IX BRAZILIAN CONFERENCE ON SOFTWARE

CBSOFT

SÃO CARLOS 2018

PROCEEDINGS

IX Workshop de Engenharia de Software Baseada em Busca

IX Workshop on Search Based Software Engineering

ORGANIZATION:

ICMC USP SÃO CARLOS

ufscar

INSTITUTO FEDERAL São Paulo Câmpus São Carlos

PROMOTION:

SBC Sociedade Brasileira de Computação

SUPPORT:

CAPES

FAPESP

Prefeitura de SÃO CARLOS

Anais – *Proceedings*

IX Workshop de Engenharia de Software Baseada em Busca

IX Workshop on Search Based Software Engineering

IX Congresso Brasileiro de Software: Teoria E Prática (CBSoft 2018)
IX Brazilian Conference on Software: Theory and Practice (CBSoft 2018)

Dados Internacionais de Catalogação na Publicação
Sociedade Brasileira de Computação
Universidade de São Paulo, São Carlos

Workshop de engenharia de software baseada em busca (9. : 2018 : São Carlos).

Anais do 9ª Workshop de engenharia de software baseada em busca [recurso eletrônico] / 9º Workshop de engenharia de software baseada em busca, 19 de setembro, 2018; editado por Thiago Gottardi. – São Carlos: Universidade de São Paulo, 2018. 62 p.

Tema central: Software: Teoria e Prática.
Disponível em: http://wesb2018.dainf.ct.utfpr.edu.br/index-en.html;.
ISBN: 978-85-7669-448-9

IX Workshop de Engenharia de Software Baseada em Busca

19 de setembro, 2018
São Carlos, SP, Brazil

ANAIS | *PROCEEDINGS* Sociedade Brasileira de Computação (SBC)

COORDENADORES DO PROGRAMA | *PROGRAM COMMITTEE CHAIRS*

Maria Cláudia Figueiredo Pereira Emer (DAINF – UTFPR)
Wesley Klewerton Guez Assunção (COTSI – UTFPR)

COORDENADORES GERAIS | *GENERAL CHAIRS*

Elisa Yumi Nakagawa (General Chair ICMC/USP)
Rosana Vaccare Braga (General Co-Chair ICMC/USP)
Valter Vieira de Camargo (General Co-Chair DC/UFSCar)
Auri Marcelo Rizzo Vincenzi (General Co-Chair DC/UFSCar)
Daniel Lucrédio (General Co-Chair DC/UFSCar)
Lucas Bueno Ruas de Oliveira (General Co-Chair IFSP/São Carlos)

COMITÊ GESTOR DO CBSOFT 2018 | *CBSOFT 2018 STEERING COMMITTEE*

Fernando Trinta (UFC) (President and General Chair CBSoft 2017)
Marum Simão Filho (UNI7) (General Co-Chair CBSoft 2017)
Rossana Andrade (UFC) (General Co-Chair CBSoft 2017)
Elisa Yumi Nakagawa (ICMC/USP) (General Chair CBSoft 2018)
Rosana Braga (ICMC/USP) (General Co-Chair CBSoft 2018)
Auri Vincenzi (DC/UFSCar) (General Co-Chair CBSoft 2018)
Valter Vieira de Camargo (DC/UFSCar) (General Co-Chair CBSoft 2018)
Daniel Lucrédio (DC/UFSCar) (General Co-Chair CBSoft 2018)
Lucas Oliveira (IFSP) (General Co-Chair CBSoft 2018)
José Carlos Maldonado (ICMC/USP) (Coordinator CEES and Program Chair SBES 2017)
Fabiano Cutigi Ferrari (DC/UFSCar) (Coordinator CEES Coordinator SBES 2017)
Fabio Mascarenhas (UFRJ) (Coordinator CELP Coordinator SBLP 2017)
Uirá Kulesza (UFRN) (Program Chair SBES 2018)
Carlos Camarão (UFMG) (Program Chair SBLP 2018)
Márcio Ribeiro (UFAL) (Program Chair SBCARS 2017)
Ingrid Nunes (UFRGS) (Program Chair SBCARS 2018)
Arilo Dias Neto (UFAM) (Program Chair SAST 2017)
Patricia Machado (UFCG) (Program Chair SAST 2017)
Guilherme Horta Travassos (COPPE/UFRJ) (Program Chair SAST 2018)
Avelino Francisco Zorzo (PUC-RS) (Program Chair SAST 2018)

COORDENADOR DOS ANAIS | *PROCEEDINGS CHAIR*

Rosana Teresinha Vaccare Braga (ICMC/USP)

EDITORAÇÃO DO LIVRO DE ANAIS | *PROCEEDINGS BOOK EDITOR*

Thiago Gottardi (ICMC/USP)

ARTE DA CAPA | *COVER ART*

Jonathan Silva (IFSP/São Carlos)

DESENVOLVIMENTO WEB | *WEB DEVELOPMENT*
Thiago Luiz Varella (IFSP/São Carlos)

GERÊNCIA DE COMUNIDADES E WEB | *WEB AND COMMUNITY MANAGERS*
Lina Garcés (ICMC/USP)
Tiago Volpato (ICMC/USP)
Ana Allian (ICMC/USP)
Warteruzannan Soyer Cunha (DC/UFSCar)
Jonathan da Silva (IFSP/São Carlos)
Thiago Luiz Varella (IFSP/São Carlos)

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

ORGANIZAÇÃO | *ORGANIZATION*
Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (ICMC/USP)
Universidade Federal de São Carlos (UFSCar)
Instituto Federal de São Paulo (IFSP-São Carlos)

APOIO | *SUPPORT*
Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)
Faber-Castell
Prefeitura de São Carlos

PATROCINADORES | *SPONSORS*
UOL/PagSeguro, B2W Digital
Serasa Experian, Monitora, IBM, Google
S2.it, CeMEAI

Prefácio

O IX Workshop de Engenharia de Software Baseada em Busca (WESB) visa consolidar e ampliar o estudo e aplicação de conceitos, técnicas, algoritmos e estratégias de busca na disciplina de Engenharia de Software. O WESB reúne anualmente a comunidade nacional que desenvolve pesquisas envolvendo otimização e meta-heurísticas para solucionar problemas da Engenharia de Software. Assim sendo, damos as boas-vindas a todos os participante e leitores dos anais do Workshop.

Os trabalhos submetidos para a IX edição do WESB foram cuidadosamente revisados por pelo menos três avaliadores do comitê de programa, que contou com pesquisadores de diferentes especialidades e de várias regiões do país. Estes anais contêm seis trabalhos completos apresentados em duas seções técnicas. Nestes trabalhos são abordados temas, como: requisitos de software, teste de software, refatoração, dentre outros. Este fórum de discussão e compartilhamento de experiências também conta com uma palestra convidada e com uma reunião de grupos de trabalho com o intuito de promover colaborações futuras entre pesquisadores que trabalham com temas que envolvam otimização e meta-heurísticas.

Agradecemos aos organizadores do CBSoft 2018 pela oportunidade de realizarmos mais uma edição do WESB. Também agradecemos aos membros do comitê técnico de programa pela avaliação criteriosa dos artigos e pelo trabalho realizado dentro dos prazos estabelecidos. Por fim, parabenizamos os autores dos trabalhos selecionados e agradecemos a todos que submeteram trabalhos para o evento.

Esperamos que o WESB 2018 contribua para a formação e consolidação de parcerias nesta área, bem como para fortalecer a área de Engenharia de Software Baseada em Busca no Brasil. Desejamos a todos um excelente evento!

Maria Cláudia Figueiredo Pereira Emer (DAINF – UTFPR)
Wesley Klewerton Guez Assunção (COTSI – UTFPR)
Organizadores do Programa - WESB 2018

Foreword

The IX Brazilian Workshop on Search-Based Software Engineering (WESB) aims to consolidate and expand the study and application of concepts, techniques, algorithms and search strategies in the discipline of Software Engineering. WESB annually brings together the Brazilian community that develops research involving optimization and meta-heuristics to solve software engineering problems. Therefore, we welcome all participants and readers of the workshop proceedings.

The pieces of work submitted to the IX edition of WESB were carefully reviewed by at least three referees of the program committee, which was composed of researchers with different expertises and from various regions of the country. These proceedings contain six complete papers presented in two technical sections. In these papers, the following topics are discussed: software requirements, software testing, refactoring, among others. This forum for discussion and sharing of experiences also has an invited talk and a working group meeting with the intention of promoting future collaborations between researchers working on similar themes involving optimization and metaheuristics.

We would like to thank the organizers of CBSoft 2018 for the opportunity of occurring one more edition of WESB. We are also grateful to the members of the technical program committee for the careful evaluation of the articles and for the work done within the deadlines. Finally, we congratulate the authors of the accepted papers and thank everyone who submitted papers for the event.

We hope that WESB 2018 will contribute to the formation and consolidation of partnerships in this area, as well as to strengthen the area of Search-Based Software Engineering in Brazil. We wish you all an excellent event!

Maria Cláudia Figueiredo Pereira Emer (DAINF – UTFPR)
Wesley Klewerton Guez Assunção (COTSI – UTFPR)
Program Co-Chairs - WESB 2018

Coordenadores do Comitê de Programa | *Program Chairs*

Maria Cláudia Figueiredo Pereira Emer (DAINF – UTFPR)
Wesley Klewerton Guez Assunção (COTSI – UTFPR)

Organizadores do Comitê de Programa | *Organizing Committee*

Silvia Regina Vergilio (Universidade Federal do Paraná)
Márcio Barros (Universidade Federal do Estado do Rio de Janeiro)
Jerffeson Teixeira de Souza (Universidade Estadual do Ceará)
Thelma Elita Colanzi (Universidade Estadual de Maringá)

Comitê de Programa | *Program Committee*

André Britto de carvalho (UFS)
Celso G. Camilo-Junior (UFG)
Geraldo R. Mateus (UFMG)
Gledson Elias (UFPB)
Gustavo A. L. de Campos (UEC)
Jerffeson T. de Souza (UECE)
Maria Cláudia F. P. Emer (UTFPR)
Márcio Barros (UNIRIO)
Silvia Regina Vergilio (UFPR)
Thelma E. Colanzi (UEM)
Wesley K. G. Assunção (UTFPR)

Sumário | *Table of Contents*

Trabalhos Completos

Uma proposta de geração automática de algoritmos de refatoração
Thainá Mariani, Silvia Vergilio, Marouane Kessentini 1

Uso da Meta-heurística Biased Random Key Genetic Algorithm no Next Release Problem com Configuração Automática de Parâmetros
Andre Andrade, Márcio Barros, Adriana Cesário de Faria Alvim 11

Comparing Low Level Heuristics Selection Methods in a Higher-Order Mutation Testing Approach
Jackson Prado Lima, Silvia Vergilio 21

Estratégias de priorização de características para projeto de arquitetura de LPS baseado em busca
Diego Silva, Matheus Cecílio, Thelma Colanzi 31

N2PC: Um Algoritmo Evolucionário Multiobjetivo baseado em Cones Poliédricos construídos através de Preferências
Pamella Soares, Raphael Saraiva, Allysson Allex Araújo, Gustavo Campos, Jerffeson Souza 41

Análise de estilo de programação com Rede Neural Recorrente para aprovação automática de pull request
Lucas Roque, Altino Dantas, Celso Camilo-Junior 51

Lista de Autores | *Author Index* **61**

Uma proposta de geração automática de algoritmos de refatoração

Thainá Mariani[1], Silvia Regina Vergilio[1], Marouane Kessentini[2]

[1]DInf - Universidade Federal do Paraná – Curitiba, Paraná – Brasil

[2]CIS - University of Michigan – Dearborn, Michigan – United States of America

{tmariani, silvia}@inf.ufpr.br, marouane@umich.edu

Resumo. *A atividade de refatoração têm como principal objetivo aplicar um conjunto de transformações em um artefato de software para melhorar sua estrutura sem alterar sua funcionalidade. Alguns estudos recentes mostram que os desenvolvedores aplicam refatorações similares em diferentes projetos de diferentes contextos. Entretanto, o aprendizado de padrões utilizados na atividade de refatoração ainda é pouco estudado. Técnicas tradicionais de aprendizado de máquina não fornecem o apoio para completamente automatizar esta tarefa, que é bastante complexa. Para lidar com esta limitação, neste trabalho é proposta uma abordagem que utiliza um algoritmo de agrupamento para gerar grupos de classes refatoradas de maneira similar por desenvolvedores. Após isso, um algoritmo de evolução gramatical é utilizado para gerar algoritmos de refatoração com base no aprendizado de características dos grupos identificados na atividade de agrupamento. Os algoritmos gerados são utilizados para detectar oportunidades de refatorações em diferentes projetos com base nas características de seus elementos. Resultados de um estudo de caso são apresentados e mostram vantagens da utilização da abordagem na prática.*

Abstract. *The refactoring activity applies a set of transformations in software artifacts to improve their structure while preserving the functionality. Recent studies show that developer apply similar refactorings in different projects from distinct contexts. However the learning of patterns used in the refactoring activity has been few studied. Traditional machine learning techniques do not completely support this complex task. To overcome such limitation, this work introduces an approach that use clustering algorithms to generate clusters of classes refactored in a similar way by developers. After this, a grammatical evolution technique is used to generate refactoring algorithms based on the characteristics of the clusters identified in the previous activity. The generated algorithms are used to detect refactoring opportunities in different projects according to the characteristics of their elements. Results of a case study are presented showing benefits with the use of the approach in practice.*

1. Introdução

A atividade de refatoração de software tem o objetivo de melhorar a qualidade do software de modo a aprimorar sua estrutura sem alterar seu comportamento externo [Fowler and Beck 1999]. Esta atividade é importante principalmente durante a fase

de manutenção de software para restabelecer a qualidade de um artefato de software melhorando diversos atributos, como por exemplo, entendimento e reusabilidade. Estudos afirmam que a fase de manutenção é responsável por 70% do custo total de um software [Sjøberg et al. 2013]. Desse modo, tal custo pode fazer com que os desenvolvedores não priorizem a atividade de refatoração de software, prejudicando assim a qualidade do software ao longo de sua evolução.

A principal tarefa da atividade de refatoração é a aplicação de pequenas reestruturações também chamadas de refatorações, como por exemplo, mover ou renomear um método. Para aplicar as refatorações é necessário identificar os elementos de um software que precisam ser refatorados, porém o número de possibilidades de aplicações pode ser muito grande uma vez que existem diversos catálogos com inúmeras refatorações [Fowler and Beck 1999]. Todos esses aspectos fazem com que a atividade de refatoração demande tempo e seja considerada difícil [Sjøberg et al. 2013]. Além disso, refatorações podem ser aplicadas várias vezes em diferentes locais, tornando a atividade repetitiva, tediosa e propensa a erros [Meng et al. 2015].

Dentre os artefatos de software existentes, o código fonte é aquele no qual a refatoração é mais aplicada e tem se mostrado mais eficiente [Mens and Tourwe 2004]. Neste contexto, algumas ferramentas e IDEs estão disponíveis com o objetivo de auxiliar o desenvolvedor na aplicação de refatorações. Alguns IDEs populares possuem suporte para a aplicação de refatorações, como é o caso do Eclipse[1] NetBeans[2], IntelliJ[3] e VisualStudio[4]. Outras ferramentas vão além e propõem a sugestão de refatorações para serem aplicadas em determinados elementos do código, como é o caso das ferramentas JDeodorant [Tsantalis et al. 2008] e AutoRefactor[5]. Entretanto, ferramentas deste gênero possuem funcionalidades limitadas, levando em consideração a melhoria de poucos atributos. Por exemplo, JDeodorant sugere refatorações somente com base na identificação de *code smells* e AutoRefactor elimina trechos de código desnecessários. Além do mais, elas não levam em consideração aspectos considerados pelos desenvolvedores ao aplicar uma refatoração.

No contexto de Refatoração baseada em busca (*Search-Based Refactoring* (SBR)) [Mariani and Vergilio 2016] abordagens existentes visam encontrar a melhor sequência de refatorações. Uma vantagem destas abordagens em relação às ferramentas tradicionais, é que os aspectos de um algoritmo de busca permitem a otimização de diversos atributos de qualidade. Além disso, alguns trabalhos [Ouni et al. 2016, Ouni et al. 2015] utilizam refatorações aplicadas por desenvolvedores em outras versões e em outros projetos para guiar a busca por soluções. Entretanto, algumas limitações dificultam o uso destas abordagens na prática. A principal limitação é a falta de generalidade, uma vez que o algoritmo de busca precisa ser configurado e executado toda vez que um novo artefato precisa ser refatorado. Além disso, o usuário deve ter a habilidade de configurar e executar um algoritmo de busca. Na maioria das vezes o desenvolvedor que não conhece esse tipo de algoritmo não opta por este tipo de abordagem.

[1] http://www.eclipse.org
[2] https://netbeans.org
[3] https://www.jetbrains.com/idea
[4] https://visualstudio.microsoft.com
[5] http://autorefactor.org

Neste contexto, um diferencial deste trabalho é o aprendizado a partir de refatorações já aplicadas na prática por desenvolvedores de modo a gerar soluções mais genéricas do que aquelas apresentadas por abordagens tradicionais de SBR e englobando mais atributos que aqueles utilizados por ferramentas tradicionais de refatoração. A abordagem proposta neste trabalho é capaz de gerar um algoritmo de refatoração que define um conjunto de passos para serem seguidos de modo a recomendar refatorações que possam ser aplicadas em diferentes programas. Abordagens existentes de aprendizado de máquina estão limitadas a poucas refatorações, como por exemplo predizer uma classe [Al Dallal 2012] ou um método [Dallal 2017, Xu et al. 2017] para ser refatorado. Além do mais, o aprendizado geralmente é feito por meio de dados gerados artificialmente e não por refatorações aplicadas por desenvolvedores, além de não apresentar soluções que possam ser reutilizadas em vários programas.

Este trabalho propõe uma abordagem baseada em Algoritmos de Agrupamento e Evolução Gramatical (*Grammatical Evolution* (GE)) com o objetivo de gerar automaticamente algoritmos de refatoração com base no aprendizado de padrões extraídos de refatorações aplicadas manualmente por desenvolvedores. A abordagem define duas principais atividades. Primeiramente, um algoritmo de agrupamento é executado com o objetivo de agrupar elementos de código que foram refatorados similarmente. Em seguida, um algoritmo de GE é executado com base em uma gramática pré-definida, e gera algoritmos de refatoração baseados nos padrões de cada grupo identificado na primeira atividade. A gramática possui uma grande flexibilidade, sendo capaz de formalizar um conjunto de regras que identificam onde e como as refatorações devem ser aplicadas. Deste modo, o uso de GE permite a geração de algoritmos mais complexos comparados a soluções geradas por abordagens tradicionais de aprendizado de máquina dos trabalhos relacionados.

A próxima seção resume os trabalhos relacionados. A Seção 3 descreve a abordagem proposta. A Seção 4 descreve um estudo de caso de aplicação da abordagem. A Seção 5 apresenta as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

A maioria dos trabalhos em SBR otimizam uma sequência de refatorações a ser aplicada em elementos de um dado projeto [Mariani and Vergilio 2016]. Alguns trabalhos utilizam o conceito de exemplos para guiar o processo de otimização [Ouni et al. 2016, Ouni et al. 2015]. Os exemplos geralmente são refatorações aplicadas em versões anteriores do projeto ou em projetos de domínio similar. Este tipo de abordagem geralmente tem como saída uma solução otimizada com base em uma métrica chamada "similaridade com exemplos". Essa métrica é comumente utilizada na função de *fitness* juntamente com outras métricas, como por exemplo, número de modificações e coerência semântica [Mariani and Vergilio 2016]. Apesar de as abordagens tradicionais de SBR apresentarem bons resultados, a sequência de refatoração é otimizada especificadamente para um programa, resultando em uma falta de generalidade e reusabilidade, uma vez que a abordagem precisa ser configurada e executada novamente para cada programa.

Os trabalhos que utilizam técnicas de aprendizado no contexto de refatoração de software [Imazato et al. 2017, Kosker et al. 2009, Phongpaibul and Boehm 2007, Jindal and Khurana 2013] utilizam refatorações aplicadas por desenvolvedores em diferentes versões de um mesmo programa, de modo a aprender padrões de refatoração para

serem utilizados em versões futuras. Neste caso, os modelos gerados não são capazes de prever oportunidades de refatoração para vários outros programas.

Os trabalhos mais relacionados com este são os que realizam o aprendizado com base em vários programas, destacando-se [Al Dallal 2012, Dallal 2017, Xu et al. 2017]. Os dois primeiros utilizam dados artificiais ao invés de refatorações aplicadas por desenvolvedores e o último utiliza refatorações reais mas não gera oportunidades de refatorações, prevendo somente como um determinado método deve ser extraído. Outra limitação dessas abordagens é a pouca quantidade de programas utilizados nos experimentos, o que pode impactar na generalidade dos resultados. Além do mais, todas elas exploram somente um tipo de refatoração, que são, respectivamente, Extrair Classe [Al Dallal 2012], Mover Método [Dallal 2017] e Extrair Método [Xu et al. 2017].

3. Abordagem Proposta

A abordagem gera automaticamente um conjunto de algoritmos de refatoração que podem ser reutilizados para sugerir refatorações para elementos de diferentes projetos. Um algoritmo de refatoração é composto por um conjunto de refatorações, cada uma delas associada com características dos elementos que devem ser refatorados. A Figura 1 apresenta uma visão geral da abordagem, que é composta por duas atividades: 1) Agrupamento de Elementos e 2) Execução da Evolução Gramatical.

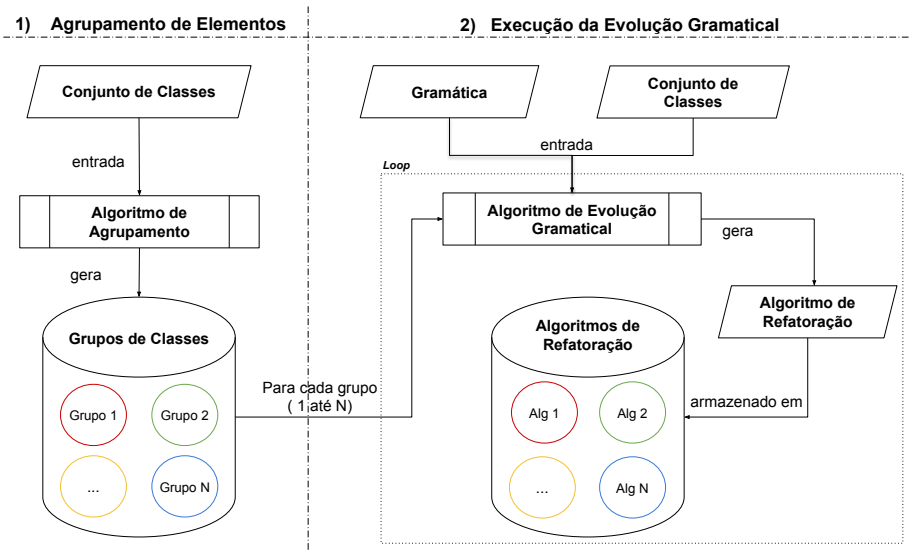


Figura 1. Visão Geral da Abordagem

A atividade de agrupamento gera um conjunto de grupos compostos por classes refatoradas de maneira similar por desenvolvedores, considerando o tipo e o número de refatorações aplicadas em cada uma. O algoritmo de agrupamento recebe como entrada um conjunto de classes extraídas de diferentes programas, sendo cada classe associada com as refatorações aplicadas. Com base nisso, o número de aplicações de cada refatoração é dado como uma dimensão do algoritmo de agrupamento. A Tabela 1 apresenta um exemplo de como a entrada é estruturada. A evolução gramatical gera os algoritmos de refatoração com base no aprendizado de características dos grupos de elementos definidos na atividade anterior. Cada algoritmo de refatoração é composto por várias

Tabela 1. Exemplo de entrada para o algoritmo de agrupamento

Classe	Refatoração	Número de aplicações
Classe1	Mover Método	2 vezes
	Subir Método	2 vezes
	Descer Método	0 vezes
	Extrair Classe	1 vez
	Extrair Método	3 vezes
	Agrupar Método	1 vez

Tabela 2. Atributos utilizados para gerar os algoritmos

Procedimento	Atributo
Selecionar uma classe	Número de classes base (*NCB*)
	Número de classes acopladas (*NCA*)
	Número de classes derivadas (*NCD*)
	Número de métodos (*NM*)
	Número de métodos da hierarquia (*NMH*)
	Número de linhas de código (*NL*)
	Número de linhas comentadas de código (*NLC*)
Selecionar um método	Número de linhas de código (*MNL*)
	Número de linhas comentadas de código (*MNCL*)

operações de refatoração, cada uma delas sugere uma refatoração para ser aplicada em elementos de determinadas características. Deste modo, uma operação de refatoração é composta por dois principais componentes:

1) Refatoração: Este componente representa a refatoração sugerida pela operação de refatoração. As refatorações utilizadas são as mesmas empregadas na fase de agrupamento, conforme descritas na Tabela 1. Estas refatorações são comumente utilizadas em trabalhos de SBR [Mariani and Vergilio 2016].

