almost always higher than values from GP, the conclusion is that the GP was significantly better on two test sets and worse in one.

By the same sense, we can say that GP outperforms GA also in *schedule*, *schedule2* and *tcas* (these last two programs with a large advantage on the three sets). Meanwhile, GA won in *printtokens2* and *replace* and ties occurred in *tot_inf*.

Table 4. Wilcoxon and Vargha and Delaney's $\hat{A}_{12}$ tests with GA (A_1) and GP (A_2).

Program	Test Set	Wilcoxon *p-vaule*	*V-D* $\hat{A}_{12}$	Magnitude
printtokens	1	$\mathbf{2.86 \times 10^{-11}}$	0.97	Large
	2	$\mathbf{1.46 \times 10^{-12}}$	0.98	Large
	3	$\mathbf{1.27 \times 10^{-8}}$	0.12	Large
printtokens2	1	$\mathbf{8.37 \times 10^{-5}}$	0.20	Large
	2	$\mathbf{3.17 \times 10^{-2}}$	0.35	Small
	3	$\mathbf{4.07 \times 10^{-8}}$	0.09	Large
replace	1	$\mathbf{2.54 \times 10^{-8}}$	0.08	Large
	2	$\mathbf{1.56 \times 10^{-4}}$	0.22	Large
	3	$\mathbf{2.31 \times 10^{-2}}$	0.33	Medium
schedule	1	$\mathbf{4.68 \times 10^{-10}}$	0.93	Large
	2	$\mathbf{1.49 \times 10^{-8}}$	0.90	Large
	3	$\mathbf{4.14 \times 10^{-7}}$	0.87	Large
schedule2	1	$\mathbf{8.63 \times 10^{-12}}$	1.00	Large
	2	0.67	0.47	Negligible
	3	$\mathbf{2.21 \times 10^{-11}}$	1.00	Large
tcas	1	$\mathbf{1.04 \times 10^{-11}}$	1.00	Large
	2	$\mathbf{6.49 \times 10^{-12}}$	1.00	Large
	3	$\mathbf{2.10 \times 10^{-6}}$	0.85	Large
tot_info	1	$\mathbf{4.71 \times 10^{-11}}$	0.01	Large
	2	0.86	0.51	Negligible
	3	$\mathbf{1.09 \times 10^{-5}}$	0.83	Large

5.2. Threats to validity

Regarding internal validity, we used algorithms that have many parameters and different settings could be achieved other results, however, we performed some preliminary tests and present the values of the most relevant parameters that were adopted. We also employed external tools and provided short descriptions of them. As external validity, a benchmark with synthetic programs was used, thus, the proposal may have a different behavior when applied to programs with others characteristics.

About the construct validity, we based our empirical study on some metrics which exist and properly provided their references. Finally, the conclusion validity may be influenced by stochastic nature of GA and GP, nevertheless, we performed statistical tests of significance and effect-size.

6. Conclusion

Fault Localization for many projects is an onerous and time-consuming task. So some works address the problem and propose heuristics to automate or aid the process. Thus, this paper proposes a novel approach to composition of previous Spectrum-based Fault Localization heuristics with Genetic Programming. Previous work has applied metaheuristics to the same problem, but with a limited search space. The GP-based method allows for complex formulae and better adaptation to a training set of programs.

With few generations and small populations (low cost) a good composite formula could be found. Overall, the GP evolved formula outperformed the standard heuristics as well as the original "Search-based Fault Localization" Genetic Algorithm-based technique. An expected development of this research is to investigate the performance and adaptability of the method on larger, multi-bug and more complex projects.

References

Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v. (2006). An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA. IEEE Computer Society.

Arcuri, A. and Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.*, 24(3):219–250.

B. Le, T.-D., Lo, D., Le Goues, C., and Grunske, L. (2016). A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 177–188, New York, NY, USA. ACM.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435.

Gong, P., Zhao, R., and Li, Z. (2015). Faster mutation-based fault localization with a novel mutation execution strategy. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10.

Hailpern, B. and Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12.

Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA. IEEE Computer Society Press.

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA. ACM.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Lucia, Lo, D., Jiang, L., and Budi, A. (2010). Comprehensive evaluation of association measures for fault localization. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Timișoara, Romania. IEEE Computer Society.

Wang, S., Lo, D., Jiang, L., Lucia, and Lau, H. C. (2011). Search-based fault localization. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 556–559, Washington, DC, USA. IEEE Computer Society.

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Yoo, S. (2012). Evolving human competitive spectra-based fault localisation techniques. In Fraser, G. and Teixeira de Souza, J., editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 244–258. Springer Berlin Heidelberg.

Uma Abordagem para a Priorização de Casos de Teste Baseada no Histórico de Detecção de Falhas

Dennis Silva[1], Ricardo Rabelo[1], Pedro Santos Neto[1], Guilherme Lima[1], Ricardo Britto[2], Pedro Almir Oliveira[3]

[1]Universidade Federal do Piauí (UFPI) – Piauí - Brasil

[2]Department of Software Engineering
Blekinge Institute of Technology
Karlskrona, Sweden

[3]Instituto Federal do Maranhão
Pedreiras, Maranhão, Brasil

dennissavio@gmail.com, {ricardoalr, pasn}@ufpi.edu.br,
guilhermefeitosa66@gmail.com, ricardo.britto@bth.se,
pedro.oliveira@ifma.edu.br

Abstract. *The test prioritization consists in determining an execution order for the system test cases, to maximize some property such as the rate of fault detection or the code coverage rate. This work proposes a prioritization approach that considers the history of fault detection and the criticality of the test cases. This criticality value is obtained considering the test case coverage and the criticality of the covered components, obtained through a fuzzy inference system. The prioritization employs ant colony optimization and evaluates the quality of the solutions by its rates of fault detection (APFD), revealing itself statistically and practically superior to the random ordering.*

Resumo. *A priorização de casos de teste consiste em determinar uma ordem de execução dos casos de teste do sistema, para maximizar alguma propriedade como a taxa de detecção de falhas ou de cobertura do código. Este trabalho propõe uma abordagem de priorização que considera o histórico de detecção de falhas e a criticidade dos casos de teste. Essa criticidade é obtida com base na cobertura do caso de teste e na criticidade dos componentes cobertos, obtida por meio de um sistema de inferência fuzzy. A priorização emprega otimização por colônia de formigas e avalia a qualidade das soluções pela taxa de detecção de falhas (APFD), mostrando-se superior à ordenação aleatória em termos estatísticos e práticos.*

1. Introdução

Teste de software é a verificação da adequação de um programa aos requisitos do projeto, consistindo geralmente na execução de um conjunto finito de casos de teste [Bourque and Fairley 2014]. Por mais bem sucedido que seja, o desenvolvimento de um software envolve constante evolução do produto por diversos motivos. Porém, as atualizações realizadas podem afetar de maneira adversa a funcionalidade do produto. Uma maneira de se reduzir esse risco é realizar o teste de regressão, que consiste na

reexecução dos casos de teste do sistema no intuito de verificar se as alterações não inseriram efeitos colaterais inesperados em módulos que estavam funcionando adequadamente em versões anteriores do projeto.

A abordagem mais comum para o teste de regressão é a reexecução de todos os casos de teste do software (*retest-all*) [Yoo and Harman 2010]. Entretanto, conforme o sistema cresce, o seu conjunto de casos de teste também tende a aumentar. Há relatos de sistemas cuja execução do conjunto completo de testes chega a consumir semanas ou até meses [Elbaum et al. 2003]. O custo de realização do teste deve ser considerado, devido aos recursos envolvidos na realização da atividade que podem inviabilizar o reteste completo do sistema. Além disso, independentemente da existência de restrições de custo, uma detecção mais prematura das falhas permite um início mais rápido da correção do sistema.