2) Procedimento: Este componente representa os procedimentos utilizados para selecionar os elementos que devem ser refatorados. Os procedimentos disponíveis são: *selecionar um método* e *selecionar uma classe*. Cada operação de refatoração está associada com um ou mais procedimentos baseados no tipo de elemento que deve ser selecionado. Por exemplo, se a refatoração *mover método* é selecionada, o procedimento *selecionar um método* deve ser executado para selecionar o método a ser movido e o procedimento *selecionar uma classe* deve ser executado para selecionar a classe que vai receber o método. Cada procedimento está associado às características do elemento que deve ser selecionado, definidas por meio de métricas da orientação a objetos, conforme Tabela 2. O elemento a ser selecionado pelo procedimento deve se encaixar entre os intervalos específicos dos valores dos atributos. Por exemplo, se o procedimento *selecionar método* está associado com o intervalo "$10 \leq NL \leq 200$ e $10 \leq NLC \leq 20$", um método que se encaixe nos dois intervalos deve ser selecionado.

A gramática definida para gerar os algoritmos (Figura 2) incorpora as refatorações, os procedimentos, as métricas e os intervalos de valores. Em relação às regras da gramática, os itens entre $\langle\ \rangle$ representam nós não terminais, | representa o operador lógico OR, $::=$ indica que a regra pode ser composta por qualquer um dos próximos valores e os próximos itens são nós terminais. Por exemplo, a regra $\langle NCB \rangle$ pode implementar qualquer um dos valores gramaticais representados por `IntervaloA` até `IntervaloN`. Em contrapartida, $\langle algoritmo \rangle$ pode implementar o valor único $\langle refatoracao \rangle$ ou uma composição representada por $\langle refatoracao \rangle$ $\langle algoritmo \rangle$. A escolha de qual opção implementar é dada pelos genes do vetor que representa cada indivíduo durante a execução do GE.

Cada refatoração da gramática está associada com o conjunto de procedimentos necessários para executá-la, cada procedimento está associado com as métricas e cada

métrica está associada com um intervalo. Os intervalos são criados em tempo de execução e eles podem variar entre 5, 10, 20, 30, 40, 50, 100, 200, 300, 400 e 500 números. Eles são gerados com base nos valores mínimo e máximo de cada métrica em cada grupo de classes. Além do mais, algumas restrições de implementação devem ser adotadas para evitar a geração de somente um intervalo que varie do valor mínimo até o máximo.

```
⟨algoritmo⟩          ::= ⟨refatoracao⟩ ⟨algoritmo⟩ | ⟨refatoracao⟩
⟨refatoracao⟩        ::= ⟨moverMetodo⟩ | ⟨subirMetodo⟩ | ⟨descerMetodo⟩ | ⟨extrairClasse⟩ | ⟨extrairMetodo⟩ | ⟨agruparMetodo⟩
⟨moverMetodo⟩        ::= ⟨selecionarMetodo⟩ ⟨selecionarClasse⟩
⟨subirMetodo⟩        ::= ⟨selecionarMetodo⟩ ⟨selecionarClasse⟩
⟨descerMetodo⟩       ::= ⟨selecionarMetodo⟩ ⟨selecionarClasse⟩
⟨extrairClasse⟩      ::= ⟨selecionarClasse⟩
⟨extrairMetodo⟩      ::= ⟨selecionarMetodo⟩
⟨agruparMetodo⟩      ::= ⟨selecionarMetodo⟩ ⟨selecionarMetodo⟩
⟨selecionarMetodo⟩   ::= ⟨MNL⟩ ⟨MNLC⟩
⟨selecionarClasse⟩   ::= ⟨NCB⟩ ⟨NCA⟩ ⟨NCD⟩ ⟨NM⟩ ⟨NMH⟩ ⟨NL⟩ ⟨NLC⟩
⟨NCB⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NCA⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NCD⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NM⟩                 ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NMH⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NL⟩                 ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨NLC⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨MNL⟩                ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
⟨MNLC⟩               ::= IntervaloA | IntervaloB | IntervaloC | ... | IntervaloN
```

Figura 2. Gramática utilizada para criar os algoritmos de refatoração

A gramática criada é dada com entrada para o GE, o qual é executado para cada grupo de classes com o objetivo de gerar um algoritmo de refatoração, buscando pela melhor combinação de operações de refatorações. Cada solução é representada por um vetor de inteiros, manipulado pelo GE de modo similar a de um algoritmo genético. O processo evolutivo gera um conjunto de soluções e para avaliá-las o vetor é mapeado em um algoritmo de refatoração com base nas regras definidas na gramática.

O algoritmo gerado é avaliado por meio de uma função de *fitness* composta de duas funções objetivo. A primeira mede a similaridade entre o algoritmo gerado e as refatorações relacionadas com as classes do grupo. A função de similaridade $Sim(o)$ leva em consideração separadamente as refatorações aplicadas e as características dos elementos do grupo. A Tabela 3 apresenta as equações utilizadas para calcular a similaridade.

Tabela 3. Equações de Similaridade

$$Sim(o) = \frac{1}{\alpha|O|}\sum_{o\in O, i=1}^{n} RSim(o_i) + CSim(o_i)$$

$$RSim(o) = \frac{1}{|R|}\sum_{r\in R, j=1}^{m} \begin{cases} 1 & \text{if } (r_j == or) \\ 0 & \text{CASO CONTRÁRIO} \end{cases}$$

$$CSim(o) = \frac{1}{|C|}\sum_{c\in C, j=1}^{m} MSim(o, c_j)$$

$$MSim(o, c) = \frac{1}{|M|}\sum_{m\in M, k=1}^{q} \begin{cases} 1 & \text{if } min(m_k) \leq c_k \leq max(m_k) \\ 0 & \text{CASO CONTRÁRIO} \end{cases}$$

A equação define duas principais funções, $RSim(o)$ que mede a similaridade com as refatorações do grupo e $CSim(o)$ que mede a similaridade com as classes do grupo. $Sim(o)$ é calculada para cada operação de refatoração o, em que $o \in O$ e $\alpha = 2$, sendo que O representa todas as operações do algoritmo de refatoração sendo avaliado. $RSim(o)$

mede a porcentagem de similaridade comparando a refatoração or que é o componente de o, com conjunto de refatorações R, que são as refatorações r relacionadas às classes do grupo, de modo que $r \in R$. $CSim(o)$ mede para cada classe $c \in C$ a similaridade das métricas avaliada por $MSim(o, c)$, em que M é o conjunto de métricas relacionadas com a operação o, de modo que $m \in M$. As funções $min(m_k)$ e $max(m_k)$ resultam no intervalo dos valores de uma dada métrica m_k, e c_k representa o valor da métrica da classe c. A similaridade só é computada se c_k está entre o intervalo de valores de m_k.

A segunda função objetivo está relacionada à qualidade dos programas, obtida simulando a aplicação do algoritmo de refatoração em um conjunto de classes de diversos programas diferentes daqueles utilizados na atividade de agrupamento. Após a simulação, a função QMOOD (do inglês, *Quality Model for Object-Oriented Design*) [Bansiya and Davis 2002] é calculada. QMOOD é um modelo utilizado para medir seis atributos de qualidade de software com base em diversas propriedades de projeto, como por exemplo, coesão e acoplamento, possuindo as métricas mais utilizadas no contexto de SBR [Mariani and Vergilio 2016]. A Tabela 4 apresenta os atributos do QMOOD e como eles são calculados. Informações sobre como as propriedades do projeto são calculadas podem ser encontradas em [Bansiya and Davis 2002].

Tabela 4. Funções de QMOOD [Bansiya and Davis 2002]

Atributo de Qualidade	Função
Reusabilidade	-0.25 * Acoplamento + 0.25 * Coesão + 0.5 * Mensagem + 0.5 * Tamanho do Projeto
Flexibilidade	0.25 * Encapsulamento - 0.25 * Acoplamento + 0.5 * Composição + 0.5 * Polimorfismo
Entendimento	-0.33 * Abstração + 0.33 * Encapsulamento - 0.33 * Acoplamento + 0.33 * Coesão - 0.33 * Polimorfismo - 0.33 * Complexidade - 0.33 * Tamanho do Projeto
Funcionalidade	0.12 * Coesão + 0.22 * Polimorfismo + 0.22 Mensagem + 0.22 * Tamanho do Projeto + 0.22 * Hierarquias
Extensibilidade	0.5 * Abstração - 0.5 * Acoplamento + 0.5 * Herança + 0.5 * Polimorfismo
Efetividade	0.2 Abstração + 0.2 * Encapsulamento + 0.2 * Composição + 0.2 * Herança + 0.2 * Polimorfismo

Para obter o valor da função objetivo, a porcentagem de melhoria de QMOOD após a simulação da aplicação da refatoração é calculada. A melhoria varia de 0% até 100%. A Tabela 5 apresenta as equações de qualidade.

Tabela 5. Equações de Qualidade

$Qua(P) = \frac{\sum_{p \in P} IMQ(p)}{\lvert P \rvert}$	$IMQ(p) = \frac{\sum_{f \in F} \frac{f(p) - f(p_o)}{f(p_o)}}{\lvert F \rvert}$

Nas equações demonstradas acima, P representa o conjunto de programas, sendo p um programa do conjunto P. F representa o conjunto de funções de QMOOD, f é uma função de F, $f(p)$ é o valor de f para p depois da simulação das refatorações e $f(p_o)$ é o valor original de f para p. Com base nas duas funções objetivo, a função de *fitness* é definida pela média aritmética dada por:

$$Fitness = \frac{Sim(o) + Qua(P)}{2} \tag{1}$$

Ao final da execução do GE, o algoritmo de refatoração com a melhor função de *fitness* é armazenado em um repositório. A geração de algoritmo continua para o próximo grupo até que não haja mais grupos. Os algoritmos gerados podem então ser executados em outros projetos para detectar novas oportunidades de refatoração.

4. Resultados de um estudo de caso

Um estudo de caso foi conduzido para avaliar como a abordagem proposta pode contribuir para a melhoria da qualidade de um programa. Com este objetivo, primeiramente, o algoritmo de agrupamento *Expectation Maximization* (EM) [Dempster et al. 1977] foi utilizado para gerar os grupos de elementos. Refatorações de 40 programas Java de diferentes domínios foram utilizadas nesta atividade. Os programas foram extraídos do *GitHub* e as refatorações foram extraídas de *http://aserg-ufmg.github.io/why-we-refactor/#/projects*. Na segunda atividade, o GE foi executado para cada grupo e os algoritmos de refatoração foram gerados. Neste estudo de caso, uma função de qualidade que avalia a modularidade de um programa, denominada *Modularity Quality* (MQ) [Paixao et al. 2017], foi utilizada. Posteriormente, a função será implementada com base em todos os atributos de qualidade de QMOOD, conforme proposta deste trabalho.

A Figura 3 apresenta um fluxograma que exemplifica a aplicação dos algoritmos gerados em um novo programa dado como entrada. Para cada algoritmo (Alg1 até Alg7) é apresentada a quantidade de operações geradas. Para ilustrar a estrutura de um algoritmo, o pseudocódigo do Algoritmo 6 é exibido, o qual define regras para a seleção de classes candidatas a receber a aplicação da refatoração "Extrair Classe".

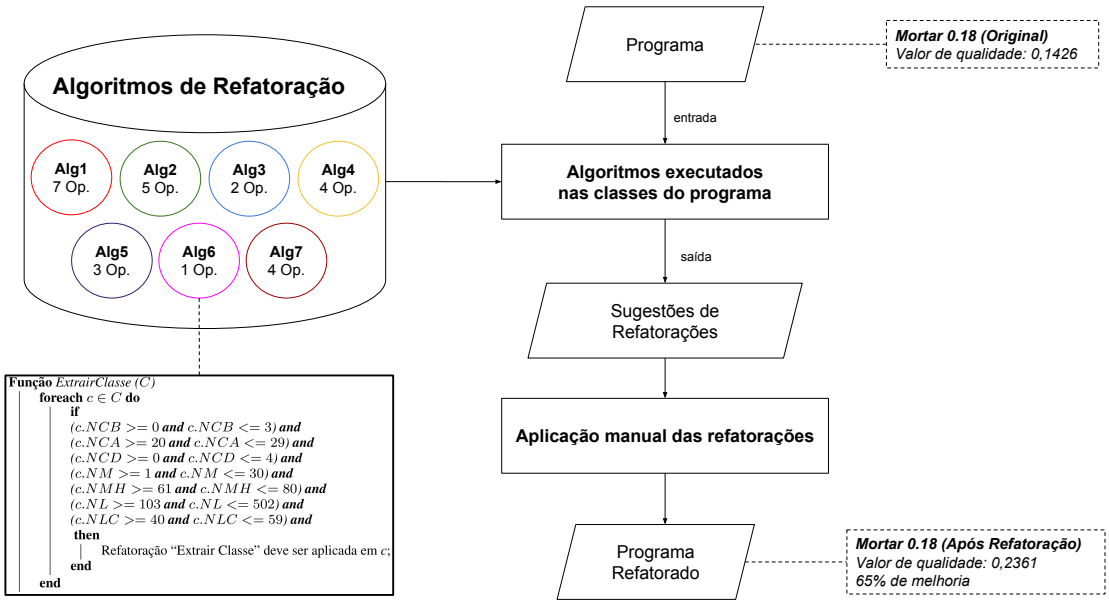


Figura 3. Exemplo de aplicação dos algoritmos gerados

Os algoritmos gerados são executados sobre o programa dado como entrada. Neste estudo, o programa *Mortar 0.18* (*https://github.com/square/mortar*) foi utilizado. O valor de MQ do programa original é de $0,1426$, podendo variar de 0 até 1. Como resultado, um conjunto de sugestões de refatorações é obtido. Depois disso, foi realizada a

aplicação manual das refatorações sugeridas e calculado novamente o valor de qualidade do programa. O valor obtido foi de $0,2361$, resultando em uma melhoria de 65%.

Além de comparar a melhoria do programa em relação à versão original, foi realizada também a comparação com base em refatorações aplicadas por desenvolvedores. Para isso, foi calculada a função de qualidade para o programa da versão posterior (*Mortar 0.19*), que consiste no programa utilizado originalmente após a aplicação de várias refatorações. No caso da nova versão disponível, as refatorações foram capazes de melhorar o valor da função de qualidade somente em 3%, contra 65% de melhoria obtida com os algoritmos gerados. Além disso, as refatorações aplicadas foram analisadas de modo a verificar se o algoritmo gerado pela abordagem seria capaz de identificar essas refatorações. Como resultado, os algoritmos gerados pela proposta deste trabalho seriam capazes de encontrar 97% dessas refatorações, além de outras refatorações sugeridas.

5. Conclusão

Neste trabalho uma proposta de abordagem para gerar algoritmos de refatoração foi apresentada. A abordagem proposta é composta por duas atividades. A primeira é responsável por executar um algoritmo de agrupamento visando agrupar classes que foram refatoradas similarmente por desenvolvedores. Deste modo, o algoritmo de agrupamento recebe como entrada um conjunto de classes extraídas de diferentes programas. A segunda atividade incorpora um algoritmo de evolução gramatical que, baseado em uma gramática pré-definida, é executado de modo a gerar algoritmos de refatoração que aprendem as características de cada grupo identificado na primeira atividade. Os algoritmos gerados são avaliados com base na similaridade com as refatorações aplicadas por desenvolvedores e com a melhoria de aspectos qualitativos.

A abordagem foi implementada e avaliada em um estudo de caso que mostrou que os algoritmos gerados foram capazes de melhorar o valor da função de qualidade em 65% no programa utilizado. Além disso, os algoritmos puderam encontrar refatorações similares àquelas já aplicadas por desenvolvedores. Como próximos passos, experimentos com outros programas serão ser conduzidos e avaliados, bem como a utilização de vários atributos de qualidade na geração dos algoritmos.

Referências

Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 54(10):1125–1141.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

Dallal, J. A. (2017). Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, 92:105 –120.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38.

Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Imazato, A., Higo, Y., Hotta, K., and Kusumoto, S. (2017). Finding extract method refactoring opportunities by analyzing development history. In *Proceedings of the 41st Annual Computer Software and Applications Conference*, COMPSAC, pages 190–195.

Jindal, S. and Khurana, G. (2013). The statistical analysis of source-code to determine the refactoring opportunities factor (rof) using a machine learning algorithm. In *International Conference on Advances in Recent Technologies in Communication and Computing*, ARTCom '13, pages 396–403.

Kosker, Y., Turhan, B., and Bener, A. (2009). An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003.

Mariani, T. and Vergilio, S. R. (2016). A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34.

Meng, N., Hua, L., Kim, M., and McKinley, K. S. (2015). Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 392–402. IEEE Press.

Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 25(3):23:1–23:53.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Hamdi, M. S. (2015). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105:18–39.

Paixao, M., Harman, M., Zhang, Y., and Yu, Y. (2017). An empirical study of cohesion and coupling: Balancing optimisation and disruption. *IEEE Transactions on Evolutionary Computation*, PP(99).

Phongpaibul, M. and Boehm, B. (2007). Mining software evolution to predict refactoring. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 265–274.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR, pages 329–331.

Xu, S., Sivaraman, A., Khoo, S. C., and Xu, J. (2017). Gems: An extract method refactoring recommender. In *Proceedings of the 28th International Symposium on Software Reliability Engineering*, ISSRE, pages 24–34.

Uso da Meta-heurística *Biased Random Key Genetic Algorithm* no *Next Release Problem* com Configuração Automática de Parâmetros

André de Souza Andrade[1], Márcio de Oliveira Barros[1], Adriana C. de F. Alvim[1]

[1]Centro de Ciências Exatas e Tecnologia –
Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Avenida Pasteur, 458 – Urca - Rio de Janeiro – RJ – Brasil

{andre.andrade,marcio.barros,adriana}@uniriotec.br

Abstract. *The Next Release Problem is a combinatorial optimization problem of the Requirements Engineering area. This work used the Biased Random Key Genetic Algorithm metaheuristic to solve a mono-objective version of the problem using IRACE and K-Fold Cross Validation for parameters tuning. Our contribution lays on presenting the results of a systematic approach to calibrate algorithmic parameters in addressing the NRP, replacing the human expertise by the scientific method.*

Resumo. *Next Release Problem é um problema de otimização combinatória da área de Engenharia de Requisitos. Este trabalho utilizou a meta-heurística Biased Random Key Genetic Algorithm para resolver uma versão mono-objetivo do problema, com o suporte da ferramenta IRACE e de K-Fold Cross Validation para a configuração automática de parâmetros. A contribuição deste trabalho reside nos resultados do uso de um método sistemático para a escolha dos parâmetros do algoritmo selecionado para encontrar soluções para o NRP, substituindo a experiência humana pelo método científico.*

1. Introdução

O planejamento de versões de software a fim de maximizar a satisfação do cliente, considerando restrições de recursos para a implementação de novos requisitos, é determinante no sucesso de um software – decidir em qual momento qual *feature* será entregue é tão relevante quanto levantar e identificar corretamente os requisitos. Tal planejamento foi modelado como um problema de otimização combinatória da área de Engenharia de Requisitos e é conhecido como o Problema do Próximo Release, em inglês *Next Release Problem* (NRP). Este problema é classificado como NP-Difícil [Bagnall et al., 2001], uma categoria de problemas cuja solução exata para a maioria de suas instâncias consome um tempo de processamento inaceitável.

Muitas soluções baseadas em busca já foram propostas para encontrar soluções para o NRP [Baker et al., 2006], mas até o presente momento, não há conhecimento de trabalhos de pesquisa publicados que utilizaram a técnica *Biased Random Key Genetic Algorithm* (BRKGA) [Gonçalves & Resende, 2011], uma meta-heurística variante de *Random Key Genetic Algorithm* (RKGA), baseada em computação evolutiva (algoritmos genéticos) e que tem demonstrado bons resultados ao ser aplicada em problemas de outros domínios do conhecimento. Além do uso de BRKGA no NRP, este trabalho propõe um novo método construtivo para acelerar a identificação de soluções para o problema e adequado à representação da solução definida pelo BRKGA.

Como toda meta-heurística, BRKGA possui parâmetros que são independentes do problema a ser abordado e dizem respeito às propriedades de um algoritmo genético, como o tamanho da população, o número de gerações, o percentual de cromossomos do estrato elite (cromossomos cuja avaliação apresenta os melhores *fitness* na população) que são levados para a próxima geração, o número de novos cromossomos que serão gerados na próxima geração (mutantes) e a probabilidade de que um gene de um cromossomo do estrato elite seja herdado por seu descendente na próxima geração em um cruzamento (*crossover*). A escolha dos valores para os dois primeiros parâmetros foi arbitrada de acordo com os resultados de um estudo de convergência para as instâncias disponíveis. Os valores dos três últimos parâmetros foram determinados pela ferramenta de configuração automática de parâmetros IRACE [López-Ibánez et al., 2011].

Dada a quantidade relativamente pequena de instâncias disponíveis para comparação com a literatura (39 instâncias), foi utilizada a técnica *K-Fold Cross Validation* [Kohavi, 1995] associada ao IRACE. Esta técnica é usada para evitar *overfitting* [Bergmeir et al, 2015], situação indesejada em que os parâmetros da meta-heurística estariam muito adequados às 39 instâncias, resultando em bons valores de *fitness*, porém com o risco desses parâmetros resultarem em valores de *fitness* ruins quando aplicados a outras instâncias do problema. A técnica consiste em particionar o conjunto de instâncias, onde os elementos de uma partição são utilizados como instâncias de teste e os elementos das demais partições como instâncias de treinamento. Cada partição tem seu momento em que será a partição de instâncias de teste, seguindo um rodízio. Foram propostas seis variações do número de partições: 3, 4, 5, 6, 8 e 13.

De posse da melhor configuração de parâmetros, a meta-heurística foi executada com o método construtivo proposto a fim de calcular medidas de tendência central e de dispersão dos seus resultados e compará-las com outros trabalhos da literatura. Os resultados demonstram que a meta-heurística BRKGA associada ao método construtivo proposto, configurada pelo IRACE com o *K-Fold Cross Validation*, gera resultados comparáveis ao melhores resultados encontrados na literatura.

2. Descrição do problema

Seja $\boldsymbol{R}$ o conjunto de $\boldsymbol{m}$ requisitos candidatos à próxima versão do software, onde cada requisito $\boldsymbol{i} \in \boldsymbol{R}$ tem um custo não negativo c_i. Seja $\boldsymbol{S}$ o conjunto de $\boldsymbol{n}$ clientes interessados na próxima versão do software e seja $\boldsymbol{R_j}$ o conjunto de requisitos de interesse do cliente $\boldsymbol{j}$, onde $\boldsymbol{R_j} \subseteq \boldsymbol{R}$, $\forall \boldsymbol{j} \in \boldsymbol{S}$. As dependências entre requisitos são modeladas como um grafo acíclico direcionado $\boldsymbol{G = (V, E)}$, onde o conjunto de vértices $\boldsymbol{V}$ é igual ao conjunto de requisitos $\boldsymbol{R}$ e o conjunto de arestas $\boldsymbol{E}$ expressa a dependência entre dois requisitos de tal forma que se $\boldsymbol{(r', r)} \in \boldsymbol{E}$, então o requisito $\boldsymbol{r}$ depende do requisito $\boldsymbol{r'}$, ou seja, se $\boldsymbol{r}$ for escolhido para compor a próxima versão, $\boldsymbol{r'}$ também deverá ser escolhido de modo a satisfazer a dependência. Seja ***ancestrais(r)*** o conjunto dos requisitos que devem ser implementados para garantir a implementação de $\boldsymbol{r}$. Estes requisitos são representados por todos os vértices para os quais existe um caminho que chega em $\boldsymbol{r}$, incluindo o próprio vértice $\boldsymbol{r}$. A definição recursiva de ***ancestrais(r)*** é:

(1) Se $\boldsymbol{u} = \boldsymbol{r}$, então $\boldsymbol{u} \in$ ***ancestrais(r)***.
(2) Se $\boldsymbol{(u,v)} \in \boldsymbol{E}$ e $\boldsymbol{v} \in$ ***ancestrais(r)***, então $\boldsymbol{u} \in$ ***ancestrais(r)***.
(3) Se $\boldsymbol{u} \in$ ***ancestrais(r)***, então $\boldsymbol{u}$ foi introduzido pelas regras 1 ou 2.