Nesse cenário, observa-se a necessidade de trabalhos orientados à melhoria da eficiência do teste de regressão, permitindo um melhor aproveitamento do tempo destinado a essa atividade. Dentre as abordagens propostas para a melhoria do custo-eficácia do teste de regressão, pode-se destacar a priorização de casos de teste, que consiste em alterar a sequência de execução dos casos de teste visando maximizar propriedades do conjunto de testes como a taxa de detecção de falhas, a taxa de cobertura, dentre outras.

O objetivo principal desse trabalho é propor uma alternativa para a melhoria da eficácia da realização do teste de regressão, na forma de uma abordagem de priorização de casos de teste que visa maximizar a taxa de detecção de falhas, expressa pela métrica APFD. Para isso, informações disponíveis no teste de regressão, como o histórico de detecção de falhas, além de métricas relacionadas à suscetibilidade a falhas dos componentes do código, foram consideradas.

Este trabalho apresenta uma abordagem em três etapas para a priorização, que considera a criticidade (importância de execução, obtida com base na cobertura e em informações dos componentes cobertos), o histórico de detecção de falhas e o tempo de execução dos casos de teste. Na primeira etapa, é atribuído um valor de criticidade a cada componente do software por meio de um sistema de inferência *fuzzy*. Na segunda etapa é calculada a criticidade dos casos de teste. Na terceira etapa acontece a priorização dos casos de teste, que emprega otimização por colônia de formigas para ordenar os casos de teste considerando suas criticidades, históricos de detecção de falhas e tempos de execução.

A abordagem foi avaliada por meio de um estudo experimental empregando oito programas de um repositório público. A métrica empregada na avaliação das soluções foi o APFD (*Average Percentage of Fault Detection*), que representa a taxa de detecção de falhas. Os resultados obtidos foram submetidos a um *sanity check* e comparados à ordenação aleatória em termos de significância estatística e prática, apresentando fortes indícios de adequação ao problema.

O restante deste trabalho está estruturado da seguinte forma: a Seção 2 apresenta conceitos relacionados ao teste de regressão e sua melhoria. A Seção 3 apresenta os trabalhos relacionados. A Seção 4 detalha a abordagem proposta no trabalho. A metodologia empregada na experimentação é apresentada na Seção 5 e os resultados obtidos são apresentados na Seção 6. Finalmente a conclusão é apresentada na Seção 7.

2. Referencial Teórico

2.1. Teste de Regressão e priorização de casos de teste

Antes do lançamento de uma nova versão de um software, é necessário verificar se as atualizações não afetam o funcionamento de componentes pré-existentes e não modificados. Essa verificação é chamada de teste de regressão [Yoo and Harman 2010]. A abordagem mais simples para o teste de regressão é a execução de todos os casos de teste (*retest-all*). Com o crescimento do sistema e o aumento do número de casos de teste, essa abordagem pode tornar-se muito onerosa ou mesmo inviável. Isso evidencia a necessidade de técnicas para melhorar a execução do teste de regressão, visando obter resultados mais cedo e explorar melhor o potencial dos casos de teste em caso de interrupção prematura do teste.

Os trabalhos relacionados à melhoria do teste de regressão costumam considerar custo (geralmente o tempo de execução dos casos de teste), eficácia (taxa com que o código é coberto ou que as falhas são detectadas), histórico de falhas, a criticidade dos testes para o negócio, dentre outras informações. Dentre as abordagens relacionadas à melhoria do teste de regressão pode ser destacada a priorização de casos de teste, que busca reordenar a execução dos casos de teste para maximizar propriedades como a taxa de detecção de falhas ou a taxa de cobertura do código [Catal and Mishra 2013]. O problema da priorização de casos de teste pode ser definido como:

$Sejam$: T, o conjunto de casos de teste de um sistema; PT, o conjunto de possíveis permutações de T; f, uma função de PT que quantifique as ordenações em números reais.

$Problema$: Encontrar $T' \in PT$, tal que $(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

A priorização não busca remover ou mesmo deixar casos de teste sem execução, mas apenas reordenar o conjunto original de testes. As técnicas de priorização podem reduzir o custo do teste por meio da paralelização de atividades, e possibilitam uma detecção mais prematura das falhas e um *feedback* mais rápido aos testadores.

2.2. Otimização por Colônia de Formigas

A otimização por colônia de formigas (*Ant Colony Optimization*, ou ACO) é um modelo de otimização computacional baseado na capacidade das formigas reais de encontrar o melhor caminho do formigueiro até uma fonte de comida utilizando feromônios [Engelbrecht 2007]. Feromônios são substâncias químicas que as formigas distribuem enquanto caminham e que orientam as companheiras que seguirão o caminho posteriormente. Os caminhos com maior tráfego apresentam maior concentração de feromônio e possuem maior probabilidade de serem escolhidos. Esse comportamento inspira o algoritmo de otimização, no qual as formigas são agentes computacionais cujas rotas representam soluções de um problema.

O algoritmo ACO envolve dois procedimentos básicos: A construção da solução, na qual m formigas constroem em paralelo m soluções para o problema; e a atualização da concentração de feromônios, na qual as soluções construídas pelas formigas tem sua qualidade mensurada por meio de uma função de avaliação. A atualização da concentração de feromônios é baseada na função de avaliação, e melhores soluções recebem mais fe-

romônios. Além dos feromônios, uma função heurística é utilizada para aprimorar a construção de soluções para o problema.

2.3. Sistemas de Inferência Fuzzy

A lógica *fuzzy* é adequada a situações nas quais a informação seja imprecisa ou incerta, e palavras sejam melhores para representar conceitos do que números [Zadeh 1996]. Um sistema de inferência *fuzzy* lida com informação qualitativa, o que facilita o mapeamento da experiência dos especialistas do domínio de interesse, visto que as entradas e saídas do sistema são linguisticamente compreensíveis.

Sistemas de Inferência *Fuzzy* são baseados em regras linguísticas de produção no formato *"Se ... Então..."*. A teoria dos conjuntos *fuzzy* e a lógica *fuzzy* proveem a base matemática para lidar com informação imprecisa, incerta ou qualitativa. Os sistemas de inferência *fuzzy* funcionam em três etapas:

- 1 - Fuzzificação - Mapeamento das entradas numéricas em conjuntos *fuzzy* representados por termos linguísticos, como "muito", "pouco", etc.
- 2 - Inferência - Produção do valor de saída do sistema *fuzzy*, com base nos valores de entrada.
- 3 - Defuzzificação - Associação de um valor numérico à saída da etapa de inferência.

3. Trabalhos Relacionados

Diversos trabalhos propõem técnicas baseadas em priorização de casos de teste para melhorar a realização do teste de regressão. A cobertura dos casos de teste é bastante empregada como critério de priorização de casos de teste, como visto em [Rothermel et al. 1999] e [Elbaum et al. 2000]. Esses trabalhos apresentam uma série de técnicas baseadas no aumento da taxa de cobertura do código como forma de aumentar a taxa de detecção de falhas [Catal and Mishra 2013], partindo do pressuposto que uma sequência de testes que vasculhe o código rapidamente apresenta maior probabilidade de encontrar de forma veloz trechos do código com falhas. Porém, isso não garante melhoria na detecção de falhas [Hao et al. 2016], e é interessante empregar mais de um critério na otimização do teste de regressão [Yoo and Harman 2007].