Por analogia, os requisitos que devem ser implementados para garantir a implementação de R_j são representados por $ancestrais(R_j) = U_{r \in R_j}\ ancestrais(r)$. Para qualquer subconjunto de requisitos $R' \subseteq R$, seu custo de implementação é dado por $custo(R') = \Sigma_{i \in R'}\, c_i$. Portanto, um cliente $j \in S$ é atendido quando todos os requisitos em $ancestrais(R_j)$ são implementados, o que gera um lucro igual a ω_j e custo igual a $custo(ancestrais(R_j)) = \Sigma_{i \in ancestrais(R_j)}\, c_i$. Dado que o orçamento para a próxima versão do software é limitado por uma constante positiva b, a solução para o *NRP* consiste em atender um subconjunto de clientes $S' \subseteq S$ tal que o lucro seja maximizado, respeitando a restrição de orçamento. Desta forma, uma solução S' pode ser representada como um vetor $X = (x_1, x_2, x_3 ,.., x_n)$ de n posições, onde $x_j = 1$ indica que o cliente j é atendido e $x_j = 0$ indica o contrário, em que se deseja maximizar o lucro ω expresso por $\Sigma_{j=1..n}\, \omega_j x_j$, sujeito à restrição de orçamento dada por $custo(U_{j=1..n}\ \{ancestrais(R_j) : x_j = 1\}) \leq b$.

Como um exemplo simples, seja uma instância definida por três clientes e oito requisitos. Seja o vetor ω = (30, 25, 50), referente aos lucros esperados dos clientes caso sua lista de requisitos seja implementada. Seja o vetor c = (6, 10, 16, 4, 1, 7, 6, 1), referente ao custo de cada requisito. Seja a restrição de orçamento b igual a 36 e o grafo de dependência entre requisitos e a relação de pertinência entre requisitos e clientes representados pela Figura 1. Com isto, é possível definir os requisitos que devem ser implementados para satisfazer cada cliente e seus custos: $R_1 = \{r_1, r_3, r_4\}$, com custo 26, $R_2 = \{r_1, r_2, r_4, r_5, r_6, r_7, r_8\}$, com custo 35, e $R_3 = \{r_2, r_6, r_8\}$, com custo 18.

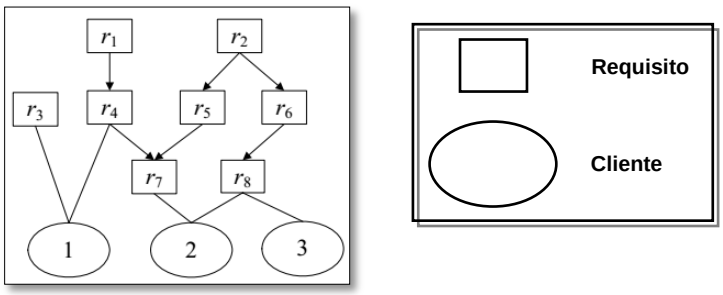


Figura 1. Dependência entre requisitos e relação entre requisitos e clientes

O espaço de soluções do NRP é o conjunto de todos os subconjuntos de clientes, ou seja, 2^n, onde n é o número de clientes. Para este exemplo, têm-se oito soluções candidatas. No entanto, devido ao orçamento, nem todas as soluções são viáveis: uma solução que atenda aos clientes 1 e 2 ou que atenda aos clientes 1 e 3 não são viáveis, pois a primeira tem um custo 51 e a segunda 44, ambas excedendo o orçamento de 36.

3. Proposta de solução

A solução proposta consiste em três componentes. As linguagens de programação utilizadas na construção destes componentes foram C++, para o método construtivo implementado no framework da meta-heurística, e R, para o *K-Fold Cross Validation*.

3.1. *BRKGA* e o método construtivo

Utilizando o framework desenvolvido em C++ pelo autor da meta-heurística BRKGA [Toso et al., 2015], foi desenvolvido um método construtivo para o NRP, conforme a Figura 2. Tal método observa a chave aleatória atribuída pela meta-heurística a cada cliente e informada ao método através da estrutura de dados *cromossomo*, e ordena os clientes em ordem crescente, do menor ao maior valor desta chave aleatória (linha 1).

Respeitando essa ordem (linhas 2 e 4), o método calcula o custo para implementar os requisitos de cada cliente (linha 10), desconsiderando os requisitos associados a clientes antecessores na ordenação e já contemplados (linha 9) na solução.

A cada requisito, o método construtivo verifica se seu custo cabe no orçamento (linha 11). Caso afirmativo, segue para o próximo requisito do cliente em avaliação ou, caso não haja mais requisitos para o cliente, segue para o próximo cliente, atualizando o repositório de requisitos visitados, o custo total dos requisitos até agora implementados e o lucro da solução, valor este que se busca maximizar (linhas 17, 18 e 19). Caso não caiba no orçamento, segue para o próximo cliente (linha 12), pois há a possibilidade de que ainda haja clientes na ordem informada pela meta-heurística, cujo conjunto de requisitos, alguns dos quais já implementados por estarem associados a clientes já incluídos na solução, caiba no orçamento. Esse procedimento de avaliação segue até o último cliente na ordem informada.

```
 1: Ordena(cromossomo)
 2: numClientes ← ObtemTamanho(cromossomo)
 3: custoTotal ← 0
 4: for i from 1 to numClientes do
 5:    requisitosCliente ← ObtemRequisitos(i)
 6:    numRequisitos ← ObtemTamanho(requisitosCliente)
 7:    custoCliente ← 0
 8:    for j from 1 to numRequisitos do
 9:       if VerificaVisitado(requisitosCliente[j]) = false then
10:          custoCliente ← custoCliente + ObtemCusto(requisitosCliente[j])
11:          if custoCliente + custoTotal >= BUDGET then
12:             break
13:          end if
14:       end if
15:    end for
16:    if custoCliente + custoTotal <= BUDGET then
17:       MarcaVisitado(requisitosCliente)
18:       custoTotal ← custoTotal + custoCliente
19:       lucroCliente ← lucroCliente + ObtemLucro(i)
20:    end if
21: end for
```

Figura 2. Pseudocódigo do método construtivo

3.2. IRACE

Como toda meta-heurística, o BRKGA possui parâmetros independentes do problema a ser abordado. Configurar tais parâmetros não é uma tarefa trivial e por si só constitui-se em um problema NP-Difícil [Birattari & Kacprzyk, 2009]. Por esse motivo, para boa parte das meta-heurísticas existentes, a definição de valores para cada parâmetro e as possíveis combinações desses valores que definem uma configuração, em geral, ocorre de três formas: a) replicando os valores de trabalhos anteriores que usaram a mesma meta-heurística em problemas similares; b) baseada na experiência do pesquisador e; c) através de ferramentas de configuração automática de parâmetros.

No primeiro caso, embora se busque a semelhança entre os problemas em suas definições e características, as instâncias de cada problema podem possuir particularidades que degradem a qualidade da solução gerada (*fitness*): onde se observou um valor de *fitness* muito bom para uma combinação de valores dos parâmetros, agora se observa um valor ruim para a mesma combinação com um conjunto diferente de instâncias.

No segundo caso, o pesquisador deve conhecer bem o problema, como detalhes de suas instâncias e a própria meta-heurística, de forma a observar se existe alguma relação entre as características das instâncias e os parâmetros da meta-heurística e usar este conhecimento para realizar a configuração dos parâmetros, o que traz uma forte dependência de um fator humano: a experiência ou expertise do pesquisador.

Já no terceiro caso, uma ferramenta de configuração automática de parâmetros tem métodos que orientam a escolha da melhor combinação de valores para os parâmetros da meta-heurística. Essa escolha é realizada com base na comparação de valores de *fitness* resultantes de sucessivas execuções da meta-heurística, que recebeu como entrada os valores dos parâmetros passados pela ferramenta de configuração automática. A ferramenta realiza as próximas escolhas de valores dos parâmetros dando preferência àqueles situados na mesma região do domínio daqueles que proporcionaram os melhores valores de *fitness* resultantes da meta-heurística [López-Ibánez et al., 2016].

O software IRACE foi escolhido como ferramenta para a configuração automática de parâmetros porque o autor da meta-heurística tem usado essa ferramenta em trabalhos recentes de 2015 e 2016 [Chaves et al., 2016], [Lopes et al., 2016], [Stefanello et al., 2015], [Andrade et al., 2015].

Para que o software IRACE possa definir uma ordem de melhores e piores configurações através de testes estatísticos e ao final de sua execução chegar à configuração que apresentou os melhores resultados, a implementação do BRKGA deve entregar para o software um valor comparável e independente das diferentes instâncias utilizadas. O valor aqui proposto é a distância relativa entre a solução encontrada e a soma dos lucros de todos os clientes, soma esta bem acima do ótimo, dada as restrições de orçamento para o custo dos requisitos. Para representar tal valor, a fórmula utilizada foi: "*(lucro total - solução encontrada)/lucro total*". Dessa forma, seja qual for a ordem de grandeza do lucro encontrado para uma instância em particular, o valor entregue ao IRACE é relativo e portanto, comparável. Como o IRACE trabalha com minimização da função objetivo, a fórmula está adequada.

3.3. *K-Fold Cross Validation*

Com o objetivo de evitar *overfitting*, a técnica *K-Fold Cross Validation* é utilizada em associação com o IRACE, que por sua vez usa instâncias de treinamento para a configuração automática de parâmetros e instâncias de teste para validar a configuração ou conjunto de configurações que resultou nos melhores valores de *fitness* para uma parte significativa das instâncias de treinamento.

K-Fold Cross Validation consiste em dividir o conjunto P de todas as instâncias em K partições de aproximadamente mesmo tamanho, P_1, P_2, ..., P_K. Tanto a etapa de treinamento quanto a de teste ocorrerão K vezes. Em cada momento $t \in \{1, 2, ..., K\}$, as instâncias de treinamento utilizadas pertencerão ao subconjunto $P \setminus P_t$ e as de teste ao subconjunto P_t. Se K não for um divisor de $|P|$, será criada uma nova partição com as instâncias restantes. O treinamento é feito em K rodadas, onde a cada rodada uma partição é considerada como partição de instâncias de teste e as demais como partições de instâncias de treinamento. No caso deste trabalho, $K \in \{3, 4, 5, 6, 8, 13\}$, valores desdobrados a partir da quantidade escolhida de instâncias que cada partição (ou *Fold*) deveria ter: 3, 5, 7, 9, 11 e 15 instâncias por partição. Por exemplo, considerando as 39 instâncias, para o caso de 5 instâncias por partição, tem-se K=8, com o detalhe que 7

partições possuem 5 instâncias e uma única partição possui 4 instâncias (*K* não é um divisor de |P|). A Figura 3 ilustra o exemplo.

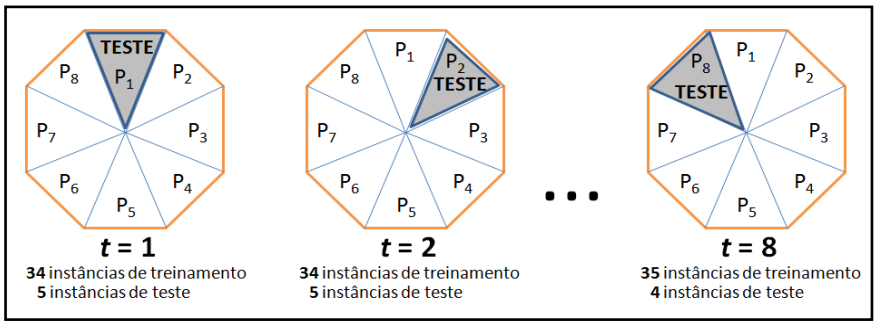


Figura 3. Exemplo com *K* = 8

4. Experimentos

Os experimentos foram divididos em três etapas distintas, com o resultado da etapa anterior servindo de entrada para a etapa subsequente.

4.1. Estudo de convergência

O objetivo do estudo de convergência foi entender o comportamento assintótico da meta-heurística associada ao método construtivo proposto, de forma a determinar valores para os parâmetros de tamanho da população e número de gerações. A decisão de fixar tais parâmetros veio da constatação de que quanto maiores os valores, melhor a qualidade das soluções geradas (*fitness*), porém na contrapartida de maiores tempos de processamento. Como este trabalho busca tempos de processamento comparáveis aos da literatura, foi realizada uma escolha parcimoniosa para estes dois parâmetros, para os quais o estudo de convergência mostrou valores de *fitness* próximos aos da literatura.

Uma versão modificada do framework em C++ foi desenvolvida para gravação em arquivo do melhor *fitness* de cada uma das 300 (trezentas) gerações para populações de cinco tamanhos diferentes: 100, 150, 200, 250 e 300 cromossomos. Cada instância foi executada 30 vezes de forma independente, ou seja, cada uma das execuções teve uma semente aleatória diferente. Também foi desenvolvido um script R para a leitura dos dados gerados, processamento e construção de gráficos para auxiliar na escolha dos parâmetros. Ao todo, foram gerados 39 gráficos, cada um correspondendo a uma instância. Vale salientar que cada valor de *fitness* no gráfico é a média dos resultados das 30 execuções. A Figura 4 mostra os gráficos para as instâncias *nrp-g4-30* e *nrp-m1-30*, onde os valores de *fitness* estão entre parênteses.

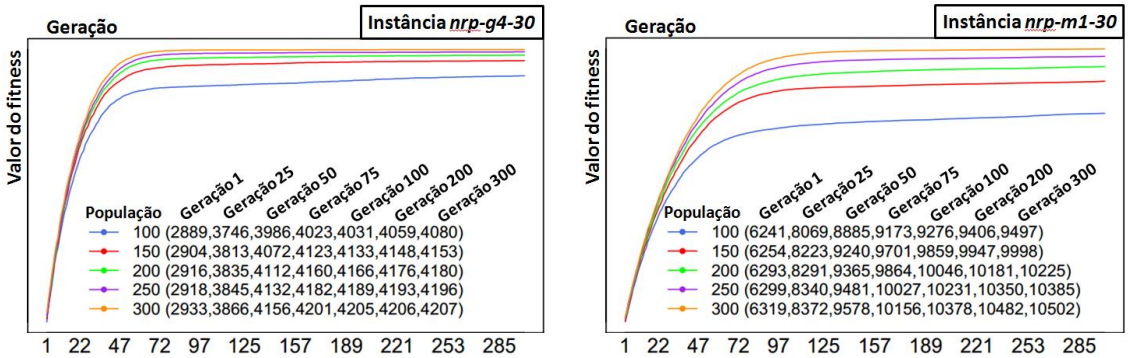


Figura 4. Estudo de convergência: instâncias *nrp-g4-30* e *nrp-m1-30*

Com a inspeção visual dos gráficos, foram definidos os seguintes valores para o número de gerações: 125, 100 e 75 respectivamente para as instâncias com orçamento de 30%, 50% e 70% do valor total do custo de todos os requisitos. A justificativa é que a curva para cada instância adquire um comportamento assintótico a partir desses valores, com um crescimento lento até o melhor *fitness* alcançado na geração 300.

Para a definição do tamanho da população, foi elaborada uma tabela com os valores de *fitness* de cada uma das instâncias para os cinco diferentes tamanhos de população ao final do número de gerações escolhido acima. Por exemplo, os valores de *fitness* das instâncias que consideram 30% do orçamento são da geração 125. A Tabela 1 exibe parte destes dados. A coluna "Ganho" informa o quanto o *fitness* aumentou em relação ao *fitness* obtido com o tamanho da população imediatamente à esquerda. Ela representa, em percentuais, a distância vertical entre duas curvas correspondentes a dois tamanhos da população na Figura 4. A partir da população de tamanho 200, esses ganhos começaram a ficar cada vez menores, sugerindo que as curvas estavam se aproximando entre si mais lentamente. Como pouco ganho há de se obter em populações de tamanho maior que 200 e como o aumento do tamanho da população faz com que o tempo de processamento aumente, o tamanho da população foi estabelecido como 200.

Tabela 1. Parte da tabela utilizada na escolha do tamanho da população

	TAMANHO DA POPULAÇÃO								
	100	150		200		250		300	
	Fitness	Fitness	Ganho	Fitness	Ganho	Fitness	Ganho	Fitness	Ganho
nrp1-30	1186	1193	0,59%	1197	0,34%	1199	0,17%	1200	0,08%
nrp1-50	1809	1817	0,44%	1820	0,17%	1822	0,11%	1823	0,05%
nrp1-70	2507	2507	0,00%	2507	0,00%	2507	0,00%	2507	0,00%
nrp2-30	4471	4634	3,65%	4679	0,97%	4746	1,43%	4786	0,84%
nrp2-50	7545	7688	1,90%	7789	1,31%	7862	0,94%	7894	0,41%
nrp2-70	11067	11129	0,56%	11179	0,45%	11212	0,30%	11224	0,11%
nrp3-30	6755	6968	3,15%	7127	2,28%	7206	1,11%	7267	0,85%

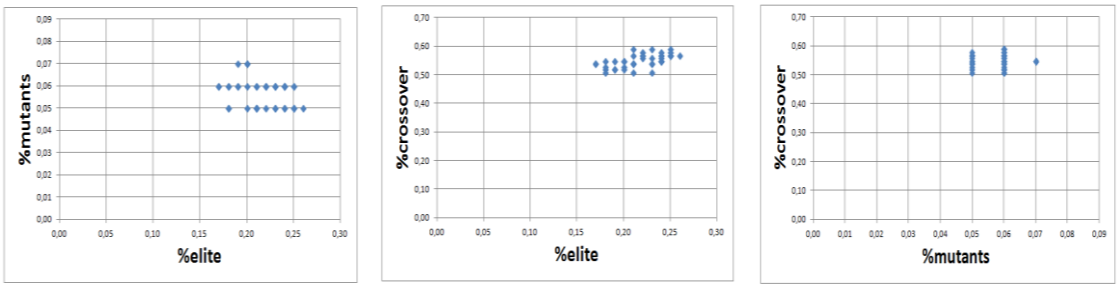
4.2. Configuração automática de parâmetros

A próxima etapa tem como objetivo definir o percentual de cromossomos do estrato elite que serão levados para a próxima geração, o percentual de novos cromossomos que serão gerados na próxima geração (mutantes) e a probabilidade, em um cruzamento, que um gene de um cromossomo do estrato elite tem, sobre um gene de um cromossomo não elite, de ser herdado por seu descendente na próxima geração. Para tanto, a ferramenta IRACE foi utilizada, associada com a técnica *K-Fold Cross Validation*. Com base em trabalhos anteriores do autor da meta-heurística, a Tabela 2 mostra os intervalos que foram informados ao IRACE como domínio de valores sobre o qual a ferramenta trabalhou para encontrar a configuração de parâmetros que apresentasse o melhor valor de *fitness*.

Tabela 2. Intervalos usados como domínio dos parâmetros pelo IRACE

Parâmetro	Descrição	Intervalo
%elite	Percentual de cromossomos da elite	[0,10, 0,30]
%mutants	Percentual de cromossomos mutantes	[0,05, 0,30]
%crossover	Probabilidade de prevalência do gene elite no *crossover*	[0,50, 0,80]

Foram realizadas 42 execuções independentes do IRACE, cada qual com cerca de 1500 execuções da meta-heurística (método construtivo): 13 execuções (K=13) quando o número de instâncias por partição é três, oito execuções (K=8) quando tal número é cinco, e assim por diante. Cada execução do IRACE gerou uma combinação de valores para os três parâmetros. A fim de orientar a escolha dos valores para os três parâmetros a partir das 42 combinações geradas pelas execuções independentes, foram gerados três gráficos do tipo *scatter plot*, mostrados na Figura 5.



(a) (b) (c)

Figura 5. Resultados das 42 execuções do IRACE para os três parâmetros

O valor 0,21 foi escolhido para o percentual de cromossomos do extrato elite, visto que ocorre com maior frequência no gráfico (b) (seis ocorrências) e está próximo à média 0,22. Para o percentual de cromossomos mutantes, a média não foi considerada devido à alta frequência (23 ocorrências) do valor 0,06 no gráfico (c), contra 16 ocorrências do valor 0,05 e apenas 3 ocorrências do valor 0,07. Por fim, para selecionar a probabilidade de prevalência do gene elite no *crossover*, foi fixado o valor de 0,21 de %elite no gráfico (b) e selecionado o valor central de 0,55 do intervalo do %crossover.

4.3. Execução com a melhor configuração

Definidos os valores para os cinco parâmetros do BRKGA, a terceira e última etapa dos experimentos foi executar a meta-heurística 30 vezes para cada uma das 39 instâncias, com o objetivo de medir a média e o desvio padrão em relação à média do *fitness* e do tempo de execução. Para garantir reprodutibilidade, a cada uma das 30 execuções foi associada uma semente aleatória diferente, utilizada por todas as instâncias. Como exemplo, a semente associada à quinta execução da instância *nrp1-30* foi a mesma semente utilizada na quinta execução das instâncias *nrp-g4-30*, *nrp-m1-30* e das demais.

5. Análise dos resultados

Para cada uma das 39 instâncias, foram obtidos a mediana, a média e o desvio padrão em relação à média dos valores de *fitness* gerado pelas 30 execuções. Para a comparação com os melhores resultados da literatura [Xuan et al., 2012], foi utilizado o teste não paramétrico de Wilcoxon-Mann-Whitney (α=0.05), que usa as medianas e não exige normalidade e homocedasticidade das amostras comparadas.

A Tabela 3 exibe informações para a comparação entre o BRKGA e os melhores resultados da literatura (BMA). Em “MELHOR FITNESS”, constam a melhor solução obtida nas 30 execuções do BRKGA, a melhor solução obtida nas execuções de BMA e a diferença percentual entre estes dois valores, representada pela coluna “%” e segundo a fórmula “100 x ((BRKGA-BMA)/BMA)”. Em “FITNESS MÉDIO”, constam a média do *fitness* obtido nas 30 execuções do BRKGA, a média do *fitness* obtido nas execuções de BMA, a diferença percentual entre estes dois valores (coluna “%”) e o desvio padrão

em relação à média de BRKGA (coluna "DP"). Em "MEDIANA (FITNESS)", constam a mediana do *fitness* obtido nas 30 execuções do BRKGA e o *p-value* (coluna "PV"), para saber se os resultados de BRKGA e BMA são significativamente diferentes. Como se tem apenas a melhor solução e a média de BMA, apenas estes dois valores foram utilizados nos testes estatísticos contra os resultados das 30 execuções do BRKGA. Em "TEMPO MÉDIO DE EXECUÇÃO", consta a média dos tempos aferidos em cada uma das 30 execuções do BRKGA. Para um melhor aproveitamento do espaço destinado a este documento, apenas as informações de algumas instâncias são exibidas.

De acordo com as duas colunas "%", com exceção da instância *nrp1-70*, onde resultados iguais foram obtidos, o BRKGA superou o BMA em todas as instâncias, tanto na melhor solução quanto na média. No entanto, o *p-value* foi maior que 0,05 em sete instâncias, indicando não ser possível afirmar que os resultados são significativamente diferentes. Percebe-se que instâncias com "%" < 1,00 possuem "PV" > 0,05 (*nrp-g2-50* como exceção, com "PV" 0,02) e aquelas com "%" > 1,00 possuem "PV" < 0,05 (*nrp2-30* e *nrp1-50* como exceções, com "PV" 0,07 e 0,53 respectivamente). Tal fato pode indicar uma relação entre o *p-value* e a diferença percentual ("%").

Em relação ao tempo de processamento, mesmo ambos os trabalhos utilizando C++ para a implementação dos algoritmos, a comparação não foi possível, uma vez que as configurações de ambiente utilizadas neste trabalho (4GB de RAM, Intel Core i7 L640 2,13 GHz e Windows 7 Professional 64 bits 2009 SP1) são diferentes da usada pelo trabalho do melhor resultado da literatura (Intel Core 2,53 GHz e Linux Ubuntu kernel 2.6). Os tempos médios de execução, baseados nas 30 execuções para cada instância, são da ordem de dezenas de segundos, um valor baixo considerando a qualidade das soluções geradas.

Tabela 3. Parte da tabela utilizada para comparação entre BRKGA e BMA

	MELHOR FITNESS			FITNESS MÉDIO				MEDIANA (FITNESS)		TEMPO MÉDIO DE EXECUÇÃO
	BRKGA	BMA	%	BRKGA	BMA	%	DP	BRKGA	PV	BRKGA
nrp1-30	1204	1201	0,25	1198,5	1188,3	0,86	6,0	1200,0	0,61	3,61 s
nrp1-50	1836	1824	0,66	1821,2	1796,2	1,40	6,5	1820,0	0,53	3,09 s
nrp1-70	2507	2507	0,00	2507,0	2507,0	0,00	0,0	2507,0	-	2,38 s
nrp2-30	4842	4726	2,45	4763,7	4605,6	3,45	50,0	4759,5	0,07	19,07 s
nrp2-50	7979	7566	5,46	7885,3	7414,1	6,36	37,9	7878,5	0,02	15,73 s

6. Conclusões e trabalhos futuros

Pelos resultados mostrados, a meta-heurística BRKGA em conjunto com o método construtivo proposto mostram-se uma solução atraente para abordar o NRP. Também é recomendada uma ferramenta de configuração automática de parâmetros, como IRACE, de forma a sistematizar a escolha dos parâmetros da meta-heurística.

Como há suporte para processamento paralelo tanto no IRACE (pacote R "parallel") quanto no framework do BRKGA (OpenMP) e os processadores atuais mais simples possuem dois núcleos (*cores*) independentes, uma proposta para um trabalho futuro seria usar esse recurso para gerar soluções de melhor qualidade em um tempo menor de execução.

Referências

Andrade, C. E., Resende, M. G., Zhang, W., Sinha, R. K., Reichmann, K. C., Doverspike, R. D., & Miyazawa, F. K. (2015). A biased random-key genetic algorithm for wireless backhaul network design. Applied Soft Computing, 33, 150-169.

Bagnall, A. J., Rayward-Smith, V. J., & Whittley, I. M. (2001). The next release problem. Information and software technology, 43(14), 883-890.

Baker, P., Harman, M., Steinhofel, K., & Skaliotis, A. (2006, September). Search based approaches to component selection and prioritization for the next release problem. In Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on (pp. 176-185). IEEE.

Bergmeir, C., Hyndman, R. J., & Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. Computational Statistics & Data Analysis, 120, 70-83.

Birattari, M., & Kacprzyk, J. (2009). Tuning metaheuristics: a machine learning perspective (Vol. 197). Berlin: Springer.

Chaves, A. A., Lorena, L. A. N., Senne, E. L. F., & Resende, M. G. (2016). Hybrid method with CS and BRKGA applied to the minimization of tool switches problem. Computers & Operations Research, 67, 174-183.

Gonçalves, J. F., & Resende, M. G. (2011). Biased random-key genetic algorithms for combinatorial optimization. Journal of Heuristics, 17(5), 487-525.

Kohavi, R. (1995, August). A study of cross-validation and bootstrap for accuracy estimation and model selection. In Ijcai(Vol. 14, No. 2, pp. 1137-1145).

Lopes, M. C., de Andrade, C. E., de Queiroz, T. A., Resende, M. G., & Miyazawa, F. K. (2016). Heuristics for a hub location- routing problem. Networks, 68(1), 54-90.

López-Ibánez, M., Cáceres, L. P., Dubois-Lacoste, J., Stützle, T., & Birattari, M. (2016). The irace package: User guide. IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2016-004.

López-Ibánez, M., Dubois-Lacoste, J., Stützle, T., & Birattari, M. (2011). The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium.

Stefanello, F., Aggarwal, V., Buriol, L. S., Gonçalves, J. F., & Resende, M. G. (2015, July). A biased random-key genetic algorithm for placement of virtual machines across geo-separated data centers. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (pp. 919-926). ACM.

Toso, R. F., & Resende, M. G. (2015). A C++ application programming interface for biased random-key genetic algorithms. Optimization Methods and Software, 30(1), 81-93.

Xuan, J., Jiang, H., Ren, Z., & Luo, Z. (2012). Solving the large scale next release problem with a backbone-based multilevel algorithm. *IEEE Transactions on Software Engineering*, *38*(5), 1195-1212.

Comparing Low Level Heuristics Selection Methods in a Higher-Order Mutation Testing Approach

Jackson A. Prado Lima, Silvia R. Vergilio[1]

[1]DInf - Universidade Federal do Paraná (UFPR) – Curitiba, PR – Brasil

{japlima,silvia}@inf.ufpr.br

Abstract. *The selection of Higher Order Mutants (HOMs), which describe complex faults in the software testing, has been successfully performed by search-based algorithms, including a hyper-heuristic approach. Such an approach uses a selection method to choose among different Low-Level Heuristics (LLHs), which are search operators related to different Second Order Mutant (SOM) generation strategies. The idea is to select a reduced number of SOMs capable to capture subtler faults and replace the First Order Mutants. However different selection methods can be used and it is important to evaluate their performance. To contribute in this direction, this work presents results comparing the methods Choice Function (CF) and Fitness-Rate-Rank based Multi-Armed Bandit (FRR-MAB) concerning six Java programs and three fitness formulations. The results show that both selection methods are statistically equivalent, considering Hypervolume quality indicator. On the other hand, when we analyze the Error Ratio we observe that FRR-MAB provides better solutions than CF.*

Resumo. *A seleção de Mutantes de Ordem Superior, que descrevem defeitos complexos no teste de software, tem sido realizada com sucesso por algoritmos de busca, incluindo uma abordagem baseada em hiper-heurística. Tal abordagem usa um método de seleção para escolher entre diferentes hiper-heurísticas de baixo nível, que são operadores de busca, relacionados com estratégias de seleção de mutantes de segunda ordem. A ideia é selecionar um número reduzido desses mutantes capazes de capturar defeitos mais difíceis de serem revelados e que possam substituir os mutantes de primeira ordem. Entretanto, diferentes métodos de seleção podem ser usados e o desempenho destes precisa ser avaliado. Para contribuir nesta direção, este trabalho apresenta resultados comparando dois métodos: Choice Function (CF) e Fitness-Rate-Rank based Multi-Armed Bandit (FRR-MAB), considerando seis programas Java e três formulações para a função de avaliação. Os resultados de hypervolume mostram que ambos métodos são estatisticamente equivalentes, mas os resultados do FRR-MAB são melhores quando considerada a proporção do erro (Error Ratio, ER).*

1. Introduction

The Higher-Order Mutation Testing (HOMT) combines faults described by n different mutants to produce a Higher-Order Mutant (HOM) of order n ($n \geq 2$). The idea is to produce new strong mutants, more difficult to kill, but this strategy can also be used to reduce the number of mutants and equivalent ones [Kintis et al. 2012], since a test

case to kill the HOM is in general capable to kill its constituent First-Order Mutants (FOMs) [Harman et al. 2010].

HOMT is capable to reduce the test effort [Harman et al. 2010, Mateo et al. 2013] and simulate faults closer to real ones (complex fault). A complex fault is usually composed of different simple faults (modifications) that can be simulated only with the use of HOMs [Harman et al. 2010]. However, HOMT has a problem related to the HOMs generation, once that the number (space) of HOMs increases exponentially in relation to the number of FOMs. To solve such a problem, some works use search-based techniques [Harman et al. 2014, Langdon et al. 2010, Langdon et al. 2010, Omar et al. 2014], whilst other works propose conventional strategies, mainly based on Second-Order Mutants (SOMs) [Kintis et al. 2012, Madeyski et al. 2014, Mateo et al. 2013, Polo et al. 2009]. However, none of them has been proved to be better for all situations.

The performance of a strategy (or technique) depends on the domain and characteristics of the program, and mainly on the testing goals. Such factors are not considered by the strategies. The best one can be only determined empirically, by conducting experiments. To overcome this limitation, we introduced in a previous work [Prado Lima and Vergilio 2017] a Hyper-Heuristic (HH) and multi-objective approach based on NSGA-II to automatically select the best strategy, according to three objectives: i) the number of HOMs generated; ii) effectiveness concerning the capacity of the HOMs to capture subtler faults; and iii) capacity of the HOMs to replace their constituent FOMs.

A HH is an automated (high-level) methodology for generation or selection of Low-Level Heuristics (LLHs) to solve complex problems [Burke et al. 2013]. The HH operates on the search space of heuristics, while the conventional heuristics act directly on the search space of potential solutions to a particular problem [Burke et al. 2013]. An LLH can be a meta-heuristic or a search operator. A generation HH uses components of existing heuristics for generating new ones, whilst a selection HH chooses the LLHs based on its recent performance and uses a selection method to choose a LLH to be applied.

Spite of the promising results of our previous work [Prado Lima and Vergilio 2017], we evaluated only the *Choice Function* (CF) [Cowling et al. 2001] selection method. However, other methods can be explored, such *Fitness-Rate-Rank based Multi-Armed Bandit* (FRR-MAB) [Li et al. 2014], which has presented better performance in HH software engineering approaches [Guizzo et al. 2015, Strickler et al. 2016, Ferreira et al. 2017]. Motivated by these works, this paper presents results comparing both methods, CF and FRR-MAB, in our HH approach for HOMT. We adopted the same methodology and systems of previous work [Prado Lima and Vergilio 2017], and observed that the performance of our approach is better using FRR-MAB accoding Error Ratio (ER) values.

The paper is organized as follows. Section 2 describes the selection methods (CF and FRR-MAB). Section 3 reviews the proposed approach. Section 4 describes how the evaluation was conducted. Section 5 presents and analyzes the obtained results. Section 6 concludes the paper and presents directions for future research.

2. Hyper-Heuristic Selection Methods

In this work we evaluate two methods, which are chosen due to their performance in previous work [Guizzo et al. 2015, Strickler et al. 2016, Ferreira et al. 2017] on software engineering problems: CF, adapted by [Maashi et al. 2014], and FRR-MAB based on Upper Confidence Bound (UCB) [Auer et al. 2002].

The selection of an LLH is usually based on the information obtained by a reward, calculated by a credit assignment procedure. To evaluate the LLH improvements we adopted an adaptation proposed by [Guizzo et al. 2015], based on the concept of Pareto dominance as ϕ_1 and elapsed iterations as ϕ_2. This adaptation obtained promising results and is easy to use. The adopted credit assignment procedures are described next for both methods.

The credit assignment of CF [Maashi et al. 2014] contains only two functions: ϕ_1 and ϕ_2 given by Equation 1.

$$\phi(l) = a\phi_1(l) + b\phi_2(l) \tag{1}$$

where $\phi(l)$ gives a score for the LLH l; ϕ_1 reflects the recent improvement of l; ϕ_2 is the elapsed CPU seconds since l has been called; and a and b are the weight parameters to balance the values of ϕ_1 and ϕ_2.

The most noticeable Multi-Armed Bandit (MAB) method is UCB [Auer et al. 2002], which provides asymptotic optimality guarantees [Li et al. 2014]. Among the different versions of UCB algorithms proposed in the literature, FRR-MAB has good results focusing in the credit assignment task [Li et al. 2014]. The FRR-MAB aims to automatically select a particular operator op from a set K to be applied, through online learning, where the quality of an op is evaluated by the improvement of the fitness that it performed ($FRR_{i,t}$) [Li et al. 2014]. For this, a decaying factor (DF) is used to calculate the value of the final reward of each op in relation to the applications performed (n) in a given sliding window (SW). Subsequently, the next op to be selected is based on these reward values [Li et al. 2014]. In this way, the FRR-MAB selects the best op at each time point t according to Equation 2.

$$op_t = \underset{i=\{1...K\}}{\operatorname{argmax}} \left(FRR_{i,t} + C \times \sqrt{\frac{2 \times ln \sum_{j=1}^{K} n_{j,t}}{n_{i,t}}} \right) \tag{2}$$

where C is a scaling factor to control the trade-off between Exploitation and Exploration.

The authors of the FRR method introduce a rank-based method that uses a Fitness Improvement Rate (FIR) [Li et al. 2014]. We adopted an adaptation proposed in previous works [Strickler et al. 2016, Ferreira et al. 2017] to evaluate the efficiency of an operator according to the current evolutionary stage of the algorithm and, a SW with fixed size is used in order to only consider recent FIR information during credit assignment [Li et al. 2014].

3. The Hyper-heuristic HOMT approach

The main elements of our HH approach are in Figure 1. First, it is defined a program under test P and its test case set T. After, the approach receives as input a set F of FOMs generated from P by a mutation testing tool, and a metadata file is generated for each

$f \in F$. These files contain information about the mutant, such as, operator used, number of the original program line that suffered the mutation, and so on, as well as results from the mutant execution with all the test cases in T. Thus, it is possible to know what test cases kill each mutant, as well as other information such as if such mutant is anomalous or equivalent. On the other hand, additional settings are defined, such as the number of generations, fitness evaluations, population size, LLHs, selection method, and so on.

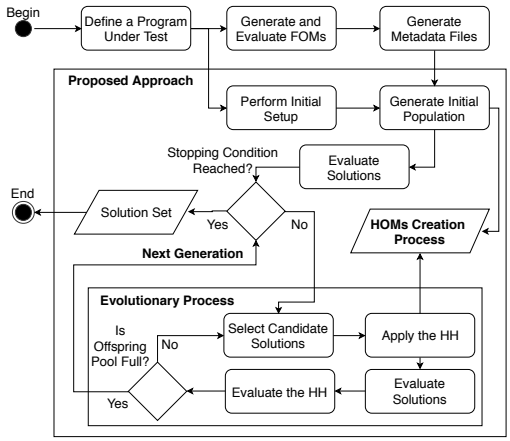


Figure 1. Elements of the Proposed Approach [Prado Lima and Vergilio 2017].

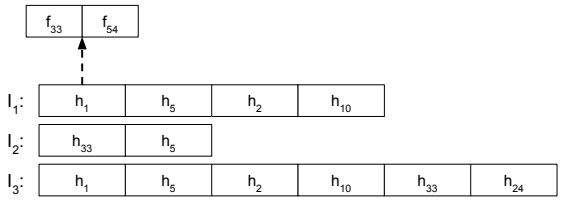


Figure 2. Representation of the population [Prado Lima and Vergilio 2017].

Each individual I in the population is a set of SOMs represented according to Figure 2. A solution to the problem is represented by an array of K individuals, where each position corresponds to a set of k SOMs. Each SOM is given by its constituent FOMs. In the figure, a population of $K = 3$ solutions (individuals) is represented. The first solution has $k = 4$ SOMs, the first one h_1 is composed by the FOMs f_{33} and f_{54}.

The initial population is evaluated (Section 3.1), and the evolutionary loop starts. Then, in each generation, two parents are selected by the selection operator to generate the new individuals. After this, an LLH (from LLHs defined in Section 3.2) is selected using the selection method chosen (CF or FRR-MAB) (Section 2). Then the operators corresponding to the selected LLH are applied and two children (offspring solutions) are generated. The children are evaluated and added to the offspring pool. These children are also used to assess the performance of the selected LLH. If the LLH generates good solutions, then it receives a good reward r, otherwise, a bad reward r. This LLH has its value of $time$ set to 0, and the other LLHs have their $time$ incremented by 1. If the offspring population is not full yet, then the process described in this paragraph is repeated until all the children of this generation are created.

With an offspring population full of solutions, NSGA-II follows its process and returns the set of solutions representing the best trade-off between the considered objectives.

3.1. Objectives

The first objective (Equation 3) is given by the number of SOMs k of the individual I, concerning the number of generated FOMs in the set F.

$$Size(I) = \frac{k}{|F|} \tag{3}$$

The second objective (Equation 4) is related to the capacity of the SOM to reveal subtler faults, that is, to reveal faults not revealed by its constituent FOMs. This function was adapted from other works [Omar et al. 2014, Harman et al. 2010].

$$WSD(I) = \sum_{i=1}^{k} \left(\frac{\frac{\mid (T_f \cup T_{h_i}) \mid - \mid (T_f \cap T_{h_i}) \mid}{\mid (T_f \cup T_{h_i}) \mid} + decoupled(h_i)}{2} \right) \tag{4}$$

$$decoupled(h) = \begin{cases} 1 & \text{if } |T_h| < |T_f|, |T_h \cap T_f| = \varnothing \text{ and } T_h \neq \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where h is the SOM composed by FOMs f_i and f_j; T is the set of available test cases; $T_{f_i}, T_{f_j} \subset T$ are sets of test cases that kill, respectively the FOMs f_i and f_j; $T_f = T_{f_i} \cup T_{f_j}$; and $T_h \subset T$ is the set of test cases that kill the SOM h. The equation evaluates the dissimilarity between T_h and the union of all test cases that kill its constituent FOMs, in order to evaluate the number of faults that can be detected by the SOM and by its constituent FOMs. The second term (Equation 5) serves to penalize equivalent mutants and those associated to T_h with a number of test cases greater than those of its constituent FOMs ($|T_h| > |T_f|$). Such SOMs can be easily killed.

The third objective (Equation 6) is related to the capacity of a SOM to replace its constituent FOMs, that is, the objective has as goal to maximize the number of HOMs so called Strongly Subsuming HOMs (SSHOMs) [Harman et al. 2010].

$$SSHOM(I) = \sum_{i=1}^{k} \left(\frac{\frac{\mid ((T_{f_i} \cap T_{f_j}) \cap T_{h_i}) \mid}{\mid ((T_{f_i} \cap T_{f_j}) \cup T_{h_i}) \mid} + intersection(h_i)}{2} \right) \tag{6}$$

$$intersection(h) = \begin{cases} 1 & \text{if } T_h \subset (T_{f_i} \cap T_{f_j}) \text{ and } T_h \neq \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The idea is to calculate the intersection ($T_{f_i} \cap T_{f_j}$) between the sets of test cases that kill the constituent FOMs f_i and f_j, and check if the test cases that kill the SOM also kill both FOMs ($T_h \subset (T_{f_i} \cap T_{f_j})$), in order to reduce the number of test cases. In addition to this, it is important that the SOM is non-equivalent ($T_h \neq \varnothing$ and $|T_h| < |T_f|$).

3.2. Search Operators and LLHs

The approach uses the Single Point Crossover that considers a variable size for the individual (according to the number k of SOMs in the solution). The mutation operators add, swap, or remove HOMs in the individuals. The first two operations generate the HOMs according traditional strategies existing in the literature [Polo et al. 2009, Mateo et al. 2013]. These strategies are described in Table 1. They are: FirstToLast, RandomMix, Different-Operators (DiffOp), and Each-Choice. They choose a pair of mutants considering the list of all FOMs, repeating the process until all the mutants in the list are used. The main difference between them is how they select the pair of non-used mutants to generate a SOM. When a SOM is generated and stored, the constituent FOMs

are marked as used. The LLHs are composed of a combination of 0 (λ) or 1 mutation operators and 0 (λ) or 1 crossover operator, according to Table 2. To generate the initial population, the user can choose one of the available strategies.

Table 1. Higher Order Mutation Strategies.

Strategy	Description
FirstToLast	It intends to prioritize the selection of unused mutants at the extremities of the mutant list (that is the top and bottom of the list). Procedure: 1) Selects the first unused mutant from the list; 2) Selects the last unused mutant from the list and generates the SOM by combining both selected mutants;
Random Mix	It randomly selects unused mutants from the list of all mutants. Procedure: 1) Randomly selects an unused mutant from the list; 2) Randomly selects an unused mutant from the list and generates the SOM by combining both selected mutants[1];
DiffOp	It aims at selecting unused mutants produced by different operators. Procedure: 1) Selects the first unused mutant from the list; 2) Selects the next unused mutant which was generated by a different operator from the already selected mutant, and generates the SOM by combining both selected mutants;
Each-Choice	It aims at sequentially selecting unused mutants. Procedure: 1) Selects the first unused mutant from the list; 2) Selects the next unused mutant from the list and generates the SOM by combining both selected mutants;

Table 2. Low-Level Heuristic Composition.

Mutation	Crossover	LLH
λ	Single Point	h1
AddDiffOp	Single Point	h2
SwapDiffOp	Single Point	h3
AddEach-Choice	Single Point	h4
SwapEach-Choice	Single Point	h5
AddFirstToLast	Single Point	h6
SwapFirstToLast	Single Point	h7
AddRandomMixLimited	Single Point	h8
SwapRandomMixLimited	Single Point	h9
Remove	Single Point	h10
AddDiffOp	λ	h11
SwapDiffOp	λ	h12
AddEach-Choice	λ	h13
SwapEach-Choice	λ	h14
AddFirstToLast	λ	h15
SwapFirstToLast	λ	h16
AddRandomMixLimited	λ	h17
SwapRandomMixLimited	λ	h18
Remove	λ	h19
λ	λ	h20

4. Experiment Setup

Our evaluation is guided by the following research question: "How does the FRR-MAB method compare against the CF method?". To this end we implemented two versions of NSGA-II [Deb et al. 2002] using each selection method defined, CF and FRR-MAB and also considered the performance regarding three fitness formulations defined in relation to the solution set size produced by the approach:

- 2OSSHOM: the fitness function is composed by the objectives 1 and 3;
- 2OWSD: the fitness function is composed by the objectives 1 and 2;
- 3O: the fitness function is composed by the objectives 1, 2, and 3.

4.1. Systems Used

Similarly to our previous work [Prado Lima and Vergilio 2017], we used six Java systems. To each program, a set of test cases is available [2]. We generated the FOMs using MuJava [Ma et al. 2005] and its all operators available. The order of the FOMs in the list used to implement the strategies is given by the generation order of such a tool. After, we executed all the available test cases to build a killing relation matrix (containing which test case kills which mutant). Then, we analyzed the remaining mutants and if necessary new

[1] We adapted Random Mix due to a possible exponential explosion and to allow scalability. Initially, the FOMs are selected randomly, but when the number of randomly generated SOMs grows and is greater than the number of FOMs, the next HOM to be chosen is randomly selected from the set of SOMs already generated.

[2] The systems and test cases used are available at `http://bit.do/SUTs4MutationTesting`.

test cases were generated. In such step, the equivalent mutants were manually identified. The set F was then composed by all non-equivalent mutants, and T by all test cases.

Table 3 presents, for each system, the number of lines of code (LOC), the size of T, the number of generated mutants ($|GM|$), the number of FOMs ($|F|$) that could be killed, and the number of equivalent mutants ($|E|$).

4.2. Configuring the Algorithms

For NSGA-II-CF, we used the same parameters (Table 5) of our previous work [Prado Lima and Vergilio 2017]. Then we used these common parameters to configure FRR-MAB, in order to guarantee the same opportunity for both algorithms in the evolution process. To configure we used irace [López-Ibánez et al. 2016] as tuner and Hypervolume (HV) [Zitzler and Thiele 1998] as a quality measure. The best configurations found are presented in Table 4 that presents the domain used in the tuning, and the parameters specific for each algorithm. The parameters a and b are used in CF, whilst C, DF, and SW are used in FRR-MAB. We set a budget of 10.000 configurations with 2.000 fitness evaluations for each tuning. With such parameters, the algorithms were run 30 times on each system, with 60.000 fitness evaluations.

Table 3. Information about the Systems.

System	LOC	$\|T\|$	$\|GM\|$	$\|F\|$	$\|E\|$
bisect	35	34	198	174	24
bub	50	261	172	144	28
find	62	144	299	277	22
fourballs	42	116	292	280	12
mid	57	136	317	257	60
triangle	88	259	605	526	79

Table 4. Parameters found after tunning.

	CF		FRR-MAB		
Parameter	α	β	C	DF	SW
Domain	[0, 1]	[0, 1]	[0, 1]	[0, 1]	[50, 300]
2OSSHOM	0.04093	0.79557	0.03538	0.15738	76
2OWSD	0.08626	0.8639	0.46707	0.90911	50
3O	0.68380	0.25971	0.18814	0.42964	68

Table 5. Common parameters for CF and FRR-MAB.

Parameter	Domain	2OSSHOM	2OWSD	3O
$\|Population\|$	[50, 300]	62	62	50
$\|Hom\ Set\|$	[0.1, 0.5]	0.48	0.39	0.47
$Initialization$	DiffOp, Each-Choice, FirstToLast, RandomMix	RandomMix		
$Selection$	KTournament, NaryTournament, Random, Binary	Random	Nary	KTournament
$\|Tournament\|$	[2, 4, 6, 8, 10]	2		

5. Results and Analyses

In our analysis we defined three solution sets to analyze the performance of the algorithms: (i) PF_{approx}, non-dominated solution set obtained from one algorithm execution; (ii) PF_{known}, non-dominated and non-repeated solution set obtained from the union of all PF_{approx} (runs) from a algorithm; (iii) PF_{true} : non-dominated and non-repeated solution set obtained from the union of all PF_{known} from k algorithms, and this one represents an approximation of True Pareto Front, which in our case is unknown.