Uma vez que o teste de regressão envolve a reutilização de testes presentes em estágios anteriores do desenvolvimento, um critério que pode ser utilizado na priorização é o histórico de detecção de falhas. Apesar da ausência de garantias de que casos de teste que já revelaram falhas de fato possuam maior potencial de detecção, esses casos de teste podem ser considerados como tendo 'qualidade comprovada' [Harman 2011]. Qu et al. (2007) apresentam uma técnica para teste caixa-preta de regressão que agrupa casos de teste com base nos tipos de falhas que eles revelam e pode ajustar dinamicamente suas prioridades. Poucas abordagens de priorização consideram o histórico de falhas, e a proposta deste trabalho considera o histórico de detecção ao empregar a métrica APFD na avaliação da qualidade das soluções.

Já houveram iniciativas no sentido de priorizar os testes de acordo com sua importância para o negócio. Khandai et al. (2011) priorizam casos de teste com base na criticidade (contribuição) das funções do programa para o negócio, que é obtida com base

na análise de projetos anteriores. A abordagem proposta neste trabalho trabalha a criticidade com base em métricas extraídas dos componentes do software em desenvolvimento, relacionadas à ocorrência de falhas.

A abordagem proposta em [Silva et al. 2016] é composta de cinco etapas que contemplam a priorização e seleção de casos de teste, que emprega a criticidade dos casos de teste como chave. A criticidade dos casos de teste é baseada na cobertura e em informações dos módulos cobertos, relacionadas à ocorrência de falhas e relevância para o negócio. A abordagem proposta neste trabalho foi construída com base na proposta de [Silva et al. 2016], mas contempla algumas limitações e traz algumas melhorias. O trabalho original continha uma etapa de seleção dos casos de teste, que foi removida pelo fato de a seleção poder suprimir a execução de casos de teste, o que pode aumentar o custo do software devido à não-detecção de certos tipos de falhas [Do et al. 2010]. A construção e a avaliação dos resultados passa a contemplar a taxa de detecção de falhas [Elbaum et al. 2000], bastante empregada em trabalhos relacionados ao teste de regressão, ao invés da ordenação dos casos de teste por criticidade do trabalho original.

4. Abordagem Proposta

Não existe uma maneira ideal de definir a criticidade de execução de um caso de teste. Visto que o teste tem por principal objetivo encontrar falhas, poderia-se definir que os casos de teste mais críticos são aqueles que alcançam esse objetivo. Infelizmente, na prática é impossível saber com antecipação quais casos irão encontrar falhas, mas as abordagens para melhoria do teste de regressão costumam se basear em alguns critérios. A abordagem de priorização proposta neste trabalho (Figura 1) considera a cobertura dos componentes pelos casos de teste, e também a criticidade (importância de realização de teste) desses componentes. A abordagem é composta de três etapas, partindo da inferência da criticidade dos componentes de software, que é utilizada para obter a criticidade dos casos de teste, empregada por sua vez na priorização dos casos de teste.

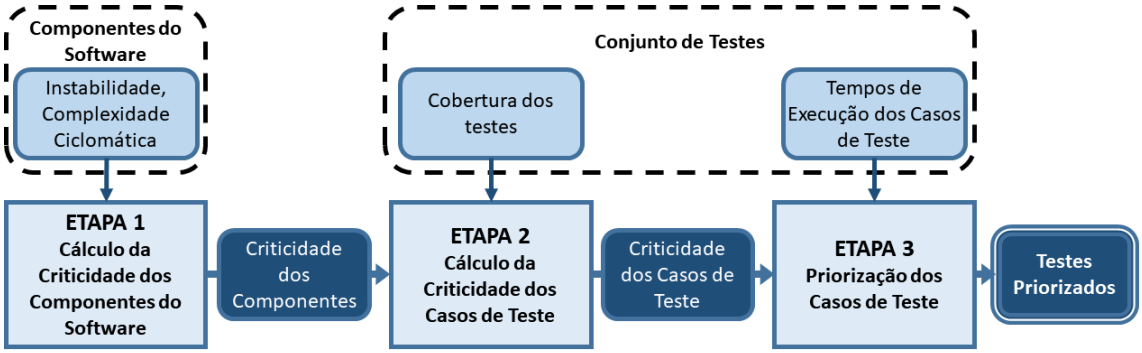


Figura 1. Abordagem Proposta

4.1. Etapa I - Cálculo da criticidade dos componentes de Software

Nessa etapa, a criticidade de cada componente do software é inferida. Uma vez que não existem funções ou métodos específicos para obter esse tipo de valor, um sistema de inferência *fuzzy* foi utilizado para mapear o conhecimento dos especialistas. As entradas desse sistema são métricas extraídas dos componentes do software diretamente ligadas à suscetibilidade a falhas de um componente [Gill and Sikka 2011]. As métricas empregadas foram:

- Complexidade Ciclomática [McCabe 1976]: Essa métrica indica a complexidade do código na forma do número de possíveis caminhos de execução.
- Instabilidade [Martin 2003]: Nível de relacionamento entre os componentes do software. Determinada pela fórmula

$$I = \frac{C_e}{C_e + C_a} \tag{1}$$

onde C_e é o acoplamento eferente, o número de dependências do componente; e C_a é o acoplamento aferente, o número de componentes que dependem do componente.

A variável de saída do sistema de inferência *fuzzy* proposto é a criticidade do componente de código. A base de regras (Tabela 1), entradas e saídas dos conjuntos *fuzzy* foram definidos considerando o nosso conhecimento até então, e validada com especialistas da área de engenharia de software.

Tabela 1. Base de regras do sistema de inferência *fuzzy*

		Instabilidade				
		Muito Baixa	**Baixa**	**Média**	**Alta**	**Muito Alta**
Complexidade	**Muito Baixa**	Muito Baixa	Muito Baixa	Baixa	Baixa	Média
	Baixa	Muito Baixa	Baixa	Baixa	Média	Média
	Média	Baixa	Baixa	Média	Alta	Alta
	Alta	Baixa	Média	Alta	Alta	Muito Alta
	Muito Alta	Média	Média	Alta	Muito Alta	Muito Alta

4.2. Etapa II - Cálculo da criticidade dos casos de teste

A abordagem considera que a execução de um caso de teste é tão crítica quanto os componentes de software por ele cobertos o são. Desta forma, a criticidade *TC* de cada caso de teste *j* é calculada utilizando como entrada a criticidade dos componentes cobertos por ele e sua informação de cobertura, conforme representado pela Equação 2:

$$TC_j = \frac{\sum_{i=1}^{n} FC_i * x_i}{\sum_{i=1}^{n} FC_i} \tag{2}$$

onde i é o índice de uma função coberta pelo teste j; n é o número de funções cobertas pelo teste j; FC_i é a criticidade do componente i; e x_i: cobertura do componente i pelo teste j.

4.3. Etapa III - Priorização dos casos de teste

A abordagem de [Silva et al. 2016] possui uma etapa de seleção que limita o tempo total de execução dos casos de teste. Na abordagem proposta, a etapa de seleção foi suprimida e a etapa de priorização foi modelada como uma adaptação do problema do caixeiro viajante, para o qual as cidades equivalem aos casos de teste e estão dispostas em um grafo completo. Esse problema é bastante propício à aplicação da otimização por colônia de formigas, motivo pelo qual a priorização empregou o *MAX-MIN Ant System* (MMAS) [Stützle and Hoos 2000], um algoritmo que explora o espaço de soluções melhor que o *Ant System*, evitando estagnação precoce.

As entradas empregadas nessa etapa são a criticidade (obtida na Etapa II), o número de falhas detectadas e o tempo de execução dos casos de teste. A heurística empregada no algoritmo foi a relação *criticidade * falhas detectadas / tempo de execução* dos casos de teste. O objetivo da abordagem é maximizar a eficiência do conjunto de testes na detecção de falhas. Para quantificar esta eficiência nas soluções encontradas, foi empregado o APFD (*Average Percentage of Faults Detected*), que representa a taxa de detecção de falhas de um conjunto de testes. O cálculo do APFD é realizado por meio da aplicação da Equação 3:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_i + ... + TF_m}{nm} + \frac{1}{2n} \quad (3)$$

Onde T é a *Suite* de testes; F é o conjunto de Falhas reveladas por T; m é o número de falhas; n é o número de casos de teste; e TF_i é a posição do primeiro caso de teste que detectou a falha i

5. Experimentação

Para a experimentação, Foram utilizados oito programas escritos em C oriundos do SIR (*Software-Artifact Infrastructure Repository*[1]), um repositório público de objetos para experimentação em engenharia de software. A Tabela 2 apresenta as principais características dos objetos.

Tabela 2. Descrição dos objetos do SIR utilizados no trabalho.

Programa	Finalidade	Linhas de código	Número de funções	Tamanho do pool de testes	Tamanho médio das suites de testes
print_tokens	Analisador léxico	402	18	4130	16
print_tokens2	Analisador léxico	483	21	4115	12
replace	Reconhecimento de padrões e substituição	516	22	5542	19
schedule	Agendador de prioridades	299	18	2650	8
schedule2	Agendador de prioridades	297	17	2710	8
space	Prevenção de colisão de aeronaves	6218	136	13585	155
tcas	Interpretador para uma linguagem de definição de arrays	138	9	1608	6
tot_info	programa de estatística	346	9	1052	7

Os programas objeto contém um número de falhas semeadas, permitindo a realização de experimentos envolvendo teste de software. Cada objeto vem com um pool de casos de teste, que envolve tanto testes de caixa-preta e caixa-branca. O conjunto original de testes é utilizado para criar 1000 *suites* de teste para cada programa, visando contemplar a maior cobertura de código da maneira mais realista possível.

Para a experimentação, a abordagem foi implementada na linguagem Java. Em relação à obtenção das entradas da primeira etapa, a complexidade ciclomática dos objetos em C foi calculada usando o programa CCCC (C and C++ Code Counter), e os acoplamentos aferente e eferente foram obtidos pela análise do relatório da árvore de invocações gerado para as funções do programa utilizando o programa *Scitools Understand*. O sistema de inferência *fuzzy* foi implementado utilizando a biblioteca *jFuzzyLogic*.

[1] sir.unl.edu/

Na segunda etapa, para calcular a criticidade dos casos de teste, é necessário conhecer a cobertura dos casos de teste. Essa informação foi obtida utilizando o *gcov*, uma ferramenta empregada junto com o gcc, um compilador da linguagem C. Na terceira etapa, o *Max-Min Ant System* (MMAS) foi empregado na priorização os casos de teste. Para explorar o espaço de busca, foram empregados 20 agentes (formigas) por rodada, em um total de 3000 rodadas. O sistema empregou α = 1.0 e β = 1.0, e uma taxa de persistência de feromônio de 50%.

Para realizar a experimentação, uma *suite* de testes foi selecionada aleatoriamente para cada programa objeto, e a abordagem proposta foi executada 1000 vezes em cada uma. Além disso, uma ordenação aleatória foi executada 1000 vezes para cada *suite*, para realizar um teste de sanidade.

6. Resultados e Discussão

A Tabela 3 apresenta os resultados obtidos para a execução de nossa abordagem e da busca aleatória. Além desses valores, a tabela também contém o APFD da ordenação original da *suite*, da ordenação ótima (nos objetos em que a obtenção foi possível) e de uma ordenação feita com um algoritmo guloso, que construiu as soluções avaliando os casos de teste por sua relação *criticidade * falhas detectadas / tempo de execução*.

Tabela 3. Comparação entre os resultados obtidos com a abordagem proposta, os conjuntos originais (não ordenados) de testes, e a ordenação aleatória.

	Controle			**Experimento**				
	Orig.	**Guloso**	**Ótimo**	**Aleat.**	**Proposta**	**Desvio (Proposta)**	**Dif. Estat.**	**Dif. Prática**
print_tokens	82.50	96.42	-	76.12	96.42	-	<0.001	0.857
print_tokens2	73.07	88.46	96.15	68.17	96.15	-	<0.001	0.923
replace	83.47	83.87	-	49.84	97.36	-	<0.001	0.974
schedule	54.54	63.63	95.45	59.83	95.45	-	<0.001	0.958
schedule2	24.99	35.00	95.00	49.07	95.00	-	<0.001	1
tcas	54.62	71.29	84.25	51.30	84.25	-	<0.001	0.996
totinfo	67.14	67.14	81.42	54.55	81.42	-	<0.001	0.895
space	83.47	83.87	-	83.07	95.62	0.34	<0.001	1

Exceto para o programa *totinfo*, observou-se que em geral a aplicação do algoritmo guloso superou a não-ordenação. Para todos os objetos de experimentação, a média dos resultados da abordagem superou o algoritmo guloso. Em todos os programas nos quais foi possível a verificação, as soluções convergiram para o valor ótimo. A estagnação encontrada para os demais programas indica que as soluções alcançaram um valor ótimo, e o baixo desvio encontrado entre os APFDs encontrados para o programa *space* evidencia que estas soluções situaram-se próximas de um valor ótimo.

A análise dos resultados mostra que os valores de APFD obtidos com a nossa abordagem foram estatisticamente superiores aos da busca aleatória. Foi examinada a significância estatística da diferença entre os resultados. Os dados não seguiam uma distribuição normal. Então, foi utilizado o teste de Wilcoxon-Mann-Whitney, que detectou uma diferença estatisticamente significativa dos resultados da nossa abordagem para os da busca aleatória, a um nível de significância de 0.001.

A melhoria prática da abordagem em comparação à busca aleatória também foi verificada, por meio da realização de uma análise do tamanho do efeito nos resultados. Para isso, a medida A de Vargha-Delaney, uma medida padrão de tamanho do efeito, foi empregada. Os valores para essa medida variam entre 0 e 1, e quanto mais próximo de 0,5 o valor, mais similares são os desempenhos das técnicas. Os valores calculados para a medida A e mostrados na Tabela 3 indicam que para todos os objetos a nossa abordagem funcionou melhor (valores entre 0.5 e 1), e obteve valores de APFD mais elevados do que a busca aleatória também em termos de significância prática.

7. Conclusões e trabalhos futuros

Nesse trabalho, foi proposta uma abordagem baseada em priorização de casos de teste para a melhoria para o teste de regressão que considera a criticidade dos casos de teste. A criticidade dos casos de teste foi obtida a partir da criticidade dos componentes de software, que por sua vez foi inferida por meio do uso de métricas e informações fortemente ligadas à ocorrência de falhas.

Foram realizados experimentos utilizando oito programas de um repositório aberto de artefatos de software. Para aumentar a confiabilidade, verificações de sanidade foram realizadas nos resultados obtidos para cada programa objeto, comparando os valores de APFD obtidos pela abordagem com os de uma busca aleatória. Com base nos resultados, pode-se afirmar que a abordagem proposta é capaz de auxiliar pesquisadores e praticantes a lidar com a priorização do teste de regressão melhor do que o algoritmo de busca aleatória. Em trabalhos futuros, planeja-se melhor avaliar a abordagem proposta neste artigo empregando softwares de grande porte, para checar o comportamento da abordagem frente a situações mais realistas e melhorar a capacidade de generalizar os resultados do estudo.

Referências

aaaa aaaa.

Bourque and Fairley 2014 Bourque, P. and Fairley, R. E., editors (2014). *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, version 3.0 edition.

Catal and Mishra 2013 Catal, C. and Mishra, D. (2013). Test case prioritization: A systematic mapping study. *Software Quality Journal*, 21(3):445–478.