We evaluated the algorithms performance using two quality indicators, average HV and Error Ratio (ER). ER corresponds to the proportion of the elements from the set PF_{known} that are present in the set PF_{true} [Van Veldhuizen 1999]. The average was computed considering the 30 fronts found by each algorithm, for each system. We compared the results, for each formulation, using Friedman [Friedman 1940] to evaluate the performance in relation to all systems, and Kruskal-Wallis [Kruskal and Wallis 1952] to

evaluate de performance in relation to each system. The statistical tests were applied with 95% of confidence [Arcuri and Briand 2014]. Table 6 shows the obtained HV mean values and standard deviation in parenthesis. The values highlighted in bold are the best ones. A cell in gray represents that there is statistical difference.

We observe that CF has the best performance for the 2OSSHOM formulation, whilst FRR-MAB has the best performance for the other ones. Although FRR-MAB presents the best values in most of the cases (67%), the statistical test shows that there is not difference between the algorithms for all formulations and systems, except for `triangle` (2OSSHOM) where there is difference and CF was the best.

Table 6. Hypervolume averages and standard deviation.

System	2OSSHOM		2OWSD		3O	
	CF	**FRR-MAB**	**CF**	**FRR-MAB**	**CF**	**FRR-MAB**
`bisect`	0.4690 (0.0118)	**0.4729 (0.0136)**	0.5037 (0.0147)	**0.5067 (0.0097)**	**0.4489 (0.0207)**	0.4466 (0.0253)
`bub`	0.4978 (0.0029)	**0.4978 (0.0025)**	0.5098 (0.0023)	**0.5099 (0.0020)**	0.4244 (0.0201)	**0.4258 (0.0212)**
`find`	**0.5081 (0.0119)**	0.5035 (0.0143)	0.5726 (0.0113)	**0.5743 (0.0111)**	0.3605 (0.0229)	**0.3687 (0.0215)**
`fourballs`	**0.4752 (0.0058)**	0.4741 (0.0064)	0.5573 (0.0129)	**0.5584 (0.0115)**	0.4297 (0.0299)	**0.4337 (0.0230)**
`mid`	**0.4157 (0.0332)**	0.4157 (0.0338)	0.6104 (0.0504)	**0.6149 (0.0468)**	0.4588 (0.0387)	**0.4712 (0.0332)**
`triangle`	**0.4639 (0.0083)**	0.4595 (0.0101)	**0.6034 (0.0096)**	0.5985 (0.0134)	0.4398 (0.0280)	**0.4404 (0.0251)**
	p-value	*chi-squared*	*p-value*	*chi-squared*	*p-value*	*chi-squared*
Friedman	0.4142	0.66667	0.1025	2.6667	0.1025	2.6667

After, it was analyzed the solution set obtained for each algorithm in each fitness formulation. Table 7 shows the number of solutions obtained in PF_{true}, the number of solution from PF_{known} obtained for each algorithm, the number of solutions from PF_{known} that are in PF_{true} in parenthesis, and the ER value. The values highlighted in bold are the best ones. Analyzing the results we observed that FRR-MAB outperformed CF, in relation to: (i) ER values were FRR-MAB was the best in 15 out 18 cases (83%) and obtained six times the best value (0.0) whilst CF obtained the best values in only five times; (ii) Regarding solutions from PF_{known} in PF_{true}, FRR-MAB was the best in ≈78% (14) of the cases. Thus, we observed that the results presented in Table 7 show that FRR-MAB outperforms CF, both in the number and quality of the solutions generated. Finally, it is observed that the number of solutions present in the PF_{true} increases when one more objective is added (formulation 3O). The great the difficulty in the search process the great the solution diversity.

Table 7. Results from the Pareto Fronts

System	Formulation	PF_{true}	CF		FRR-MAB	
			PF_{known}	ER	PF_{known}	ER
`bisect`	2OSSHOM	33	32 (32)	**0.0**	**33 (33)**	**0.0**
	2OWSD	154	**150 (149)**	**0.0067**	146 (141)	0.0342
	3O	366	**294 (207)**	**0.2959**	292 (175)	0.4007
`bub`	2OSSHOM	24	**24 (24)**	**0.0**	**24 (24)**	**0.0**
	2OWSD	155	**157** (149)	0.051	153 **(152)**	**0.0065**
	3O	499	385 (256)	0.3351	**389 (273)**	**0.2982**
`find`	2OSSHOM	60	59 (58)	0.0169	**60 (60)**	**0.0**
	2OWSD	226	210 (168)	0.2	**217 (205)**	**0.0553**
	3O	456	295 (194)	0.3424	**349 (277)**	**0.2063**
`fourballs`	2OSSHOM	27	**27 (27)**	**0.0**	**27 (27)**	**0.0**
	2OWSD	219	**210** (183)	0.1286	205 **(194)**	**0.0537**
	3O	321	248 (158)	**0.3629**	**287 (176)**	0.3868
`mid`	2OSSHOM	11	**11 (11)**	**0.0**	10 (10)	**0.0**
	2OWSD	182	**177** (159)	0.1017	173 **(167)**	**0.0347**
	3O	233	**208** (105)	0.4952	181 **(142)**	**0.2155**
`triangle`	2OSSHOM	18	**18 (18)**	**0.0**	17 (17)	**0.0**
	2OWSD	178	**172** (83)	0.5174	156 **(114)**	**0.2692**
	3O	473	325 (213)	0.3446	**367 (272)**	**0.2589**

Then we can conclude that our approach using FRR-MAB selection method provides better solutions and captures properly the objectives considering different formulations. However, considering HV both selection methods are statistically equivalent.

6. Concluding Remarks

This paper compares two LLH selection methods, CF and FRR-MAB, used to select the best SOM generation strategy for an HH-based approach. In the approach an LLH is a search-operator related to conventional strategies for SOM generation. The solutions are evaluated considering different objectives such as: i) number of SOMs generated; ii) capacity of the SOMs to capture subtler faults; and iii) capacity of the SOMs to replace their constituent FOMs, which can lead to a reduction in the number of used mutants. The approach was implemented with NSGA-II

The selection methods present statistically equivalent results considering Hypervolume. Being that, FRR-MAB present the best HV means in most cases (67%). In relation to the solutions quality and Error Ratio, the FRR-MAB outperformed CF, by presenting the best value in 83% of the cases.

As our approach is generic and allows different configurations, algorithms, objectives, selection methods, and LLHs, we intend to conduct future evaluations with other selection methods besides CF and FRR-MAB, such as, a learning automata, roulette-based heuristic selection, other selection methods based on MAB reinforcement learning, and other algorithms such as MOMBI-II and MOEAD/D. Furthermore, we intend to explore higher mutation orders and different strategies to generate HOMs.

References

Arcuri, A. and Briand, L. (2014). A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R. (2013). Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695.

Cowling, P., Kendall, G., and Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. In Burke, E. and Erben, W., editors, *3rd Practice and Theory of Automated Timetabling (PATAT'00)*, volume 2079 of *Lecture Notes in Computer Science*, pages 176–190. Springer.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6(2):182–197.

Ferreira, T. N., Prado Lima, J. A., Strickler, A., Kuk, J. N., Vergilio, S. R., and Pozo, A. (2017). Hyper-heuristic Based Product Selection for Software Product Line Testing. *IEEE Computational Intelligence Magazine*, 12(2):34–45.

Friedman, M. (1940). A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86–92.

Guizzo, G., Fritsche, G. M., Vergilio, S. R., and Pozo, A. T. R. (2015). A hyper-heuristic for the multi-objective integration and test order problem. In *Genetic and Evolutionary Computation Conference*, GECCO, pages 1343–1350.

Harman, M., Jia, Y., and Langdon, W. B. (2010). A manifesto for higher order mutation testing. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 80–89.

Harman, M., Jia, Y., Mateo, P. R., and Polo, M. (2014). Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *International conference on Automated Software Engineering*, pages 397–408.

Kintis, M., Papadakis, M., and Malevris, N. (2012). Isolating first order equivalent mutants via second order mutation. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), 2012*, pages 701–710. IEEE.

Kruskal, W. H. and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260):583–621.

Langdon, W. B., Harman, M., and Jia, Y. (2010). Efficient multi-objective higher order mutation testing with genetic programming. *Journal of systems and Software*, 83(12):2416–2430.

Li, K., Fialho, A., Kwong, S., and Zhang, Q. (2014). Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. on Evolutionary Computation*, 18(1):114–130.

López-Ibánez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M., and Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Ma, Y., Offutt, J., and Kwon, Y. (2005). MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133.

Maashi, M., Özcan, E., and Kendall, G. (2014). A multi-objective hyper-heuristic based on choice function. *Expert Systems with Applications*, 41(9):4475–4493.

Madeyski, L., Orzeszyna, W., Torkar, R., and Józala, M. (2014). Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. on Software Engineering*, 40(1):23–42.

Mateo, R. P., Macario, P. U., Aleman, F., and Luis, J. (2013). Validating second-order mutation at system level. *IEEE Trans. on Software Engineering*, 39(4):570–587.

Omar, E., Ghosh, S., and Whitley, D. (2014). Comparing search techniques for finding subtle higher order mutants. In *Genetic and evolutionary computation Conference*, GECCO, pages 1271–1278.

Polo, M., Piattini, M., and García-Rodríguez, I. (2009). Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131.

Prado Lima, J. A. and Vergilio, S. R. (2017). A multi-objective optimization approach for selection of second order mutant generation strategies. In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, SAST, pages 6:1–6:10, New York, NY, USA. ACM.

Strickler, A., Prado Lima, J. A., Vergilio, S. R., and Pozo, A. T. R. (2016). Deriving products for variability test of feature models with a hyper-heuristic approach. *Applied Soft Computing*, 49:1232–1242.

Van Veldhuizen, D. A. (1999). Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. Technical report, DTIC Document.

Zitzler, E. and Thiele, L. (1998). Multiobjective optimization using evolutionary algorithms — a comparative case study. In *Parallel Problem Solving from Nature*, PPSN V, pages 292–301. Springer.

Estratégias de priorização de características para projeto de arquitetura de LPS baseado em busca

Diego Fernandes da Silva, Matheus Alberto Vaz Cecílio, Thelma E. Colanzi

Departamento de Informática – Universidade Estadual de Maringá (UEM)
CEP 87.020–900 – Maringá – PR – Brasil

{dizzusmon, maatheusaalberto}@gmail.com, thelma@din.uem.br

Resumo. *Um desafio de projeto de Arquitetura de Linha de Produto (PLA) é a maximização da sua modularidade, uma vez que a PLA inclui todos os elementos arquiteturais que viabilizam o reúso de artefatos para a derivação dos produtos de uma Linha de Produto de Software (LPS). A abordagem MOA4PLA permite otimizar projetos de PLA utilizando algoritmos de busca. Uma das otimizações modulariza características utilizando a mesma taxa de probabilidade de modularização para todas as características da LPS. Todavia, é frequente a existência de características prioritárias, as quais permitem a derivação de determinados produtos logo no início do ciclo de vida da LPS. Diante disso, o objetivo deste trabalho é propor três estratégias para permitir que o arquiteto priorize a modularização de determinadas características na MOA4PLA. Após a implementação das três estratégias, um estudo de caso foi realizado utilizando duas PLAs e os resultados evidenciaram efetiva modularização das características priorizadas.*

Abstract. *A challenge of the Product Line Architecture (PLA) design is its modularity maximization, since the PLA includes all architectural elements that provide the artifacts reuse for the derivation of products of the Software Product Line (SPL). The MOA4PLA approach allows to optimize PLA designs using search algorithms. One of its optimizations modularizes SPL features using equal probability rate to every SPL feature. Nevertheless, it is common to have priority features that allow the derivation of certain products at the beginning of the SPL life cycle. In this context, the objective of this work is to propose three strategies to prioritize the modularization of features in MOA4PLA. The three strategies were implemented and a case study using two PLAs was carried out. Results evidenced effective modularization of the prioritized features.*

1. Introdução

Linha de Produto de Software (LPS) é uma abordagem para desenvolvimento de produtos de software por meio do reúso de artefatos, reduzindo o tempo e custo de elaboração de um produto [Pohl, Bockle e Linden, 2005]. Um dos principais artefatos de uma LPS é sua arquitetura, a *Product Line Architecture* (PLA). A PLA abrange o projeto comum a todos os produtos derivados da LPS, pois ela descreve uma arquitetura genérica que concede uma solução para a gama de produtos da LPS [Contieri Junior *et al*. 2011]. Cada produto da LPS é formado por características (*features*), que podem ser obrigatórias, opcionais ou alternativas. Uma característica obrigatória está presente em todos os produtos, as opcionais estão em apenas alguns produtos e as alternativas formam um grupo de características relacionadas, a partir do qual um subgrupo de características está presente em cada produto [Pohl, Bockle e Linden, 2005].

Na engenharia de LPS, é costumeiro possuir uma lista das características envolvidas nos produtos cuja derivação deve ocorrer logo no início do ciclo de vida da LPS, as quais deveriam ter prioridade de modularização. Além disso, a modularização de características opcionais e alternativas costuma ser mais relevante do que a modularização de características obrigatórias, uma vez que estas impactam em menor grau no reúso e extensibilidade da PLA já que são comuns a todos os produtos da LPS.

O projeto de PLA é influenciado por diversos fatores, tais como modularização de características, difusão de funcionalidades, reúso e extensibilidade da PLA, etc. Alguns deles podem ser conflitantes levando a diversas possibilidades de projeto para uma determinada LPS. Métricas relacionadas a esses fatores podem ser usadas como objetivos a serem otimizados por algoritmos de busca a fim de obter um conjunto de alternativas de projeto de PLA considerando o compromisso entre os fatores otimizados.

Nesse sentido, a abordagem MOA4PLA (*Multi-Objective Approach for Product-Line Architecture Design*) [Colanzi *et al*., 2014] visa a otimização de projetos de PLA utilizando algoritmos de busca. Essa abordagem produz um conjunto com as melhores soluções considerando os objetivos otimizados, tais como modularização de características, extensibilidade e variabilidade da PLA, coesão, acoplamento, etc. O arquiteto de LPS deve então escolher uma das alternativas de projeto (soluções) retornadas pela abordagem para adotar como a PLA da LPS. A MOA4PLA contém um operador de busca que visa a melhoria da modularização de características (*Feature-driven Operator*). Esse operador seleciona aleatoriamente quais características serão modularizadas, utilizando a mesma taxa de probabilidade para todas as características.

A motivação para o presente trabalho vem da possibilidade de aprimorar o *Feature-driven Operator* permitindo que o arquiteto defina quais características devem ser priorizadas pelo referido operador de busca. Desse modo, o arquiteto tem a possibilidade de interagir com a abordagem não somente ao final do processo de otimização, quando escolhe uma das soluções, mas também no início do processo ao definir a ordem de prioridade entre as características.

Após um mapeamento sistemático da literatura, detectou-se a inexistência de trabalhos referentes a estratégias de priorização para modularizar características de LPS. Os trabalhos encontrados focam, por exemplo, em priorizar características de LPS a serem implementadas em cada iteração de um método ágil [Díaz *et al*. 2011] e em priorizar características visando maximizar o reúso de componentes de software [Efat *et al*. 2014]; no entanto, as métricas de reúso deste último trabalho são incompatíveis com a abordagem MOA4PLA e não estão relacionadas à Engenharia de LPS.

Nesse contexto, três estratégias de priorização de características a serem modularizadas foram propostas e implementadas na OPLA-Tool [Féderle *et al.* 2015] (ferramenta que automatiza a abordagem MOA4PLA). Essas estratégias foram nomeadas como: Manual, Automática e Mista. Subsequentemente, cada estratégia foi devidamente avaliada em um estudo de caso envolvendo duas PLAs e os resultados evidenciaram a efetividade das estratégias propostas.

2. Projeto de LPS baseado em busca

A abordagem MOA4PLA (*Multi-Objective Approach for Product-Line Architecture Design*) é uma abordagem sistemática e automatizada que usa algoritmos de busca multiobjetivos para avaliar e melhorar o projeto de PLA [Colanzi *et al*. 2014]. OPLA-Tool [Féderle *et al*. 2015] é a ferramenta que automatiza a aplicação da MOA4PLA. Ela

permite o uso de diferentes algoritmos de busca multiobjetivos para otimizar projeto de PLA, tais como o NSGA-II [Deb *et al*. 2002].

A partir de um projeto de PLA original, modelado em um diagrama de classes UML, operadores de busca específicos para projeto de PLA são aplicados para obter soluções descendentes. Cada solução obtida no processo de busca é avaliada por funções objetivo que fazem parte do modelo de avaliação da MOA4PLA. Ao final do processo, várias alternativas de projeto, contendo os melhores *trade-off* entre os objetivos otimizados, são obtidas automaticamente de acordo com os objetivos selecionados para otimização. Desse modo, o arquiteto pode selecionar uma das alternativas para adotar como PLA para uma dada LPS sendo poupado do trabalho de criar e avaliar as diferentes possibilidades de projeto para o domínio em questão.

O modelo de avaliação da MOA4PLA [Verdecia *et al*. 2017] contém um conjunto de funções objetivo baseadas em métricas de software, as quais fornecem indicadores sobre várias propriedades arquiteturais, tais como modularização de características, extensibilidade de LPS, variabilidade, acoplamento, coesão e tamanho. O arquiteto deve escolher um subconjunto de funções objetivo relacionadas às propriedades que ele deseja otimizar. As duas funções objetivo utilizadas no presente trabalho são FM (*Feature-driven Metrics*) e CM (*Conventional Metrics*). A função objetivo FM avalia a modularização de características da LPS em termos de difusão e entrelaçamento de características nos elementos arquiteturais e de coesão baseada em característica. A função CM fornece indicadores sobre propriedades arquiteturais convencionais, como coesão e acoplamento. As equações das funções objetivo e definições das métricas contidas nas funções são apresentadas em [Cecílio 2017] e em [Verdecia *et al*. 2017].

A MOA4PLA inclui seis operadores de mutação específicos para projeto de PLA: *Move Method, Move Attribute, Add Class, Move Operation, Add Package Manager, Feature-driven Operator* [Colanzi *et al*. 2014]. Este último tem como objetivo modularizar uma característica que esteja entrelaçada com outra(s) em um componente selecionado aleatoriamente. Ao selecionar o componente, o operador verifica se há entrelaçamento de características nas interfaces e classes do componente selecionado. Caso exista, uma característica entrelaçada é selecionada aleatoriamente para ser modularizada em um novo componente. Todos os elementos arquiteturais associados à característica sendo modularizada são movidos para o novo componente. O pseudocódigo desse operador é apresentado na próxima seção.

3. Estratégias de Priorização de Características

Como mencionado introdutoriamente, três estratégias de priorização de características foram implementadas: (i) estratégia manual, (ii) estratégia automática e (iii) estratégia mista. As estratégias são apresentadas nas próximas subseções e impactam diretamente na lista de características a serem modularizadas pelo *Feature-driven Operator*.

Além das estratégias propostas, outra contribuição deste trabalho é a adaptação do *Feature-driven Operator* para receber a lista de características priorizadas pelo arquiteto. A Figura 1 apresenta o pseudocódigo desse operador, onde A é a PLA fornecida como entrada. Os trechos em negrito consistem das modificações efetuadas para a inclusão da lista de características priorizadas. Primeiramente, coleta-se o conjunto dos componentes da PLA A, o conjunto de elementos arquiteturais de A e um componente (c_x) escolhido aleatoriamente a partir do conjunto de componentes (linhas 2, 3 e 4, respectivamente). Em seguida coletam-se todas as características do

componente c_x (linha 5) e a lista de características priorizadas (Fcp - linha 6). Caso haja mais que uma característica associada a c_x (linha 7), o próximo passo é encontrar, dentre as características associadas a c_x, qual delas é prioritária em Fcp (linhas 9 a 14). Caso não haja característica de c_x priorizada em Fcp, uma característica de c_x é selecionada aleatoriamente (linha 15). A característica fp será modularizada no componente cz, selecionado de acordo com o número de componentes de *A* associados à característica fp (linhas 17 a 24). Quando esse número é 0, um novo componente cz é criado. Quando há somente um componente associado a fp, ele mesmo é o alvo da modularização de fp e quando o número é maior que 1, um componente é selecionado aleatoriamente. Os elementos arquiteturais associados exclusivamente a fp são movidos para cz (linha 27) e um novo relacionamento é estabelecido entre cz e o componente original de cada elemento arquitetural movido (linha 28).

```
1    begin{
2    C = A.getAllComponents()
3    E = set of architectural elements of A
4    cx = a randomly selected component from C
5    Fc = cx.getAllFeatures()
6    Fcp = cx.getPrioritizedFeatures()
7    if Fc.size()>1 then{
8        fp = null
9        for each feature in Fcp {
10           if feature ∈ Fc then{
11                 fp = feature
12                 exit()
13           }
14       }
15       if fp == null then fp = a randomly selected feature from Fc
16       cf = A.getAllComponentsAssociatedWithFeature(fp);
17       if cf.size == 0 then{
18             cz = A.createComponent()
19             cz.addFeature(fp)
20       }
21       else{
22            if cf.size == 1 then cz = cf.get(0)
23            else cz = a randomly selected component from cf
24      }
25      for each e ∈ E | (e ≠ cz or e ∉ cz){
26          if e.getFeatures.size()==1 and  e.isAssociatedWithFeature(fp) then{
27                move e to cz
28                A.addRelationship(cz,original component of e)
29          }
30   return A
31   }
```

Figura 1. Pseudocódigo do *Feature-driven Operator* adaptado para as estratégias de priorização de características.

3.1 Estratégia Manual

Na estratégia manual, o arquiteto tem a liberdade de organizar a lista de características a serem modularizadas da PLA da maneira como preferir. Para tal, é criada uma lista para armazenar as características priorizadas. O arquiteto manipula a lista, concedendo as prioridades de cada característica. O pseudocódigo da estratégia manual é apresentado na Figura 2. A escolha do arquiteto é armazenada na lista manualList na linha 3 do pseudocódigo, a qual é retornada pela função e utilizada no *Feature-driven Operator*.

```
1  function manualApproach(){
2  List<Feature> manualList;
3     for each Feature of the LPS, add Feature in manualList following the order selected
   by the architect;
4     save manualList;
5     return manualList;
6  }
```

Figura 2. Pseudocódigo para a Estratégia Manual.

3.2 Estratégia Automática

Na estratégia automática a priorização se baseia no tipo das características, que são obrigatória, alternativa e opcional. A ordem de prioridade entre características estabelecida na estratégia automática é: opcional, alternativa e por último, obrigatória. A ordem de priorização entre características do mesmo tipo é definida aleatoriamente.

O fato das características opcionais e alternativas poderem estar espalhadas pela PLA torna-as mais relevantes para a modularização devido à necessidade de acoplá-las aos componentes da arquitetura do produto durante a geração ou evolução de um produto específico da LPS. As características opcionais são prioritárias em relação às alternativas porque uma característica opcional pode não aparecer em um certo produto, diferentemente das alternativas, que consistem em um grupo, a partir do qual pelo menos uma deverá ser inserida. Portanto, quanto maior a modularização das características opcionais, mais facilmente elas poderão ser acopladas aos produtos derivados da LPS. As características obrigatórias tem menor prioridade que as outras, já que são características comuns a todos os produtos da LPS, não impactando tanto no reúso da PLA, uma vez que todos os produtos derivados da LPS devem conter os elementos arquiteturais referentes às características obrigatórias.

A Figura 3 apresenta o pseudocódigo dessa estratégia. A ordenação das características de acordo com seu tipo é realizada na linha 4. Essa lista é retornada pela função e utilizada no *Feature-driven Operator* como sendo a lista `Fcp` (Figura 1).

```
1  function automaticApproach(){
2  List<Feature> autoList;
3     for each Feature of the LPS, add Feature in autoList;
4     sort autoList by feature type (optional, alternative and mandatory);
5     save autoList;
6     return autoList;
   }
```

Figura 3. Pseudocódigo para a Estratégia Automática.

3.3 Estratégia Mista

A estratégia mista é uma composição das estratégias manual e automática, pois ela permite que o arquiteto escolha quais características quer priorizar e depois complementa a lista com as características ordenadas por tipo. Seu pseudocódigo é apresentado na Figura 4. A lista `autoList` contém todas as características da PLA (linha 4). O arquiteto deve inserir as características cuja modularização deseja priorizar na lista `mixedList` (linha 5) e à medida em que uma característica é inserida em `mixedList`, ela é excluída de `autoList` (linha 6). Depois, `autoList` é ordenada de acordo com o tipo de característica e inserida no final de `mixedList` (linhas 7 e 8). Como saída, é fornecida uma lista ordenada, primeiramente, com as características priorizadas pelo arquiteto e por fim, com as demais características ordenadas automaticamente por tipo (linhas 10-12). Essa lista é retornada pela função e utilizada no *Feature-driven Operator* como sendo a lista `Fcp` (Figura 1).