Do et al. 2010 Do, H., Mirarab, S., Tahvildari, L., and Rothermel, G. (2010). The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617.

Elbaum et al. 2003 Elbaum, S., Kallakuri, P., Malishevsky, A., Rothermel, G., and Kanduri, S. (2003). Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification and Reliability*, 13(2):65–83.

Elbaum et al. 2000 Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2000). Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, New York, NY, USA. ACM.

Engelbrecht 2007 Engelbrecht, A. P. (2007). *Computational Intelligence: An Introduction*. Wiley Publishing, 2nd edition.

Gill and Sikka 2011 Gill, N. S. and Sikka, S. (2011). Fault proneness of classes in object-oriented systems. *International Journal of Advances in Embedeed System Research*.

Hao et al. 2016 Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., and Xie, T. (2016). To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, 42(5):490–505.

Harman 2011 Harman, M. (2011). Making the case for morto: Multi objective regression test optimization. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 111–114.

Khandai et al. (2011) Khandai, S., Acharya, A. A., and Mohapatra, D. P. (2011). Prioritizing test cases using business criticality test value. *International Journal of Advanced Computer Science and Applications*, 3(5).

Martin 2003 Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

McCabe 1976 McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Qu et al. (2007) Qu, B., Nie, C., Xu, B., and Zhang, X. (2007). Test case prioritization for black box testing. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 465–474, Washington, DC, USA. IEEE Computer Society.

Rothermel et al. 1999 Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 179–188.

Silva et al. 2016 Silva, D., Oliveira, P. A., Rabelo, R., Campanhã, M., Neto, P. A. d. S. N., and Britto, R. (2016). A hybrid approach for test case prioritization and selection. In *IEEE Congress on Evolutionary Computation (CEC)*.

Stützle and Hoos 2000 Stützle, T. and Hoos, H. H. (2000). Max-min ant system. *Future generation computer systems*, 16(8):889–914.

Yoo and Harman 2007 Yoo, S. and Harman, M. (2007). Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 140–150, New York, NY, USA. ACM.

Yoo and Harman 2010 Yoo, S. and Harman, M. (2010). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120.

Zadeh 1996 Zadeh, L. A. (1996). Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems*, 4(2):103–111.

Ternarius: um operador de mutação para o reparo de software baseado em busca com representação *subpatch*

Vinícius P. L. De Oliveira[1], Eduardo F.D. Souza[1], Altino Dantas[1], Lucas Roque[1], Celso G. Camilo-Junior[1], Jerffeson T. Souza [2]

[1]Universidade Federal de Goiás (UFG)
Instituto de Informática (INF)
Alameda Palmeiras, Quadra D, Câmpus Samambaia. Goiânia – Goiás – Brasil

[2]Universidade Estadual do Ceará
Mestrado Acadêmico em Ciência da Computação (MACC)
Avenida Dr Silas Munguba, 1700, Fortaleza – Ceará – Brasil

Grupo Intelligence for Software (i4Soft)
http://ic1.inf.ufg.br/i4soft
{viniciusdeoliveira,eduardosouza,celso}@ufg.br
{altinoneto,lucasroque}@ufg.br
prof.jerff@gmail.com

Abstract. *Software maintenance is a costly and complex task that has been addressed by various approaches which try to automatically repair buggy programs. In this context, GenProg is a search-based tool that has promising results, however, its original patch representation and genetic operators are still in evolution. Thus, this work proposes a novel mutation operator, based on sub patch representation, capable of performing three types of perturbation in the solutions. The proposal was evaluated with the IntroClass benchmark. The results indicate the approach viability by achieving around 30% more fix than the original mutation and getting 94.69% of qualified fixes.*

Resumo. *A manutenção de software é uma tarefa complexa e custosa que tem sido abordada por várias pesquisas que tentam reparar automaticamente programas defeituosos. Nesse contexto, a GenProg é uma ferramenta baseada em busca que tem resultados promissores, no entanto, sua representação original de patch e seus operadores genéticos continuam sendo evoluídos. Assim, este trabalho propõe um novo operador de mutação, baseado em representação de subpatch, capaz de realizar três tipos de perturbação nas soluções. A proposta foi avaliada com o benchmark IntroClass. Os resultados indicaram a viabilidade da abordagem, que conseguiu cerca de 30% mais de reparos do que a mutação original e obtendo 94,69% de correções com qualidade.*

1. Introdução

A humanidade está cada vez mais habituada à utilização de softwares, em tarefas simples ou complexas. Todavia, nem sempre as aplicações se comportam da forma desejada, e com a crescente utilização das mesmas, as falhas ou *bugs*, representam uma problemática significante. Nesse contexto, a tarefa de manter um software funcionando corretamente é tipicamente complexa e cíclica, envolvendo a busca e recuperação de falhas ou implementação de novas funcionalidades.

A manutenção de software se tornou um processo oneroso. Evidências indicam que a quantidade de *bugs* detectados diariamente é maior que a capacidade dos profissionais em resolvê-los [Le Goues et al. 2012a], atrelado ao fato de que esta fase corresponde a 70% do custo financeiro do ciclo de vida de um software [Pressman 2005].

Neste cenário, o Reparo Automatizado de Software (RAS) surgiu como uma tentativa de diminuir esse desequilíbrio entre a quantidade de *bugs* identificados e a capacidade de solucioná-los, além dos custos relacionados a eles, buscando transformar um comportamento inaceitável de uma execução de um software, em um comportamento aceitável de acordo com uma especificação [Monperrus 2015].

Diversas técnicas de reparo automatizado têm sido propostas. DirectFix [Mechtaev et al. 2015] e Angelix [Mechtaev et al. 2016] são exemplos de metodologias baseadas em semântica, que buscam reparar sistemas através de execução simbólica. Prophet [Long and Rinard 2016] e DeepFix [Gupta et al. 2017], por sua vez, utilizam modelos de aprendizado para determinar a possibilidade de certa alteração reparar o software defeituoso. Finalmente, com técnicas baseadas em busca, podemos citar o SPR [Martinez et al. 2014] e a GenProg [Le Goues et al. 2012b].

A GenProg é uma ferramenta que utiliza programação genética para reparar automaticamente softwares defeituosos. A técnica utiliza *patch*, uma sequência de alterações ou operações aplicáveis ao código original, como representação de soluções, isto é, como variantes do código defeituoso. Apesar de apresentar resultados promissores [Le Goues et al. 2012a], alguns de seus aspectos ainda requerem evolução. Por exemplo, visando melhorar a qualidade da busca da GenProg, [Oliveira et al. 2016] propuseram uma representação por *subpatch* - aumentando a granularidade da representação e, assim, permitindo uma melhor troca de material. A nova representação melhorou significativamente o desempenho da ferramenta, quando aplicados os novos e específicos operadores de cruzamento. Contudo, os autores não exploraram novos operadores de mutação.

Assim, o presente trabalho propõe um novo operador de mutação denominado *"Ternarius"* objetivando explorar os benefícios da alta granularidade da representação *subpatch* e aumentar os tipos de pertubações possíveis nas soluções. Sabendo que a atual operação de mutação da GenProg apenas adiciona novas operações aos indivíduos, a principal hipótese desse estudo é que o aumento dos tipos de mutação (mutação com perturbações mais granulares e retirada de operações do indivíduo) melhore a exploração do espaço de busca e, consequentemente, aumente a eficácia da ferramenta.

Para validar a proposta, os experimentos buscam responder as seguintes perguntas de pesquisa:

RQ 1 *O operador Ternarius aplicado à GenProg consegue produzir mais reparos em comparação ao operador canônico?*

RQ 2 *Qual a qualidade dos reparos produzidos pelo operador de mutação Ternarius?*

Com isso, as principais contribuições são:

- Um novo operador de mutação para o reparo de software baseado em busca com representação *subpatch*;
- Uma avaliação de eficácia da proposta e qualidade dos *patches* gerados.

O restante deste trabalho é organizado da seguinte forma. A seção 2 apresenta os princípios da GenProg; Seção 3 descreve o novo operador de mutação proposto; Seção 4 descreve o *design* dos experimentos, além dos resultados e análises; E finalmente, a seção 5 apresenta as considerações finais.

2. Background

2.1. GenProg

A GenProg é uma ferramenta baseada em busca que utiliza programação genética para encontrar uma sequência de operações, chamado *patch*, que quando aplicadas ao software defeituoso, repara-o de acordo com a especificação definida. A ferramenta utiliza um conjunto de casos de teste como especificação para determinar se o sistema foi reparado ou não [Le Goues et al. 2012b].

Para buscar por um reparo, a GenProg recebe um programa e um conjunto de casos de teste com pelo menos um teste negativo, que indica a presença de *bug*. A fim de facilitar a correção da falha, a ferramenta estima onde possivelmente está o erro, com o auxílio dos casos de teste. Posteriormente, são gerados variantes do programa original que são representadas por um *patch*, contendo uma sequencia de edições que será realizada visando a correção do *bug*.

Cada edição é composta de três subespaços, *operation*, *fault* e *fix*, representadas da seguinte forma: *operation(fault,fix)*. *Operation* representa as modificações possíveis dentro de um *patch*, são elas *append* (inserção de um novo *statement*), *swap* (troca de um *statement*) e *delete* (exclusão de um *statement*). *Fault* define o possível ponto de falha, onde a operação será realizada, enquanto *fix* representa o código que será utilizado para modificar o ponto da falha [Le Goues et al. 2012b].

Inicialmente os *patches* são gerados de maneira aleatória, e cada um deles tem seu potencial de correção calculado através dos casos de teste. O objetivo é encontrar um *patch* que ao ser aplicado no código original, gere uma variante capaz de fazer passar todos os casos de teste, chamada de variante plausível. A busca tem continuidade por meio da evolução dos *patches* gerados inicialmente, aplicando sobre eles operadores genéticos de cruzamento e mutação [Le Goues et al. 2012b].

O operador de mutação original da GenProg usa uma probabilidade uniformemente distribuída para realizar uma das três ações: 1) excluir uma operação, 2) inserir operação nova ou 3) substituir uma operação existente. A composição de cada nova operação é realizada a partir de *statements* do programa original.

O operador de cruzamento combina edições dos patches (indivíduos). Por exemplo, define-se um ponto de corte para cada *patch* e, posteriormente, combina-se as partes para gerar novos *patches*. Como cada edição é indivisível na representação *patch*, as

edições possíveis e consideradas na combinação são as geradas aleatoriamente no inicio da busca ou pela mutação [Oliveira et al. 2016].

Além dos operadores, outro aspecto que pode ser melhorado na GenProg é a forma como os *patches* são representados, dada a limitação de combinação dos indivíduos e consequentemente a qualidade da busca.

Visando melhorar a eficiência da ferramenta foi proposta uma nova forma de representação por *patch*, chamada de *subpatch*, para aumentar a granularidade da busca. Nesta, as edições podem ser divisíveis, sendo possível a combinação de subespaços das edições existentes. Essa nova abordagem torna possível que o cruzamento gere operações totalmente novas, o que aumenta as combinações de material genético, consequentemente aumentando a diversidade da população. Ressalta-se que, apesar de aumentar a diversidade, os indivíduos gerados possuem distâncias aceitáveis para o papel de *exploitation* do operador [Oliveira et al. 2016]. A Figura 1 apresenta as duas formas de representação.

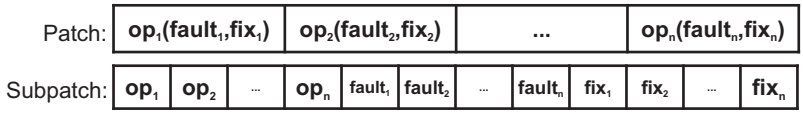


Figura 1. Representação por patch e por subpatch.

2.2. Operador de cruzamento *Unif1Space*

Neste trabalho será utilizado o operador de cruzamento *Unif1Space* com memorização, que apresentou melhor desempenho em trabalhos anteriores quando aplicado à GenProg. Dessa forma, focar-se-á na influência do operador de mutação.

O *Unif1Space* é a aplicação do operador uniforme clássico a um subespaço de uma edição do patch [Oliveira et al. 2016]. Este operador favorece a exploração, pois possui uma capacidade de combinação de material genético superior a um operador de um ponto de corte, por exemplo. Além disso, privilegia operações que necessitam de mudança em apenas um dos subespaços de uma operação do *patch*.

Ainda, o *Unif1Space* pode utilizar um processo de memorização para mitigar as perdas por mapeamento entre representações [Oliveira et al. 2016]. Tal processo consiste em armazenar partes de edições, que foram inutilizadas durante o processo de cruzamento, e reutilizá-las em indivíduos futuros, evitando-se assim perda de material genético. Essa memória genética não é compartilhada entre toda a população, mas sim entre os indivíduos de uma mesma linhagem genética, ou seja, um indivíduo só pode utilizar memória genética proveniente de um antepassado dele. Isso reduz a inserção de valores aleatórios no indivíduo [Oliveira et al. 2016].

3. Operador de mutação *Ternarius*

A granularidade da representação tem grande influência no resultado da operação de mutação. Observa-se que a granularidade da representação original da GenProg impede a alteração de elementos da edição que possuem baixo impacto na busca. A Figura 2 apresenta um exemplo em que se tem um *patch* eleito para a mutação e um *patch* desejado. Observa-se que bastaria a mudança do Fault_1 da primeira operação do *patch* existente para Fault_2 que a operação do *patch* desejado seria obtida. Por isso, uma mutação mais

granulada, no nível de subespaço, poderia, através da mudança em subespaços, reativar operações inativas e aumentar as chances de criação de novas operações.

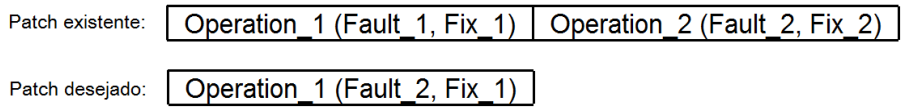


Figura 2. Limitações da representação de *patch* com operações indivisíveis.

O operador de mutação tem grande impacto e relevância na busca da GenProg. Conforme dito anteriormente, este operador é responsável pela inserção de uma edição totalmente nova no *patch*, resultando em mais material para a busca. Apesar da importância do aumento dos indivíduos, estudos demonstram que *patches* gerados por seres humanos são, comumente, curtos [Martinez and Monperrus 2015].

Além disso o crescimento acentuado dos *patches* pode causar problemas para a busca, pois *patches* muito grandes tendem a mudar completamente um código, que eventualmente seria reparado com uma alteração pontual. Este fato também impacta no processo de busca, pois quanto maior o individuo menor é a influência exercida pela mutação. Sendo assim, a capacidade de redução do tamanho do indivíduo pode ser benéfica para a busca. A Figura 2 demonstra também que a deleção da segunda operação do *patch* existente aproximaria ele do *patch* desejado.

A fim de explorar melhor as características da representação de maior granularidade e incluir uma estratégia de controle de crescimento dos indivíduos, foi proposto um novo operador de mutação, intitulado de ”*Ternarius*”. Este operador é capaz de modificar uma variante de até três formas:

- Deletando operações existentes selecionadas aleatoriamente;
- Alterando um valor de um subespaço, selecionado aleatoriamente, de uma operação aleatoriamente escolhida;
- Acrescentando novas operações de edições a um indivíduo.