3.4 Aspectos de Implementação e Uso das Estratégias de Priorização

A implementação das três estratégias propostas implicaram em algumas modificações da interface gráfica da OPLA-Tool [Féderle et al. 2015]. Devido ao fato da estratégia mista também utilizar uma lista de características associadas a um tipo, foi pertinente tratar tal estratégia na mesma interface da estratégia automática, somente inativando comandos de acordo com a escolha do arquiteto. A estratégia manual foi implementada em uma interface diferente das outras estratégias já que precisa de menos informações.

```
1     function mixedApproach(){
2     List<Feature> autoList;
3     List<Feature> mixedList;
4       autoList <- getAllFeatures();
5       while there is Feature to be prioritized {
6             add Feature f in mixedList in the order selected by the architect;
7             remove f from autoList;
        }
8       sort autoList by feature type (optional, alternative and mandatory);
        append autoList to mixedList;
9       save mixedList;
10      return mixedList;
11    }
```

Figura 4. Pseudocódigo para a Estratégia Mista.

Para organizar tal separação de interfaces, foi adicionada uma caixa de seleção na aba de configuração de execução da OPLA-Tool (Figura 5) com as opções *Manual* e *Auto/Mixed* (canto inferior esquerdo da figura) já que as estratégias automática e mista compartilham a mesma interface. O botão *Feature Prioritization* foi adicionado ao lado da caixa de seleção para direcionar o arquiteto para a aba de priorização das características, de acordo com a estratégia selecionada. Duas novas abas foram criadas: *Automatic/Mixed Feature Prioritization,* responsável pelas estratégias automática e mista, e *Manual Feature Prioritization*, responsável pela estratégia manual.

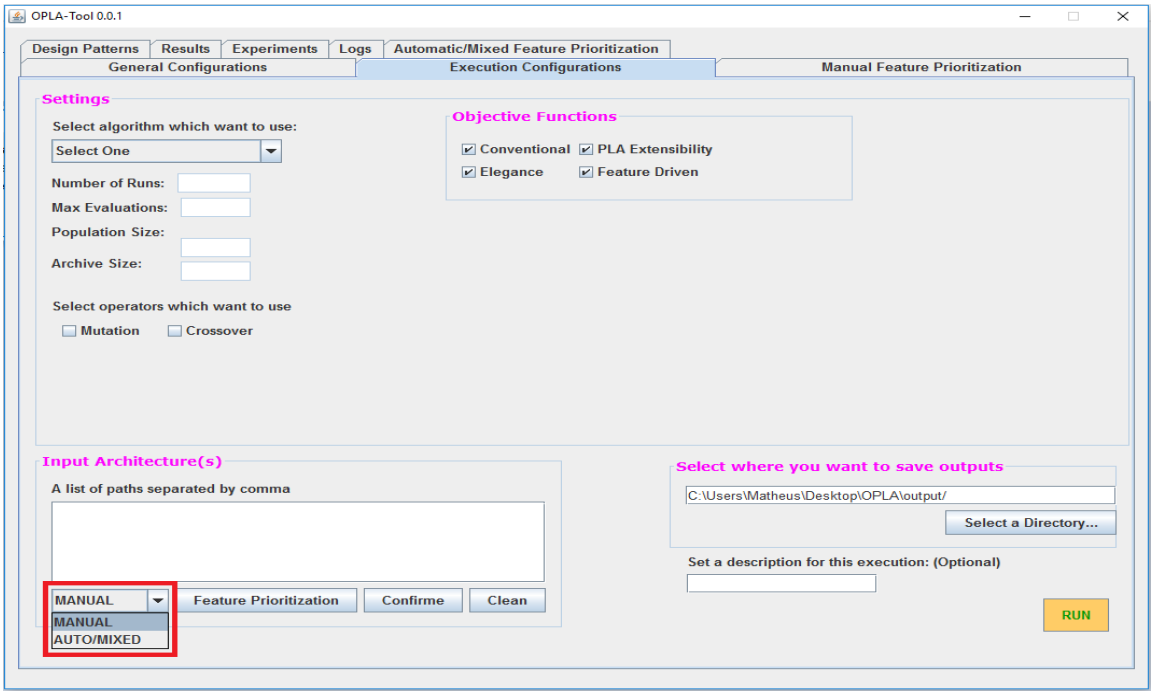


Figura 5. Caixa de seleção para escolha da estratégia a ser utilizada.

Como pode ser observado na Figura 6, a interface gráfica das estratégias automática e mista contém uma tabela com as PLAs disponíveis, uma tabela com a lista das características da PLA selecionada (*Architecture Features*) e outra tabela para as características a serem priorizadas tendo como base o tipo (*Automatic Prioritized Features*). O texto em português que aparece na Figura 6 é meramente explicativo e não faz parte da interface. Para a definição de qual estratégia será usada, as opções *Automatic* e *Mixed* estão disponíveis na seção *Choose Approach* no centro superior da tela. A PLA fornecida como entrada para a OPLA-Tool não contém informação a respeito do tipo de cada característica. Devido a esse fato, foi necessária uma maneira para associar cada característica a seu tipo. Então, a caixa de seleção destacada em vermelho na Figura 6 foi adicionada para permitir que o arquiteto classifique cada característica da tabela *Architecture Features* antes de movê-la para a tabela *Automatic Prioritized Features*. Os

tipos das características disponíveis na caixa de seleção são obrigatório, opcional e alternativo (subdividido em alternativa inclusiva e alternativa exclusiva). O arquiteto de LPS pode extrair essa informação do modelo de característica da LPS.

Quando a estratégia automática foi selecionada, após ter associado todas as características aos seus respectivos tipos, o arquiteto deve clicar no botão *Generate Prioritized Features List,* o qual ordenará e salvará a lista para seu posterior uso.

Na estratégia mista, além dos passos citados anteriormente, o arquiteto pode preencher a tabela *Mixed Prioritized Features,* a partir das características da tabela *Automatic Prioritized Features,* de acordo com a prioridade de modularização desejada. Ao clicar no botão *Generate Prioritized Features List,* uma lista é gerada contendo as características da tabela *Mixed Prioritized Features* seguidas das características da tabela *Automatic Prioritized Features,* sendo que esta última é ordenada por tipo de característica. A lista resultante é utilizada pelo *Feature-driven Operator*.

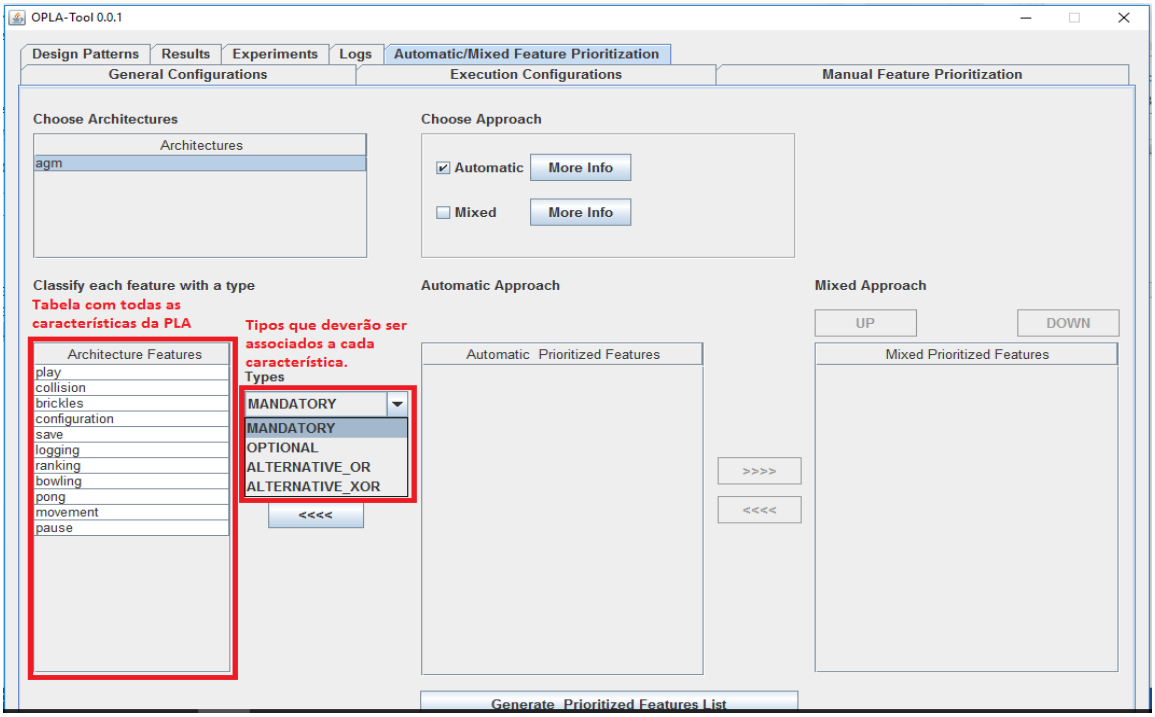


Figura 6. Indicação dos tipos das características e botão inserir na interface gráfica das estratégias automática e mista.

A interação com a estratégia manual é similar à das outras duas estratégias. O arquiteto tem a liberdade de arrastar as características da tabela que contém as características da PLA para a tabela das características priorizadas (e vice-versa). Os botões *UP* e *DOWN* movem a característica selecionada na lista de priorização para cima e para baixo, respectivamente, a fim de permitir que o arquiteto altere a ordem de prioridade das características.

4. Definição do Estudo de Caso

Um estudo de caso foi desenvolvido com o objetivo de comparar e verificar se as estratégias implementadas impactam diretamente na modularização das características de 2 PLAs. As duas PLAs utilizadas são: Arcade Game Maker (AGM) e Mobile Media (MM). AGM é uma LPS criada pelo [SEI, 2018] que inclui três jogos de arcade. A MM é uma LPS que apoia o gerenciamento de diferentes tipos de mídias [Contieri Junior et al, 2011]. Ambas PLAs são baseadas em componentes e seguem o estilo em camadas. A

Tabela 1 apresenta os números de elementos arquiteturais das duas PLAs bem como seus valores originais de *fitness* para as funções objetivo CM e FM.

Tabela 1. Informações sobre os projetos de PLA utilizados no estudo

PLA	# Componentes	# Interfaces	# Classes	# Variabilidades	# *Características*	*Fitness* Original
AGM	9	14	30	5	11	(CM: 89.1, FM: 758.0)
MM	8	15	14	7	14	(CM: 72.1, FM: 1122.0)

A escolha das características cuja modularização seria priorizada se deu de forma aleatória. Porém, algumas características que dificilmente seriam modularizadas devido ao entrelaçamento com outras características em um mesmo componente não foram priorizadas, como por exemplo, a característica obrigatória *Play* da AGM.

Para cada PLA, foram realizados 4 experimentos utilizando a ferramenta OPLA-Tool, sendo um para cada estratégia de priorização e outro sem a utilização de estratégia de priorização, o qual foi utilizado como experimento controle a fim de possibilitar a comparação entre os resultados alcançados.

Na OPLA-Tool, o algoritmo NSGA-II foi executado 30 vezes em cada experimento, utilizando os seguintes parâmetros: tamanho da população= 100, taxa de mutação= 0,9 e número de gerações= 30000. Além disso, foram aplicados todos os operadores de mutação. As funções objetivo utilizadas foram FM e CM (Seção 2).

5. Análise e Discussão dos Resultados

Esta seção tem como finalidade analisar os resultados dos experimentos realizados no estudo de caso, com o objetivo de verificar se as estratégias propostas neste trabalho se mostraram eficientes. Cada um dos 8 experimentos do estudo de caso retornou entre 3 e 12 alternativas de projeto. Resultados detalhados estão disponíveis em [Cecílio 2017].

Percebeu-se que todas as soluções obtidas pelos experimentos têm maior número de pacotes que as PLAs originais, o que é um indício de que algumas características foram modularizadas em novos pacotes criados pelo *Feature-driven Operator*. Além disso, notou-se que os experimentos realizados com as três estratégias de priorização obtiveram soluções com diferentes números de pacotes, os quais possivelmente possuem diferentes características priorizadas modularizadas.

Nesse contexto, decidiu-se selecionar duas soluções de cada experimento para uma análise detalhada. Foram selecionadas as soluções de menor valor de função FM, pois esta função avalia a modularização de características impactada diretamente pelo *Feature-driven Operator*, e as soluções de melhor *trade-off* entre as funções objetivo por representar a solução de melhor equilíbrio entre os objetivos otimizados. A solução de melhor *trade-off* é a que tem a menor distância euclidiana à solução ideal. Em cada solução foi analisado se as características priorizadas foram modularizadas, além do número de características priorizadas e do valor da função FM.

Nos experimentos realizados com a AGM, 3 características (*Brickles, Pong e Bowling*) foram priorizadas na estratégia manual (Tabela 2), as quais foram modularizadas com sucesso. 11 características foram priorizadas com a estratégia automática, sendo que apenas 3 delas não foram modularizadas (*Logging, Collision* e *Movement*). Já para a estratégia mista, foram priorizadas 2 características manualmente (*Configuration* e *Movement*) somado a 9 características automaticamente (*Logging, Ranking, Save, Bowling, Brickles, Pong, Collision, Pause e Play*), e somente 2 características (*Logging* e *Collision*) não foram modularizadas (Tabela 2). No experimento com a PLA de controle, 8 das 11 características foram modularizadas.

Nos experimentos com a PLA MM, 3 características (*ViewPlayMedia*, *SMSTransfer* e *LabelMedia*) foram priorizadas utilizando a estratégia manual, sendo que apenas uma delas (*ViewPlayMedia*) foi modularizada (Tabela 2). Utilizando a estratégia automática, 14 características foram priorizadas e 11 foram modularizadas. Na estratégia mista foram modularizadas as mesmas 11 características modularizadas pela estratégia automática, dentre as 14 priorizadas, sendo 4 priorizadas manualmente (*Photo, Video, Music e Favourites*), somado a 10 automaticamente. O experimento de controle conseguiu modularizar 12 das 14 características.

Tabela 2. Número de características priorizadas e modularizadas em cada PLA

	AGM		**MM**	
	Priorizadas	Priorizadas Modularizadas	Priorizadas	Priorizadas Modularizadas
Controle	0	8	0	12
Manual	3	3	3	1
Automática	11	8	14	11
Mista	2 + 9	9	4 + 10	11

Os resultados mostram que as estratégias foram efetivas para modularizar grande parte das características priorizadas. Percebeu-se que as características priorizadas que não foram modularizadas são aquelas altamente entrelaçadas com outras características, o que dificulta sua modularização em pacotes distintos, que é o caso da característica *SMSTransfer* da PLA MM. Observou-se ainda que em alguns casos a estratégia mista foi mais efetiva em modularizar características do que as estratégias manual e automática, como por exemplo a característica *LabelMedia* da PLA MM que, apesar de priorizada nas três estratégias, só foi modularizada no experimento da estratégia mista. Esta estratégia também conseguiu modularizar a característica *Movement* que é obrigatória na AGM, diferentemente do que ocorreu no experimento com a estratégia automática. *Movement* é uma característica muito entrelaçada e o fato de ser obrigatória diminuiu sua prioridade de modularização na estratégia automática.

A Tabela 3 apresenta o valor de *fitness* das soluções analisadas. Dado que se desejava minimizar FM e CM, todas as soluções têm *fitness* melhor que o *fitness* da PLA original. Nota-se uma certa similaridade nos valores de *fitness* das soluções obtidas pelas estratégias automática e mista, especialmente nos valores da função FM. O *fitness* das soluções da estratégia manual são os piores, inclusive considerando o experimento de controle. Isso se deve ao fato de que para esta estratégia somente 3 características foram priorizadas por PLA e nos demais experimentos todas as características tiveram possibilidade de modularização. Outro ponto a destacar é que soluções com o mesmo número de características priorizadas que foram modularizadas possuem diferentes valores de *fitness* por diferentes razões, uma delas é que a ordem de modularização e o número de pacotes criados para modularização de uma mesma característica influenciam na forma de modularização. Um exemplo envolvendo a MM é que uma solução obtida pela estratégia automática tem pacotes exclusivos para as características *Photo* e *Create_Delete,* enquanto que a solução obtida pela estratégia mista tem um pacote com elementos arquiteturais associados exclusivamente a *Create_Delete* e outro com elementos nos quais *Create_Delete* está entrelaçada, incluindo Photo.

Os resultados nos permitiram identificar as seguintes evidências: (a) as estratégias são eficientes para priorizar as características a serem modularizadas, (b) as estratégias mista e automática propiciam melhor modularização de características, (c) em situações em que existam características obrigatórias entrelaçadas, o ideal seria utilizar a estratégia mista (ou ainda a manual), pois o arquiteto pode escolher quais características priorizar durante a modularização, como observado no experimento com a AGM, e (d) as soluções cujo valor de FM é melhor têm maior número de

características modularizadas do que as soluções de melhor *trade-off*, o que valida o resultado obtido pela função objetivo FM.

Tabela 3: *Fitness* das soluções analisadas

	AGM		MM	
	Melhor FM (CM, FM)	**Melhor *Trade-off* (CM,FM)**	**Melhor FM (CM, FM)**	**Melhor *Trade-off* (CM, FM)**
Controle	(43.1, 508.0)	(34.1, 568.0)	(60.0, 801.0)	(51.0, 818.0)
Manual	(41.2, 683.0)	(40.7, 700.0)	(41.1, 882.0)	(40.8, 979.0)
Automática	(30.7, 545.0)	(30.5, 556.0)	(41.0, 835.0)	(30.9, 838.0)
Mista	(50.8, 522.0)	(21.0, 561.0)	(40.8, 799.0)	(30.9, 814.0)

6. Considerações Finais

Neste trabalho foram propostas três estratégias para priorização de modularização de características de uma LPS. A implementação se deu na OPLA-Tool, ferramenta que utiliza algoritmos evolutivos multiobjetivos para otimizar projetos de PLA. Os resultados do estudo de caso envolvendo 2 PLAs e as estratégias propostas mostraram que a maioria das características priorizadas foram modularizadas conforme o esperado.

A principal contribuição deste trabalho é a extensão da OPLA-Tool de modo a aprimorar o operador de busca que otimiza a modularização de características de LPS. Essa extensão permite que o arquiteto interaja com a OPLA-Tool a fim de estabelecer prioridades para a modularização de características e, com isso, possibilitar que a otimização de um projeto de PLA dado como entrada seja guiada de modo a convergir para a modularização de características atendendo as prioridades estabelecidas pelo arquiteto ou de acordo com o tipo de característica. É válido destacar que não havia na literatura estratégias nesse sentido para o contexto de otimização de projeto de PLA.

Como trabalhos futuros, pretende-se realizar outros experimentos aplicando as três estratégias propostas e analisar a qualidade das alternativas de projeto encontradas do ponto de vista de arquitetos de LPS.

Agradecimentos

Os autores agradecem à CAPES e ao CNPq pelo apoio financeiro.

Referências

Cecílio, M. A. V. (2017). “Definindo a prioridade de modularização por tipo de característica de linha de produto de software”. 2017. 52 f. Monografia, Universidade Estadual de Maringá, Maringá.

Colanzi, T. E.; Vergilio, S. R.; Gimenes, I. and Oizumi, W. N. (2014) “A search-based approach for software product line design”, In: Software Product Line Conference, pp. 237-241, Florence, Italy.

Contieri Jr, A. C.; Correia G. G.; Colanzi, T. E.; Gimenes, I.; OliveiraJr, E.; Ferrari, S.; Masiero, P. C.; Garcia, A. F. (2011) “Extending UML Components to Develop Software Product-Line Architectures: Lessons Learned”, In: European Conf. on Software Architecture, pp. 130-138, Essen, Alemanha.

Deb, K.; Pratap, A.; Agarwal, S. and Meyarivan, T. (2002) “A fast and elitist multiobjective genetic algorithm: NSGA-II”. IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197.

Díaz, J.; Garbajosa, J.; Pérez, J.. (2011) “Prioritization of Features in Agile Product Line Engineering”. In: XP 2011, LNBIP 77, p. 318–319, 2011.

Efat, Md. Iftelharul Alam; et al. (2014) “Feature Prioritization for Analyzing and Enhancing Software Reusability”. 3rd Intl. Conference on Informatics, Electronics & Vision 2014. pp. 1-5

Féderle, É. L.; Ferreira, T. N.; Colanzi, T. E. and Vergilio, S. R. (2015) “OPLA-Tool: a support tool for search-based product line architecture design”. In: Software Product Line Conference, pp. 370–373.

Pohl, K.; Böckle, G.; Linden, F. v. D. (2005). “Software Product Line Engineering: Foundations, Principles, and Techniques”.

SEI (2018). Arcade Game Maker Pedagogical Product Line. In: http://www.sei.cmu.edu/productlines/ppl/

Verdecia, Y. D.; Colanzi, T. E.; Vergilio, S. R. and Santos, M. C. B. (2017) “An enhanced evaluation model for search-based product line architecture design”. In: XX Ibero-American Conference on Software Engineering (CIbSE - ICSE 2017).

N2PC: Um Algoritmo Evolucionário Multiobjetivo baseado em Cones Poliédricos construídos através de Preferências

Pamella Soares[1]✉, Raphael Saraiva[1], Allysson Allex Araújo[2], Gustavo Campos[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.
pamella.soares@aluno.uece.br

Abstract. *Multiobjective Evolutionary Algorithms have been widely adopted in Search Based Software Engineering. However, in scenarios composed of many objectives, the Decision Maker (DM) has to face a number of non-dominated solutions. In addition, it has been discussed strategies to incorporate the DM's preferences and, consequently, to present solutions in his/her region of interest. Given these motivations, this work proposes and evaluates an evolutionary algorithm called N2PC, whose assumption is based on delimiting a certain region of interest through polyhedral cones formed according to the DM's preferences. Regarding the experiments, it was compared the N2PC performance to different NSGA-II configurations. It was concluded that N2PC was superior in convergence and distribution when compared to the other algorithms under evaluation.*

Resumo. *Algoritmos Evolucionários Multiobjetivos têm sido amplamente utilizados na Engenharia de Software Baseada em Busca. Todavia, sabe-se que, em um cenário composto por muitos objetivos, o Tomador de Decisão (TD) pode deparar-se com uma quantidade muito grande de soluções não-dominadas. Aliado a isso, muito se discute sobre estratégias para incorporar as preferências que o TD detém e, consequentemente, apresentar soluções que estejam numa região de seu interesse. Diante de tais motivações, o presente trabalho propõe e avalia um algoritmo evolucionário denominado N2PC, cuja premissa baseia-se em restringir uma certa região de interesse através de cones poliédricos formados a partir das preferências do TD. Em relação aos experimentos, avaliou-se a performance do N2PC em contraste à diferentes configurações do NSGA-II. Em termos de resultados, constatou-se que o N2PC foi superior em convergência e distribuição de soluções quando comparado aos demais algoritmos avaliados.*

1. Introdução

A Engenharia de Software Baseada em Busca (em inglês *Search Based Software Engineering* - SBSE) modela problemas da Engenharia de Software (ES) como problemas de busca [Harman et al. 2012], em que objetiva-se encontrar soluções viáveis que correspondam a valores ótimos quantificados através de uma função objetivo [Coello et al. 2002]. Nesse contexto, diversos problemas em ES são inerentemente complexos e caracterizados por envolverem mais de um objetivo, os quais muitas vezes são conflitantes entre

si [Harman 2007]. Para lidar com tais desafios, Algoritmos Evolucionários Multiobjetivos (AEM's) têm se provado eficientes em identificar soluções que melhor representem o compromisso entre todos os objetivos que, neste caso, refere-se ao conjunto de soluções não dominadas, também conhecido como fronteira (ou frente) de Pareto [Deb 2001].

Porém, diante de um problema multiobjetivo, o Tomador de Decisão (TD) pode deparar-se com uma quantidade muito grande de soluções possíveis na fronteira de Pareto [Zio and Bazzo 2012]. Este problema tende a se tornar mais crítico conforme a quantidade de objetivos aumenta [Ishibuchi et al. 2014]. Tendo em vista tal desafio, diversas abordagens têm sido desenvolvidas com o propósito de restringir as soluções da fronteira de Pareto em uma certa região de interesse, pois, conforme Adra, Griffin e Fleming (2007) destacam, nem sempre o TD está interessado em toda fronteira de Pareto, mas apenas na região que satisfaz suas preferências em relação aos objetivos analisados.