Cada uma dessas alterações possuem uma probabilidade de serem aplicadas ao indivíduo, conforme a ordem descrita acima. Esta ordem é necessária para evitar que uma modificação anule o efeito de outra que ocorreu anteriormente. A figura 3 apresenta um exemplo do novo operador de mutação, demonstrando as três etapas.

A Figura 3 apresenta um exemplo onde o novo operador de mutação é aplicado em um *patch* com três operações. Inicialmente, o *patch* é submetido à etapa de deleção em que é escolhido uma operação aleatória do *patch* e é feita a deleção desta operação, neste caso foi escolhido a terceira operação “r(5,6)”. O resultado da deleção é submetido à etapa de mudança de valores de subespaço, em que uma operação é escolhida aleatoriamente e um subespaço desta operação é selecionado aleatoriamente, nesta caso o subespaço *operator* foi escolhido, é então sorteado um valor dentro do subespaço escolhido (*operator*), sendo assim o valor selecionado “d” (*delete*) é substituído por “a” (*append*). Finalmente, é feito a mutação gerando assim uma nova operação de edição que será incluída no fim do *patch*, neste caso a operação resultante da mutação foi “s(1,3)”.

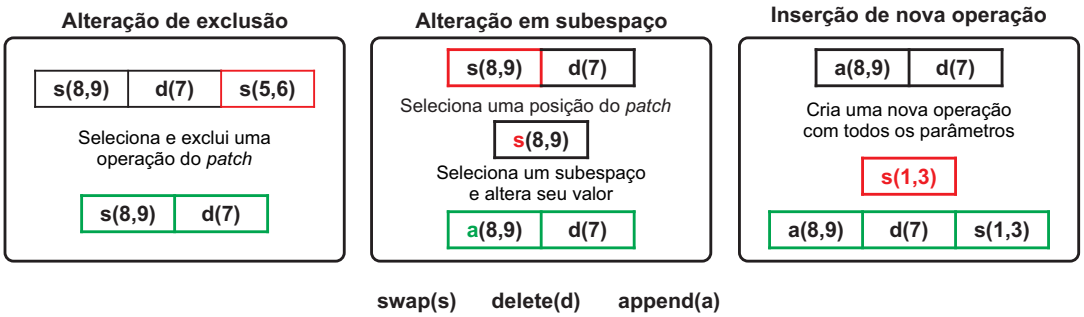


Figura 3. Exemplo da aplicação do operador *Ternarius*. Destaques em vermelho indicam itens selecionados ou criados, e em verde indicam os resultados em cada fase.

4. Avaliação da proposta

Para responder as questões de pesquisa, o operador de mutação *Ternarius* foi implementado na ferramenta GenProg. Assim, ambos os operadores de mutação foram avaliados a partir do *Benchmark* denominado *IntroClass* [Goues et al. 2015]. Este *dataset* publicado online e que vem sendo utilizado em pesquisas relacionadas a reparo automatizado de software, é composto por versões de programas construídos por estudantes de programação. Nele podem ser encontrados programas corretos, versões defeituosas e casos de teste capazes de relevar os defeitos para cada programa. A Tabela 1 apresenta informações do *IntroClass* que foram utilizadas neste experimento. A coluna "Reparo" indica a quantidade de casos de teste que foram utilizados durante o processo de geração de soluções para cada programa, enquanto a coluna "Validação" apresenta a quantidade de casos de teste que foram utilizados para avaliar a qualidade dos reparos produzidos.

Tabela 1. Quantidade de programas, versões e casos de teste do *Benckmark* utilizado (*IntroClass*) para avaliação do operador de mutação proposto.

		Casos de teste	
Programa	**Versões**	**Reparo**	**Validação**
checksum	8	6	10
digits	18	6	10
median	14	7	6
smallest	14	7	8
syllables	14	6	10

Tendo os dados provenientes do *benckmark*, a ferramenta GenProg foi executada 30 vezes para cada uma das versões dos programas. Os parâmetros utilizados foram: 30 gerações, população de 30 indivíduos, elitismo de três indivíduos, taxa de cruzamento de 50%, taxa de mutação de 100% e seleção por torneio binário [Goues et al. 2015]. A alta taxa de mutação comumente utilizada é justificada pelas características e papel já relatadas do operador de mutação na GenProg. O critério de parada foi alcançar 30 gerações ou encontrar um reparo para o programa. Tanto no operador de mutação canônico, quanto na implementação com o novo operador de mutação, o tipo de cruzamento utilizado foi o Unif1Space com memorização, proposto em [Oliveira et al. 2016].

Conforme indicado na seção 3, o operador de mutação *Ternarius*, diferente do

canônico, pode aplicar três ações diferentes durante a perturbação. Por isso, foram avaliadas 5 variações deste operador, sendo que cada variação corresponde a combinações das probabilidades de ocorrência de cada operação. A convenção "delete_insert_update" indica as taxas de: a) deleção de uma operação do *patch*, b) inserção de uma nova operação no *patch* e c) manipulação de material genético dentro de um subespaço, respectivamente. É importante pontuar que não foram apresentadas variações da probabilidade de ocorrer operação de inserção na mutação porque, em testes preliminares, foi observado que alterações neste parâmetro sempre produzia resultados substancialmente inferiores.

4.1. Resultados e Análises

Inicialmente serão apresentados os dados relativos à eficácia de cada uma das estratégias de mutação. Para esta análise, foram condensados todos os resultados de todos os programas por cada uma das variações de operador de mutação. A Tabela 2 apresenta o percentual médio de reparos obtidos em relação ao total de defeitos existentes em todos os programas, considerando as 30 execuções para cada versão e operador de mutação.

Tabela 2. Percentual médio de reparos encontrados entre todos os programas, para cada variação da mutação proposta e da mutação canônica, considerando 30 execuções. ▲ indica superioridade e ▼ inferioridade em relação ao resultado da mutação canônica com o respectivo cruzamento.

	ID	delete_insert_update (%)	Reparos
Mutação canônica	MC	0_100_0	27,962%
Ternarius	T1	0_100_100	27,654% ▼
	T2	0_100_10	**36,358%** ▲
	T3	10_100_0	29,382% ▲
	T4	80_100_0	34,876% ▲
	T5	80_100_10	26,600% ▼

Como pode ser visto, na segunda linha constata-se que 27,9% de todos os defeitos foram reparados quando se utilizou o operador de mutação canônico. Da linha 3 à linha 7, são apresentados os resultados das cinco diferentes configurações do operador *Ternarius*. É possível observar que as variações T1 e T5 obtiveram resultado pior do que a mutação canônica, embora a diferença não chegue a 0,5%. No caso do T1, o fato de haver 100% de chance de mutação em um dos subespaços pode estar causando mais perturbações do que o razoável para manter material genético de qualidade, impedido assim um melhor desempenho para esta configuração. Já a T5 apresenta uma probabilidade de 10% de realizar alterações em subespaços, porém, com 80% de chance de deletar operações do *patch*. O desempenho desta combinação pode não ter sido melhor pela diminuição de material genético imposto pelo um alto percentual de deleção em combinação com as perturbações nos subespaços, pois dessa forma a mutação em tais subespaços passa a ter um grande impacto na composição dos indivíduos. Nesse sentido, o resultado de 34,8% obtido pela T4, o segundo melhor, deve ser destacado uma vez que também possui alta taxa de deleção de operações. Entretanto, neste caso não há chance de mudança nos subespaços de modo que, tal resultado é obtido graças ao operador de cruzamento utilizado, o qual permite a troca de material entre subespaços.

Finalmente, a configuração T2 foi a que obteve o melhor resultado, chegando a reparar em média 36,3% do defeitos no *benchmark*. Como se vê, esta configuração

(0_100_10) não realiza deleção de operações, sempre insere novas operações e realiza poucas perturbações em subespaços; Dessa forma há sempre um boa quantidade de material genético disponível para a combinação e a atuação pontual nos subespaços promove a melhoria de eventuais materiais de baixa qualidade.

Para uma observação mais detalhada da configuração *Ternarius* que obteve os melhores resultados considerando todos os programas, a Tabela 3 apresenta os percentuais médios de reparos alcançados pela configuração T2 em cada um dos cindo programas.

Tabela 3. Percentual médio de reparos encontrados para cada um dos programas considerando 30 execuções da mutação canônica e da proposta T2. ▲, ▼ e – indicam respectivamente que a proposta foi superior, inferior ou igual em relação à mutação canônica com o mesmo cruzamento aplicadas ao mesmo programa.

Programas	Mutação canônica	T2
checksum	12,92%	17,92% ▲
digits	5,56%	5,56% –
median	32,38%	46,19% ▲
smallest	49,76%	70,95% ▲
syllables	5,71%	5,71% –

Como pode ser visto, em três dos cinco programas, a configuração T2 conseguiu produzir, em média, mais reparos do que a mutação canônica da ferramenta GenProg. Observa-se que no caso do programa "checksum" saiu-se de 12,92% para 17,92%, um ganho real de 38,7% ao se utilizar o *Ternarius* configurado como 0_100_10. Para os casos dos programas "median" e "samllest" o ganho foi ainda maior, chegando a 42,6% em ambos. Entretanto, nota-se que para os programas "digits" e "syllables", mesmo sendo a melhor configuração, a mutação T2 não foi suficiente para melhorar os resultados obtidos pela a mutação original. Não obstante, deve ser observado que para tais programas o percentual de reparos encontrados é consideravelmente pequeno (5,56% e 5,71%) indicando que os defeitos neles contidos podem apresentar alguma característica particularmente complexa para a GenProg como um todo, não sendo contornável apenas com mudança no mecanismo de mutação. Como o *benchmark IntroClasse* não fornece detalhes sobre os tipo de defeito em cada programa, seria necessária uma inspeção manual em cada uma das versões para se obter eventuais *insights* sobre como contornar esse problema, porém, esta tarefa foi deixado para evolução do presente estudo.

Diante do que fora exposto, pode-se concluir que, no cenário estudado, o operador de mutação *Ternarius* conseguiu apresentar, em média, mais reparos do que o operador de mutação atualmente presente na ferramenta GenProg considerando todos programas juntos, e, na pior das hipóteses, obteve os mesmos resultados quando observado cada um dos programas individualmente. Respondendo-se assim a **RQ1**.

Visando analisar a qualidade dos reparos produzidos com cada um dos operados de mutação, todos os reparos encontrados foram validados utilizando-se os casos de teste de validação disponíveis no *benchmark*. Na Tabela 4, verifica-se a acurácia dos operadores de mutação através da medida da relação entre a quantidade de reparos válidos e o total de reparos obtidos pela ferramenta. A tabela reporta ainda os quantitativos de casos de teste de validação que foram executados, quantos passaram (positivos) e quantos falharam

(negativos). A linha MC é relativa ao operado canônico e as linhas abaixo desta trazem os dados para as variações do *Subsapce Mutation*.

Tabela 4. Dados de validação dos reparos obtidos pelo operador de mutação canônico e pelas variações do Ternarius.

	Quantidade de casos de teste			Quantidade de Reparos			
Mutação	Executados	Positivos	Negativos	Encontrados	Falsos	Válidos	Acurácia
MC	3534	3498	36	453	13	440	**97,13%**
T1	3508	3420	88	448	29	419	93,53%
T2	4518	4404	114	589	45	**544**	92,36%
T3	3686	3587	99	476	35	**441**	92,65%
T4	4358	4280	78	565	30	**535**	**94,69%**
T5	3372	3296	76	430	24	406	94,42%

Observando os valores de acurácia, constata-se que o operador canônico obteve o valor mais elevado, 97,13%, uma vez que dos 453 reparos encontrados 440 foram validados. Isso significa que é alta a chance de um reparo produzido por tal operador ser de fato um reparo válido e não um falso positivo. Nesse aspecto, a variação do *Ternarius* que mais se aproximou do MC foi a T4 com 535 reparos válidos de 565 reportados, performando, assim, 94,64% de acurácia. A configuração T2, que encontrou a maior quantidade de reparos (589), apresentou 92,36% sendo o pior valor nesta métrica.

As observações anteriores indicam um *trade-off* entre a eficácia em encontrar reparos e a garantia do quão válidos são os mesmos. Por esse raciocínio, ainda que se escolham as configurações T2, T3 e T4 ter-se-ão mais reparos válidos do que a quantidade de reparos válidos provenientes da mutação canônica. Portanto, em resposta à **RQ2** pode-se afirmar que dentre as melhores configurações do operador *Ternarius*, 94,69% é o indicativo de qualidade dos reparos obtidos pela configuração T4.

5. Considerações Finais

A manutenção de sistemas se tornou um processo ainda mais dispendioso com a popularização dos softwares. Por esse motivo, várias propostas de Reparo Automatizado de Software têm sido desenvolvidas, porém ainda há espaço para melhorias.

Este trabalho propôs um novo operador de mutação para a GenProg, uma consolidada ferramenta de reparo automatizado. O operador nomeado *Ternarius*, além de inserir novas operações aos indivíduos (*patches*), como é feito na GenProg tradicional, pode também excluí-las ou realizar perturbações nos subespaços das operações.

Os experimentos indicaram que o princípio do operador proposto é promissor ao aumentar o percentual médio de reparos de 27,9% para 36% (cerca de 30% de aumento) considerando todos os defeitos do *benchmark* utilizado. Além disso, constatou-se um *trade-off* entre encontrar soluções para os defeitos e a taxa de validade destas.

Como evolução desta pesquisa, pretende-se verificar se eventuais características do *dataset* utilizado não privilegiam alguma variação do operador proposto, ampliar o conjunto de programas avaliados para outros *benchmarks* e verificar o comportamento do operador de mutação *Ternarius* em combinação com outros operadores de cruzamento e fora do contexto da GenProg.

Referências

Goues, C. L., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., and Weimer, W. (2015). The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012a). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012b). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM.

Martinez, M. and Monperrus, M. (2015). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205.

Martinez, M., Weimer, W., and Monperrus, M. (2014). Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE Press.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 691–701. IEEE.

Monperrus, M. (2015). Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille.

Oliveira, V. P. L., Souza, E. F. D., Le Goues, C., and Camilo-Junior, C. G. (2016). Improved crossover operators for genetic programming for program repair. In Sarro, F. and Deb, K., editors, *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 112–127, Cham. Springer International Publishing.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

Lista de Autores | *Authors*

A

Alencar, Thayse 11
Araújo, Allysson 21

B

Basniak, Rodrigo 1
Bastos, Laudelino 1
Britto, Ricardo 41

C

Camilo-Junior, Celso 31, 51
Campos, Gustavo 11

D

Dantas, Altino 31, 51

E

Emer, Maria Cláudia 1

F

Freitas, Diogo 31

H

Harrison, Rachel 31

L

Leitão-Júnior, Plínio 31
Lima, Dayvison 11
Lima, Guilherme 41

N

Neto, Adolfo 1
Neto, Pedro 41

O

Oliveira, Pedro 41
Oliveira, Vinícius 51

R

Rabelo, Ricardo 41
Roque, Lucas 51

S

Saraiva, Raphael 11, 21
Silva, Dennis 41
Souza, Eduardo 51
Souza, Jerffeson .11, 21, 51

Y

Yeltsin, Italo 21