Muitas das abordagens propostas nesse contexto são baseadas na concepção de incorporar as preferências do TD no processo de otimização [Wagner and Trautmann 2010]. De acordo com Miettinen (1999), as preferências do TD podem ser incorporadas em três momentos: i) *a priori*: as preferências são definidas antes de iniciar o processo de busca; ii) *interativamente*: as preferências são incorporadas durante o processo de busca; iii) *a posteriori*: as preferências são utilizadas no final do processo de busca em prol da escolha de uma solução de interesse na frente de Pareto.

Nesse contexto, Mkaouer *et al.* (2013) destacam o uso de três estratégias genéricas para modelagem de preferências em AEM's. A primeira, denominada *Weighting Coefficients*, caracteriza-se por requisitar um coeficiente para expressar o peso de cada objetivo. Quão maior esse peso, mais importante é o objetivo. Na segunda abordagem, chamada *Desirability Threesholds*, requisita-se do TD um valor objetivo absolutamente satisfatório e um valor objetivo marginalmente inviável. Esses limites representam os parâmetros que definem *Desirability Functions*. Finalmente, tem-se o *Reference Point Method* (RPM), no qual o TD fornece, para cada objetivo, um "nível de aspiração" desejado que ele busca alcançar. Mais do que uma ferramenta de tomada de decisão, através do RPM, torna-se possível fomentar o aprendizado do TD sobre diferentes cenários [Wierzbicki 1999].

Naturalmente, o uso de abordagens baseadas em preferências também demonstra-se pertinente sob o contexto de SBSE [Ferreira et al. 2017]. Em grande parte, a motivação para o desenvolvimento de tais propostas deve-se a dificuldade em modelar precisamente o conhecimento tácito do TD através de uma função objetivo [Aljawawdeh et al. 2015]. Adicionalmente, muito se discute sobre a resistência que o TD pode apresentar ao não sentir-se parte do processo de resolução [Miettinen 1999]. Conforme destacado por Meignan *et al.* (2015), "a falta de confiança em sistemas automatizados constituí-se como um obstáculo crítico para o uso efetivo de métodos de otimização".

Diante dos fatos mencionados, apesar do conceito do RPM ser recorrente na literatura, o seu uso atrelado a ideia de cones poliédricos ainda não foi pesquisado em SBSE. Além disso, constata-se quão necessário se faz o desenvolvimento e o aprimoramento de abordagens que contemplem adequadamente as preferências do TD. Em virtude disso, esta pesquisa tem como objetivo principal apresentar e avaliar um novo algoritmo denominado N2PC, cuja premissa baseia-se em restringir uma certa região de interesse através de cones poliédricos formados a partir das preferências obtidas *a priori* do TD.

2. NSGA-II baseado em Cones Poliédricos (N2PC)

A abordagem proposta neste trabalho é uma adaptação do algoritmo NSGA-II elaborado por Deb et al. (2002). Em síntese, a ideia é obter, *a priori*, as preferências do TD sobre os objetivos a serem otimizados e, a partir desses níveis de aspiração, formar cones poliédricos que restrinjam o espaço de busca para a região de interesse do TD. A partir dessas inspirações, o algoritmo proposto foi nomeado como N2PC (*Non-dominated Sorting Genetic Algorithm II Polyhedral Cone*).

O conceito de cones poliédricos refere-se a um tipo particular de cone definido por um conjunto de soluções de um sistema homogêneo de desigualdades lineares construído com um número finito de hiperplanos delimitadores, isto é, os lados do cone poliédrico em M dimensões [Sinha et al. 2014]. Tal estratégia demonstra-se pertinente sob o contexto da presente pesquisa devido a capacidade de modificar o princípio de dominância de um AEM e guiar a busca para a região preferida do TD a qual é definida de acordo com o conceito de níveis de aspiração oriundo do RPM. No contexto dessa pesquisa, os níveis de aspiração são representados através de valores de preferência complementares especificados para cada objetivo. Por exemplo, o TD almeja soluções da região de interesse que reflitam um cenário o qual atenda um mínimo de 70% e um máximo de 100% de um dado Objetivo A. Consequentemente, os valores de preferências em relação a um Objetivo B serão definidos como um mínimo de 0% e um máximo de 30%.

Conforme ilustrado na Figura 1, a arquitetura do N2PC divide-se em três módulos: i) *Módulo Gerenciador*, ii) *Módulo Automático* e iii) *Módulo Interativo*. A seguir, será detalhado o processo de execução do N2PC considerando a relação entre tais módulos.

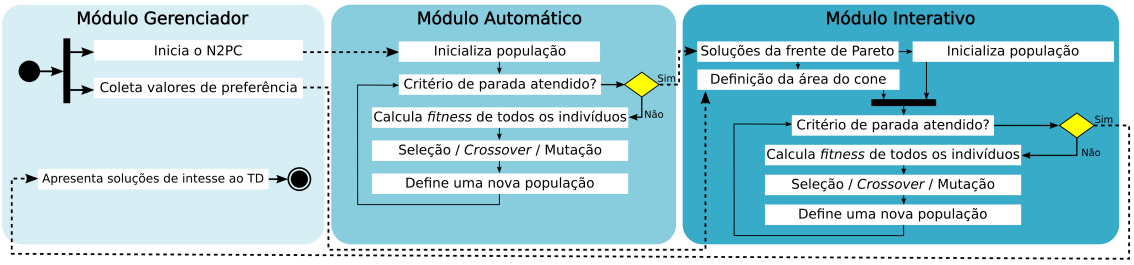


Figura 1. Fluxograma do N2PC

Primeiramente, o *Módulo Gerenciador* inicializa o algoritmo e coleta as informações de preferência correspondentes aos valores (limites) desejáveis pelo TD para cada objetivo, ou seja, os níveis de aspiração [Wierzbicki 1999]. Em seguida, no *Módulo Automático*, executa-se o processo evolucionário canônico sem considerar as preferências do TD com o propósito de gerar uma fronteira de Pareto, a qual será utilizada como entrada para o *Módulo Interativo*. A ideia dessa execução prévia é realizar um *tuning* das soluções que irão compor a população inicial utilizada no *Módulo Interativo*. Finalmente, no *Módulo Interativo*, define-se, considerando as preferências do TD, a área que o cone irá restringir da fronteira de Pareto obtida anteriormente. A partir de tal fronteira, realiza-se outro processo evolucionário no qual as soluções contidas na região do cone passam a dominar as que estão externas a ele, direcionando, assim, a busca para uma região de interesse condizente com as preferências estabelecidas pelo TD. Assim, espera-se que o N2PC seja capaz de convergir para soluções mais adequadas em termos subjetivos e, inclusive, melhores do que o NSGA-II devido ao processo de *tuning* e a restrição do espaço

de busca. Ao atingir o critério de parada, a fronteira de Pareto identificada é enviada para o *Módulo Gerenciador* de modo que tais soluções sejam apresentadas ao TD.

A Figura 2 ilustra a utilização do cone baseando-se numa região de interesse cujo o espaço de busca é composto por dois objetivos: minimizar $f1$ e maximizar $f2$. A formação do cone poliédrico depende i) dos valores de preferência do TD representados pelas duas soluções extremas da região de interesse e ii) de uma solução denominada *Solução NAD* a qual forma o vértice do cone poliédrico no espaço m-dimensional. As soluções extremas, advindas da frente gerada pelo *Módulo Automático*, são encontradas através do RPM o qual utiliza os níveis de aspiração do TD, valorados entre um intervalo de 0% e 100% para cada objetivo, a fim de definir os limites da região de interesse. Em relação a *Solução NAD*, esta é definida como a pior possível para os dois objetivos, onde para $f1$ seria o maior valor e para $f2$ seria o menor valor possível. Após identificar estas três soluções, dois vetores são formados, delimitando assim, a área onde o cone deverá ser construído no espaço dos objetivos. Como exemplificado na Figura 2, a intenção é que, ao ter uma região de interesse previamente definida, o processo evolucionário direcione a busca para soluções de interesse na frente de Pareto a partir da *Solução NAD*.

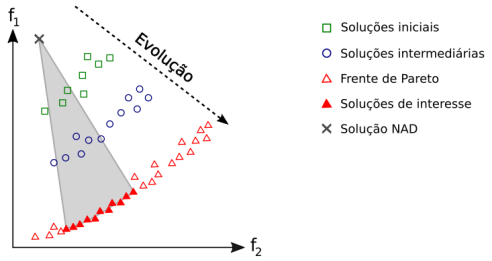


Figura 2. Cone poliédrico em duas dimensões

Para formação dos vetores e, consequentemente, da área do cone, a presente abordagem propõe o uso de uma rede neural como classificador das soluções que são factíveis no espaço dos objetivos, especificamente as que estão contidas dentro da área delimitada pelos dois vetores. O classificador proposto é baseado nos princípios de um discriminador linear, idênticos ao modelo neural de McCulloch e Pitts (MCP) [McClulloch and Pitts 1943]. A Figura 3 esboça a rede neural proposta e os principais parâmetros do classificador. Conforme pode-se perceber, o modelo é organizado em uma rede perceptron com duas camadas: i) uma camada de entrada, alimentada com estímulos que são os dois valores das funções objetivos pertencentes a uma determinada solução e ii) uma camada de saída, responsável por computar um valor de saída para o classificador $o = G(y, y^*, y^1, y^2, W) \in \{0, 1\}^2$, onde Ω_Y o mapeamento do espaço dos objetivos $y \subset R^M$ tal que: $y = f(x) \in \Omega_Y$ representa o valor de entrada do classificador; $y* \in \Omega_Y$ representa a solução NAD, para o problema; y_1 e $y_2 \subset R^M$ são os vetores geradores do conjunto $\Omega_{Y_{cone}} \subset \Omega_Y$; e W representa o conjunto de parâmetros associados a algum classificador específico. Tal classificador indicará se os valores de entrada no espaço dos objetivos pertencem ao conjunto de soluções factíveis da região de definição do cone.

Considerando os dois neurônios representados na Figura 3 e as quatro combinações de valores possíveis na saída da rede (A_1,A_2,B_1,B_2), o classificador proposto é capaz de identificar três classes distintas para qualquer valor de entrada, dependendo deste valor pertencer a um dos três conjuntos de valores distintos, ou seja: A_1,

$B_1 \cap A_2$, e B_2. Consequentemente, considerando que o conjunto de soluções factíveis $Y_{cone} = B_1 \cap A_2$, mais especificamente pode-se afirmar que as soluções factíveis no espaço dos objetivos correspondem aqueles valores de entrada, $y' \in \Omega_Y$, capazes de gerar valores de saída na rede iguais a: $o_1 = 0$ e $o_2 = 1$.

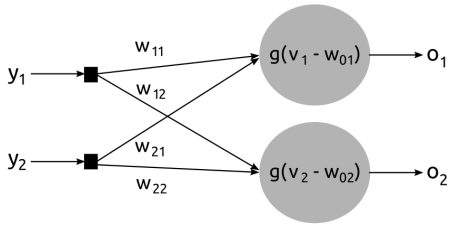


Figura 3. Rede Perceptron com duas camadas empregada como classificador

Vale ressaltar na Figura 3 que os parâmetros W são obtidos da seguinte maneira: $w_{11} = -(y_2^1 - y_2^*)/(y_1^1 - y_1^*)$; $w_{12} = w_{22} = 1$; $w_{21} = -(y_2^2 - y_2^*)/(y_1^2 - y_1^*)$; $v_1 = w_{11}y_1 + w_{21}y_2$; $v_2 = w_{12}y_1 + w_{22}y_2$; $w_{01} = y_2^* - ((y_2^1 - y_2^*)/(y_1^1 - y_1^*))y_1^*$; $w_{02} = y_2^* - ((y_2^2 - y_2^*)/(y_1^2 - y_1^*))y_1^*$; e para $i \in \{1, 2\}$, $o_i = g(v_i - w_{0i}) = 1$ se $v_i \geq w_{0i}$ ou $o_i = g(v_i - w_{0i}) = 0$ se $v_i < w_{0i}$. A expressão que define o argumento da função de ativação de cada um dos dois neurônios na rede representa um conjunto de pontos, H, na reta $vi = w_{0i}$, isto é, que passa pelos pontos $y^* y^i$, e que divide o espaço dos objetivos $\Omega_Y \in R^2$ em duas regiões (conjuntos) de valores A_i e B_i, tal que: dado um valor de entrada $y' \in \Omega_Y$, se $o_i = g(vi - w_{0i}) = 1$ então $y' \in A_i$ ou se $o_i = g(v_i - w_{0i}) = 0$ então $y' \in B_i$.

3. Estudo Empírico

Nesta seção serão apresentadas a descrição do problema usado para validar o N2PC, as especificações da configuração experimental, as questões de pesquisa, as análises dos resultados e, finalmente, as ameaças que podem afetar a validade dos experimentos.

3.1. Problema do Planejamento de *Releases*

O Planejamento de *Releases* é uma atividade importante que consiste no escalonamento de um conjunto de requisitos do sistema em *releases*, a fim de atender às necessidades dos clientes respeitando as restrições relacionadas [Ruhe and Saliu 2005]. A decisão sobre quais requisitos devem ser desenvolvidos em cada *release* envolve vários aspectos conflitantes a serem equilibrados (satisfação do cliente, valor do negócio, riscos envolvidos, recursos disponíveis, entre outros). Nesse contexto, para avaliar o algoritmo proposto pelo presente trabalho, optou-se por adaptar a modelagem proposta por Saraiva *et al.* (2017). A única mudança em relação a proposta original foi a exclusão do objetivo Preferências.

Em síntese, objetiva-se maximizar a satisfação dos *stakeholders* e minimizar os riscos do projeto. Este último é considerado como sendo a quantificação do perigo para o projeto causada pelo possível adiamento da implementação para um requisito definido. Esses objetivos estão sujeitos às seguintes restrições: i) garantir que a soma dos custos de todos os requisitos alocados para cada *release* não exceda o orçamento correspondente; ii) assegurar que cada *release* tenha pelo menos um requisito alocado e; finalmente, iii) respeitar as precedências e as relações de acoplamento entre os requisitos.

3.2. Configurações do Experimento

O presente estudo empírico foi conduzido utilizando dados reais correspondentes a um processador de texto e obtidos em [Karim and Ruhe 2014]. O *dataset* possui 50 requisitos, 4 clientes, 5 *releases* e um orçamento definido com 60% do custo máximo estabelecido para cada *release*. Para representar o TD, foram simulados três diferentes perfis de avaliação a partir de níveis de aspiração definidos. O *Perfil 1* prefere soluções que favoreçam mais a satisfação do que o risco, ou seja, um nível de aspiração que corresponde entre 70% e 100% do objetivo Satisfação. O *Perfil 2* opta por soluções que têm satisfação e risco equilibrados, logo, um nível de aspiração entre 30% e 70% para ambos os objetivos. Por fim, o *Perfil 3* prefere soluções mais favoráveis ao risco do que à satisfação, ou seja, um nível de aspiração entre 70% e 100% do objetivo Risco.

Em relação à técnica de otimização, para fins de análise, considerou-se a geração da população inicial do *Módulo Interativo* de duas formas: i) adicionando as soluções otimizadas (em média 15% do tamanho da população) oriundas do *Módulo Automático*, especificamente as da região limitada pelas preferências do TD (N2PC) ou ii) inicializando a população inicial de modo padrão como realizado no NSGA-II (N2PC-R). O motivo para tal distinção refere-se à necessidade de avaliar a capacidade do algoritmo em convergir mais rapidamente ao inicializar-se com soluções dentro da região de interesse.

Quanto aos parâmetros de configuração dos algoritmos (N2PC-R, N2PC e NSGA-II), foram obtidos empiricamente através de testes preliminares: 100 indivíduos, 100 gerações, uma taxa de *crossover* de 90% e uma taxa de mutação de 1%. Cada perfil foi executado 30 vezes a fim de lidar com a natureza aleatória das meta-heurísticas na coleta dos resultados. Para fins de comparação e cálculo das métricas de qualidade para convergência e distribuição das soluções, foi necessária a geração de uma frente de Pareto de referência (PF_{ref}), visto que uma frente de Pareto real (ótima) é desconhecida para o *dataset* avaliado. A PF_{ref} adotada neste trabalho constitui-se das melhores soluções provenientes de diferentes técnicas de busca, sendo essa a frente disponível que mais se aproxima da frente real [Zhang 2010]. Nesse caso, as meta-heurísticas usadas para geração dessa frente foram: NSGA-II, MoCell, IBEA e SPEA-2.

3.3. Questões de Pesquisa

Os resultados do estudo empírico são discutidos através da análise das seguintes Questões de Pesquisa (QP):

- **QP_1 (Teste de sanidade): O N2PC demonstra-se eficiente ao incorporar as preferências do TD?** *Raciocínio*: Esta QP tem como objetivo identificar se as soluções geradas pelo algoritmo N2PC estão dentro da região de interesse definida *a priori* pelo TD e, consequentemente, de acordo com suas preferências subjetivas.

- **QP_2 (Qualidade das soluções): Qual a capacidade do N2PC em melhorar a qualidade das soluções em termos de convergência e de sua distribuição na região de interesse?** *Raciocínio*: O propósito desta QP é avaliar o impacto da utilização do cone no processo de otimização em relação a qualidade das soluções geradas.

3.4. Resultados e Análises

A Figura 4 apresenta as soluções obtidas ao final do processo de busca do NSGA-II e do N2PC, e como as soluções intermediárias deste último convergem em direção a uma região delimitada pelo cone poliédrico nos diferentes perfis analisados. Nota-se que, nas soluções do *Perfil 1*, os valores de risco tendem à aumentar na medida em que a satisfação geral dos clientes é melhorada, ocupando a região superior da frente de Pareto. De maneira análoga, as soluções do *Perfil 3* estão compreendendo a região que mais favorece a minimização dos riscos do projeto que a satisfação geral dos clientes, contendo soluções na parte inferior da frente de Pareto. Finalmente, o *Perfil 2* apresenta soluções que podem beneficiar tanto na satisfação geral dos clientes, como na redução do risco do projeto. Respondendo a $\mathbf{QP_1}$ e contribuindo como um teste de sanidade, pode-se concluir que, de fato, o uso de cones poliédricos mostra-se eficiente ao direcionar a busca para soluções de uma região de interesse condizente com as preferências estabelecidas *a priori* pelo TD.

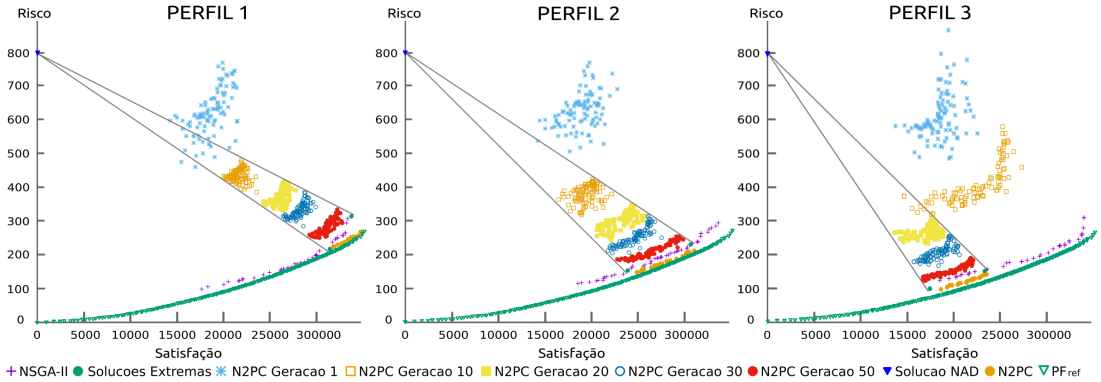


Figura 4. Resultados obtidos para cada cenário

A Tabela 1 apresenta os resultados dos testes estatísticos de Wilcoxon (WC), que identifica a ocorrência de diferença estatística entre as amostras considerando um nível de confiança de 95% utilizando-se da correção de Bonferroni; e Vargha-Delaney ($\hat{A}_{12}$), o qual retorna o número relativo de vezes que um algoritmo produziu valores superiores ao outro. Visando analisar a convergência dos algoritmos, ambos os testes foram aplicados para análise da métrica de qualidade *Generational Distance* (GD) [Van Veldhuizen and Lamont 2000] calculada para cada perfil. Esta métrica mede o quanto a fronteira de Pareto conhecida encontrada pelos algoritmos avaliados está próxima da PF_{ref}. Resumidamente, considera-se que um valor próximo à 0 indica maior proximidade entre a frente analisada e a PF_{ref}.

Algoritmos	Perfis	N2PC-R		N2PC		NSGA-II	
		WC	$\hat{A}_{12}$	WC	$\hat{A}_{12}$	WC	$\hat{A}_{12}$
N2PC	Perfil 1	1.8e-10	0.00	-	-	-	-
	Perfil 2	1.8e-10	0.00	-	-	-	-
	Perfil 3	1.8e-10	0.00	-	-	-	-
NSGA-II	Perfil 1	1.00	0.41	1.8e-10	1.00	-	-
	Perfil 2	7.5e-05	0.82	1.8e-10	1.00	-	-
	Perfil 3	9.7e-10	0.98	1.8e-10	1.00	-	-
$NSGA\text{-}II_{200}$	Perfil 1	3.1e-07	0.09	1.8e-10	1.00	3.4e-06	0.12
	Perfil 2	1.8e-10	0.00	2.1e-08	0.94	1.8e-10	0.00
	Perfil 3	0.05	0.30	6.0e-10	0.98	6.0e-10	0.01

Tabela 1. Valores estatísticos de Wilcoxon e Vargha-Delaney para a métrica GD.

Inicialmente, observando os dados da Tabela 1, percebe-se que houve diferença

estatística na maioria das comparações. Há apenas um caso, no *Perfil 1*, onde o *p-value* foi acima de 0,05, isto é, entre o NSGA-II e o N2PC-R. Porém, nota-se que o N2PC-R obteve menores resultados para os *Perfis 2* e *3*. Ou seja, os valores de GD do NSGA-II foram maiores ao do N2PC-R para esses dois últimos perfis cerca de 82% e 98% das vezes, o que valida a utilização do cone poliédrico durante o processo de otimização. Quando compara-se o N2PC com N2PC-R, para todos os perfis o N2PC obteve resultados menores que o N2PC-R, dado $\hat{A}_{12} = 0$, o que comprova a melhora na convergência quando utilizada soluções do *Módulo Automático* dentro do *Módulo Interativo*. Similarmente, ao contrastar o NSGA-II com o N2PC, verifica-se que o NSGA-II obteve resultados maiores em 100% das execuções para todos os perfis.

Objetivando-se realizar uma comparação mais justa, visto que a população inicial do N2PC contém soluções otimizadas obtidas pelo *Módulo Automático*, foi acrescentada uma outra configuração do NSGA-II com mais 100 gerações no processo de busca denominada NSGA-II$_{200}$. Pode-se constatar que assim como os demais cenários, também há uma superioridade do N2PC em relação ao NSGA-II$_{200}$ sendo que este obteve resultados de GD maiores em 100%, 94% e 98% das vezes para os *Perfis* 1, 2 e 3, respectivamente. Em geral, o N2PC mostrou-se eficiente ao apresentar soluções mais próximas da PF_{ref} quando comparado aos demais algoritmos para todos os perfis analisados.

Tendo em vista a análise em termos da distribuição das soluções, os mesmos testes estatísticos foram aplicados para a métrica de qualidade *Spacing* [Yen and He 2014]. Tal métrica representa o desvio padrão entre as distâncias de soluções consecutivas de um determinado conjunto de soluções. Quanto mais próximo à 0, melhor distribuídas estão as soluções. A Tabela 2 apresenta os resultados dos testes estatísticos (WC e $\hat{A}_{12}$):

Algoritmos	**Perfis**	**N2PC-R**		**N2PC**		**NSGA-II**	
		WC	$\hat{A}_{12}$	**WC**	$\hat{A}_{12}$	WC	$\hat{A}_{12}$
N2PC	**Perfil 1**	1.8e-10	0.00	-	-	-	-
	Perfil 2	1.8e-10	0.00	-	-	-	-
	Perfil 3	1.8e-10	0.00	-	-	-	-
NSGA-II	**Perfil 1**	8.0e-05	0.82	1.8e-10	1.00	-	-
	Perfil 2	0.11	0.67	1.8e-10	1.00	-	-
	Perfil 3	4.6e-06	0.87	1.8e-10	1.00	-	-
NSGA-II$_{200}$	**Perfil 1**	0.0046	0.24	9.7e-10	0.98	2.1e-08	0.05
	Perfil 2	1.8e-10	0.00	0.0018	0.77	1.8e-10	0.00
	Perfil 3	1.00	0.59	2.0e-10	0.99	0.0046	0.20

Tabela 2. Valores estatísticos de Wilcoxon e Vargha-Delaney para a métrica Spacing.

Similar aos resultados da análise do GD, o N2PC também mostrou-se superior em relação à métrica *Spacing* quando comparado aos demais algoritmos. Observa-se que houve diferença estatística na maioria das comparações. Ocorrendo somente um caso em que o valor *p-value*, no *Perfil 2*, foi acima de 0,05 entre o NSGA-II e o N2PC-R. Verifica-se que o N2PC mostrou-se superior nas demais comparações, obtendo menores resultados de *Spacing* em 100% das vezes para todos os perfis quando comparado ao NSGA-II e, cerca de 98%, 77% e 99% das vezes para os *Perfis* 1, 2 e 3, respectivamente, ao ser contrastado com o NSGA-II$_{200}$.

Respondendo a **QP2**, através das análises estatísticas realizadas, constata-se que o uso do cone poliédrico no processo de otimização melhora a qualidade das soluções tanto em termos de convergência quanto em termos de diversidade. Isso porque, como pode-se observar, os valores de GD e *Spacing* apresentam-se menores na maioria das comparações realizadas com os demais algoritmos e suas variações de configuração.

3.5. Ameaças à validade

Discute-se a seguir as ameaças à validade da avaliação empírica realizada neste trabalho [Wohlin et al. 2012].

Em relação à validade interna, apesar de ter sido empregado um *dataset* baseado em dados reais, os valores de risco tiveram que ser simulados, visto que não se encontram disponíveis no *dataset*, e o número de *releases* ampliado. Ademais, a falta de avaliação com mais *datasets*, problemas e, de fato, envolvendo pessoas, prejudica a capacidade de generalização dos resultados, demonstrando-se assim, uma ameaça à validade externa. No que concerne à validade de construção, as métricas utilizadas para estimar a satisfação dos clientes, o risco e orçamento são consideradas que são "estáveis" para todo o ciclo de vida do projeto para o qual o planejamento das *releases* se baseou. Entretanto, tais valores podem mudar conforme o projeto evolui, requerendo a re-execução da abordagem.

4. Considerações Finais

Visando agregar os benefícios da otimização multi-objetiva baseada em preferências à SBSE, o presente artigo propõe e avalia um algoritmo denominado N2PC. Tal algoritmo utiliza o conceito de cones poliédricos construídos a partir de preferências fornecidas, *a priori*, pelo TD. A partir dessas informações, possibilita-se restringir a fronteira de Pareto para uma região de interesse do TD. Um estudo empírico foi feito para avaliar a convergência e distribuição das soluções geradas pelo N2PC. Verificou-se que, de maneira geral, o N2PC superou os demais algoritmos multiobjetivos avaliados. Além disso, constatou-se que a abordagem baseada no uso de cones poliédricos mostra-se factível como auxílio na escolha de uma solução da região de interesse estabelecida pelo TD.

Como parte dos trabalhos futuros, pretende-se conduzir mais experimentos com outros problemas da engenharia de software, além de ampliar o número de objetivos avaliados. Adicionalmente, objetiva-se realizar um estudo empírico envolvendo a presença de usuários com experiência em engenharia de software.

Referências

Adra, S. F., Griffin, I., and Fleming, P. J. (2007). A comparative study of progressive preference articulation techniques for multiobjective optimisation. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 908–921. Springer.

Aljawawdeh, H. J., Simons, C. L., and Odeh, M. (2015). Metaheuristic design pattern: Preference. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1257–1260. ACM.

Coello, C. A. C., Van Veldhuizen, D. A., and Lamont, G. B. (2002). *Evolutionary algorithms for solving multi-objective problems*, volume 242. Springer.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

Ferreira, T. N., Vergilio, S. R., and de Souza, J. T. (2017). Incorporating user preferences in search-based software engineering: A systematic mapping study. *Information and Software Technology*, 90:55–69.

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., 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.

Ishibuchi, H., Masuda, H., and Nojima, Y. (2014). Selecting a small number of non-dominated solutions to be presented to the decision maker. In *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, pages 3816–3821. IEEE.

Karim, M. R. and Ruhe, G. (2014). Bi-objective genetic search for release planning in support of themes. In *SSBSE'14*, pages 123–137. Springer.

McClulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in neurons activity. *Bulletin of mathematical biophysics*, 5:115–133.

Meignan, D., Knust, S., Frayret, J.-M., Pesant, G., and Gaud, N. (2015). A review and taxonomy of interactive optimization methods in operations research. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(3):17.

Miettinen, K. (1999). Nonlinear multiobjective optimization, volume 12 of international series in operations research and management science.

Mkaouer, M. W., Kessentini, M., Bechikh, S., and Tauritz, D. R. (2013). Preference-based multi-objective software modelling. In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, pages 61–66. IEEE Press.

Ruhe, G. and Saliu, M. O. (2005). The art and science of software release planning. *IEEE software*, 22(6):47–53.

Saraiva, R., Araújo, A. A., Dantas, A., Yeltsin, I., and Souza, J. (2017). Incorporating decision maker's preferences in a multi-objective approach for the software release planning. *Journal of the Brazilian Computer Society*, 23(1):11.

Sinha, A., Korhonen, P., Wallenius, J., and Deb, K. (2014). An interactive evolutionary multi-objective optimization algorithm with a limited number of decision maker calls. *European Journal of Operational Research*, 233(3):674–688.

Van Veldhuizen, D. A. and Lamont, G. B. (2000). On measuring multiobjective evolutionary algorithm performance. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 204–211. IEEE.

Wagner, T. and Trautmann, H. (2010). Integration of preferences in hypervolume-based multiobjective evolutionary algorithms by means of desirability functions. *IEEE Transactions on Evolutionary Computation*, 14(5):688–701.

Wierzbicki, A. P. (1999). Reference point approaches. In *Multicriteria decision making*, pages 237–275. Springer.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yen, G. G. and He, Z. (2014). Performance metric ensemble for multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 18(1):131–144.

Zhang, Y. (2010). *Multi-objective search-based requirements selection and optimisation*. PhD thesis, University of London.

Zio, E. and Bazzo, R. (2012). A comparison of methods for selecting preferred solutions in multiobjective decision making. In *Computational intelligence systems in industrial engineering*, pages 23–43. Springer.

Análise de estilo de programação com Rede Neural Recorrente para aprovação automática de *pull request*

Lucas Roque, Altino Dantas, Celso G. Camilo-Junior

[1]Universidade Federal de Goiás (UFG)
Instituto de Informática (INF)
Alameda Palmeiras, Quadra D, Câmpus Samambaia. Goiânia - Goiás - Brasil
Grupo Intelligence for Software (i4Soft)

http://ic1.inf.ufg.br/i4soft
{lucasroque,altinoneto,celso}@inf.ufg.br

Resumo. *Embora reconhecido como importante, o estilo de programação é um aspecto comumente negligenciado pelos desenvolvedores. Seguir o padrão de programação presente em um projeto pode ser útil para entender e manter o sistema. Geralmente, as empresas criam suas próprias diretrizes para codificar correções de bugs ou novos recursos. No entanto, desenvolver este conjunto de regras não é uma tarefa simples, e há inconsistências até mesmo na literatura especializada. Portanto, este artigo propõe uma nova abordagem para a análise de estilo de programação, usando uma rede neural recorrente que aprende o estilo de programação presente em um projeto e determina se um trecho de código submetido a ele segue o padrão do projeto. Tal abordagem será usada em um repositório de desenvolvimento distribuído baseado em Pull Request, auxiliando na aprovação automática de incrementos para o projeto.*

Abstract. *Although recognized as important, programming style is one aspect commonly neglected by developers. Follow the pattern of programming presents in a project may be useful to understand and maintain the system. Usually, the companies build their own guidelines for coding bug fixes or new features. However, developing this set of rules is not a simple task, and there are even inconsistencies in the specialized literature. Therefore, this paper proposes a new approach to programming style analysis, using a recurrent neural network that learns the programming style presents in a project and determines whether a piece of code submitted to it, follows the project's pattern. Such an approach will be used in a distributed development repository based on Pull request, aiding in the automatic approval of increments to the project.*

1. Introdução

Em geral, existem duas propriedades estruturais que afetam a qualidade do software: a corretude e o estilo. A corretude é determinada pelo resultado do software, se este atende ou não à especificação. O estilo diz respeito a como as funcionalidades em todos os níveis são especificadas, descritas, caracterizadas e documentadas [Dromey 1995]. Por razões óbvias, a corretude é sempre levada a sério durante a validação do software, no entanto, o estilo de programação é usualmente negligenciado, gerando códigos de baixa qualidade. O resultado é, por vezes, um software com baixa manutenibilidade, pois maus

hábitos de programação podem tornar impossível para outros programadores entenderem o programa [Ala-Mutka et al. 2004].

Na prática, as regras de estilo de programação são quase sempre indeterminadas, inconsistentes e contraditórias. Diferentes maneiras de usar dispositivos como recuo, alinhamento e comentários podem ser encontradas em livros e artigos que abordam o tópico. A falta de consenso pode ser atribuída à falta de definição do que exatamente é o estilo de programação e quais fatores realmente contribuem para tal. Esta indefinição no estilo de programação está ligada à subjetividade do tema que dificulta o estabelecimento de regras universais [Oman and Cook 1990].

Além de lidar com a questão da subjetividade, abordagens baseadas em regras também precisam ser definidas para linguagens específicas, uma vez que cada linguagem de programação possui sua própria sintaxe. Isso significa que uma empresa que possui programas em várias linguagens precisaria usar ou criar um conjunto de regras específicas para cada uma delas.

Outro aspecto que precisa ser considerado é o modo como os softwares de versionamento foram modificados nos últimos anos, com o surgimento de sistemas de controle de versão baseados em *pull request*. Ao contrário do CVS e SVN, onde os colaboradores precisavam ser participantes da equipe de desenvolvimento e com uma série de permissões concedidas, os atuais sistemas baseados em *pull request* permitem que qualquer usuário contribua para o projeto, aumentando a necessidade de controlar o que realmente deve ser incluído [Rodríguez-Bustos and Aponte 2012].

Neste trabalho, em vez de propor uma ferramenta que siga um conjunto de regras pré-definidas, propõe-se um método que aprende o estilo de codificação de um projeto e garante que qualquer novo código inserido no projeto seja checado quanto à conformidade em relação ao estilo aprendido. Supõe-se que assim como um novo desenvolvedor é treinado para seguir um conjunto de regras, uma Rede Neural Artificial (RNA) também pode cumprir esta tarefa. Contudo, uma RNA não requer um conjunto de regras definidas a priori, e sim analisa ela mesma o código fonte do projeto, podendo definir o conjunto de regras a ser seguido considerando tanto aspectos objetivos quanto os subjetivos.

Esta proposta possui dois potenciais cenários de aplicação. No primeiro, um projeto de desenvolvimento estaria em andamento, com funcionalidades desenvolvidas e, em algum momento, delegar-se-ia a necessidade de usar uma forma de análise do estilo de programação. Neste caso, uma RNA poderia ser treinada com o próprio projeto, garantindo que o estilo corrente do projeto seria continuado. Outro cenário seria um projeto não iniciado, ou o código existente não seguindo um padrão bem definido. Neste caso, o treinamento poderia ser realizado em um projeto ou mais projetos de qualidade conhecida, e o estilo de programação aprendido seria aplicado no decorrer do projeto a ser iniciado.

Portanto, a principal contribuição deste trabalho é a introdução de um novo método para analisar o estilo de programação, baseado em redes neurais recorrentes, sem necessidade de definição de regras a priori ou específicas por linguagem de programação, passível de aplicação em repositórios de desenvolvimento distribuído baseados no paradigma *Pull Request*.

O restante deste trabalho está organizado da seguinte forma: na Seção 1.1 são apresentados conceitos nos quais esta pesquisa se apoia; a Seção 2 traz as definições

da abordagem proposta; em 3 encontra-se a metodologia dos experimentos realizados; em 4 são analisados os resultados obtidos; e finalmente na Seção 6 são apresentadas as considerações finais.

1.1. Background

Esta seção apresenta os dois principais fundamentos deste trabalho, destacando o controle de versão baseado em *Pull Request* e as redes neurais artificiais, mais especificamente as recorrentes.

1.1.1. Paradigma de controle de versão baseado em *Pull Request*

Influenciado por grandes projetos de código aberto, emergiu uma nova forma de controle de versão distribuída baseado no paradigma de *pull*, dando origem a sistemas como Mercurial, Git e Bazaar [Rodríguez-Bustos and Aponte 2012]. A principal diferença deste modelo está na maneira como a colaboração é feita. Nesse modelo, o repositório principal não é compartilhado entre os colaboradores, em vez disso, cada usuário tem um clone do projeto e pode fazer alterações independentemente [Gousios et al. 2015].

Repositórios que implementam tal paradigma têm várias vantagens. Do ponto de vista técnico, o principal é que os desenvolvedores podem atuar como clientes e servidores, recebendo contribuições em seus projetos e adicionando a outros, além de poder trabalhar sem estar conectado ao repositório central. O controle também é mais simples, pois funciona com um conjunto de alterações entre revisões, não versões em que o arquivo possui seu próprio histórico. Do ponto de vista estrutural, esse paradigma permite a implementação de papéis intermediários, como os responsáveis por testar, revisar ou integrar. Além disso, mantém-se um histórico dos eventos nos repositórios intermediários, algo que não era possível antes, e que representa um avanço no rastreamento de mudanças [Rodríguez-Bustos and Aponte 2012].

Podemos citar como exemplos de repositórios que implementam tal paradigma, o GitHub e o BitBucket, que utilizam o modelo Git e permitem criação de repositórios públicos ou privados. Para contribuir em repositórios públicos basta clonar um dos projetos disponíveis. A partir desse momento, o usuário tem um "ramo" de projeto e pode fazer qualquer modificação nele sem causar falha no projeto principal. Se alguém deseja que sua contribuição seja incorporada ao projeto, é necessário realizar um processo chamado *Pull Request*, através do qual solicita-se que a modificação seja analisada como um possível incremento ao projeto [Gousios et al. 2014].

Quando uma solicitação *pull* é realizada, esta é adicionada a uma lista para ser revisada, primeiro por um ou mais revisores e, posteriormente, pelo responsável pelo projeto. Tal revisão é realizada para garantir que o novo incremento atenda às métricas de qualidade do projeto, levando em consideração a cerretude e o estilo de programação [Yu et al. 2014].

1.1.2. Redes Neurais Artificiais

A estrutura das Redes Neurais Artificiais (RNAs) foi desenvolvida inspirada em modelos conhecidos de sistemas nervosos biológicos e do próprio cérebro humano. Os

neurônios artificiais, as unidades de processamento das RNAs, são modelos simplificados dos neurônios biológicos. Computacionalmente, os neurônios artificiais produzem saídas contínuas não-lineares a partir dos dados a eles submetidos como entrada. Para tanto, cada sinal de entrada é ponderado por valores denominados pesos sinápticos. A combinação linear de todos os sinais e seus respectivos pesos é então submetida a uma função de ativação responsável pela saída ou resposta do neurônio.

Uma das vantagens das RNAs é a capacidade de aprender a partir da apresentação de padrões, isto é, exemplos conhecidos, tornando-a capaz de generalizar respostas e produzir uma saída próxima à desejada a partir de sinais de entrada não utilizados no treinamento. Por treinamento, entenda-se como o processo de submissão de amostras conhecidas e ajuste dos valores dos pesos sinápticos até que a saída produzida pela rede seja próxima àquela esperada. A este processo também se dá o nome de algoritmo de aprendizagem [Silva et al. 2010]. Após o processo de treinamento, tem-se um "modelo" treinado, que na prática é a rede em si com os valores de todos os seus pesos sinápticos.

A arquitetura de uma RNA define a forma com seus vários neurônios são dispostos, definindo a direção das conexões sinápticas entre os neurônios. Basicamente, uma RNA pode ser dividida em três partes, chamadas camadas, subdivididas como: i) Camada de entrada; ii) camadas ocultas ou intermediárias; (iii) camada de saída. A primeira é responsável por receber os sinais de entrada, a segunda é responsável pelo processamento interno dos dados de entrada, isto é, atribuir pesos que representem a relação de influência entre as entradas e as saídas esperadas; por fim, as respostas são produzidas pela camada de saída. As principais arquiteturas de RNAs podem ser divididas em: redes *feedfoward* de camada única ou múltipla, redes recorrentes e redes reticuladas [Silva et al. 2010].

Redes Neurais Recorrentes (RNR) são uma arquitetura de RNA na qual as saídas são reintroduzidas na rede. Esse mecanismo qualifica tais redes para o processamento dinâmico de informações e, portanto, pode ser usado em previsões de séries temporais, otimização e identificação de sistemas, controle de processos e outros. Todavia, as conexões recorrentes podem levar a comportamentos complexos, mesmo com um número reduzido de pesos sinápticos a serem ajustados [Tsoi and Back 1997].

Manter o tempo de memória flexível e ajustar os pesos sinápticos eficientemente para as memórias de longo prazo são dois desafios nas RNRs. Neste sentido, o uso de um bloco LSTM (Long Short-Term Memory) é atualmente o estado da arte para atenuar tais questões. Redes neurais recorrentes com blocos LSTM são as mais utilizadas em aplicações como reconhecimento e geração de escrita, tradução de linguagem e análise de áudio [Greff et al. 2017].

Considerando os resultados satisfatórios obtidos pelas redes neurais recorrentes utilizando blocos LSTM na análise e escrita de textos em linguagem natural, e a suposição de que códigos fonte, analogamente a textos em linguagem natural, são regulares, possuindo uma naturalidade de escrita [Devanbu 2013], esta arquitetura foi escolhida para validação da proposta apresentada neste trabalho.

2. Proposta

Conforme apontado anteriormente, a proposta deste trabalho utiliza uma Rede Neural Recorrente para analisar estilo de programação com o objetivo de ajudar na aprovação

automática de *Pull Requests* em ambiente de desenvolvimento distribuído. Para isso, considera-se duas fases distintas: i) Fase de Construção, na qual os códigos de alguns projetos anteriores (ou versões anteriores do mesmo projeto) são pré-processados e enviados para um Algoritmo de Aprendizagem de uma Rede Neural Recorrente; e ii) Fase de Operação, na qual trechos de código são confrontados com o estilo aprendido na primeira fase. A figura 1 ilustra as fases e outros detalhes da abordagem proposta.

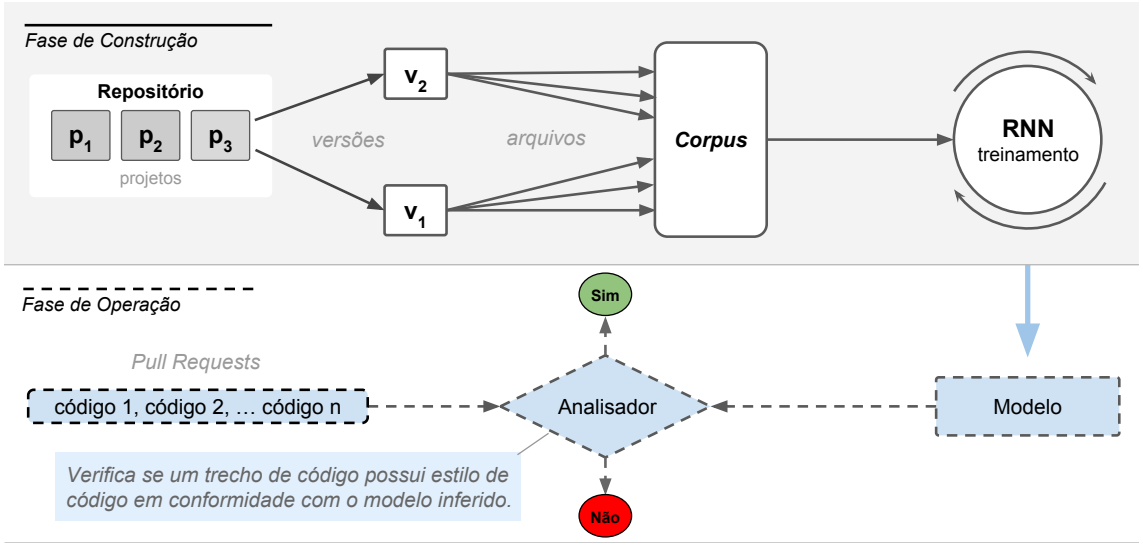


Figura 1. Ilustração da Proposta apresentada em duas fases.

Pela Figura 1 compreende-se que a primeira fase requer a existência de projetos dos quais códigos fonte são extraídos para composição do *corpus* utilizado no treinamento. Por *corpus*, entenda-se um arquivo com vários arquivos de código fonte concatenados. O processo de construção desse *corpus* deve garantir que seu conteúdo seja composto por códigos de uma única linguagem de programação. Além disso, tais códigos devem ser relevantes para analisar o estilo de programação, ou seja, arquivos de configuração ou *bytecodes* não podem ser considerados, por exemplo. Por tudo isso, percebe-se que que a composição do *corpus* é um fator chave para o sucesso da proposta, uma vez que um *corpus* demasiadamente ruidoso denegriria a capacidade da rede capturar o padrão de codificação que de fato expressa o estilo.

Ainda na Fase de Construção, o *corpus* é submetido para algum algoritmo de treinamento que, utilizando uma arquitetura recorrente, seja capaz de processar o volume de dados e capturar o padrão de naturalidade presente nos códigos presentes no *corpus*. Sabendo que os algoritmos de treinamento possuem caráter estocástico, este processo pode ser repetido afim de se obter o modelo mais acurado.

Após a definição do melhor modelo produzido na primeira fase, inicia-se a Fase de Operação. Nesta, um trecho de código é submetido para um analisador que decide se o código enviado segue ou não o estilo de programação aprendido. Esta fase da abordagem corresponde ao componente que efetivamente deve ser implementado dentro do sistema de controle do repositório para que seja executado a cada nova solicitação de inclusão de código à versão corrente do software sob desenvolvimento.

O processo de análise do estilo do código, isto é, a tarefa do componente identifi-

cado como “Analisador” na Figura 1, dá-se da seguinte forma: um *parser* verifica quando uma classe ou função é criada ou chamada, um objeto é instanciado, ocorre uma palavra reservada, surge caracteres de início ou fim de instrução, e submete o trecho de código imediatamente anterior a estas situações para o modelo, que por sua vez responde de acordo com o estilo aprendido, qual deveria ser o caractere posterior ao trecho de código submetido. A Figura 2 mostra alguns exemplos de possíveis entradas e suas respectivas saídas geradas pelo modelo de aprendizado.

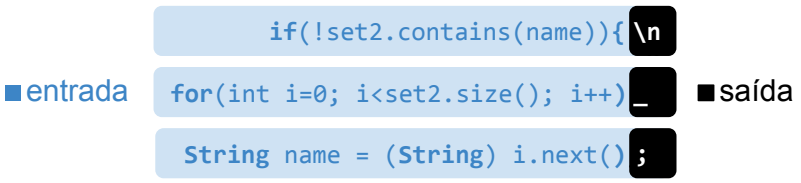


Figura 2. Exemplos de saídas produzidas pelo modelo treinado considerando diferentes entradas.

Como pode ser visto, primeiramente o modelo aprendido recebe uma instrução de um “*if*” até uma abertura de chave, então, o modelo de aprendizado sugere que o próximo caractere seja uma quebra de linha “\n”. No segundo e terceiro exemplo a rede recebe trechos que são finalizados com o fechamento de parênteses, porém sugere caracteres diferentes. No segundo, um espaço (representado por “_”) e no terceiro um “;”.

Considerando a análise de todas as linhas de um código (uma classe java, por exemplo) submetido em um *Pull request*, é definido o quanto o código está em conformidade com o estilo de programação do projeto. Esta tarefa é factível pela comparação do caractere sugerido pela rede, com o caractere que realmente está presente no código sob análise. Assumindo que a RNA realmente aprendeu o estilo de programação, este procedimento é capaz de determinar se a submissão segue o estilo presente no projeto.

3. Experimentos

Como forma de avaliar nossa proposta, utilizamos uma Rede Neural Recorrente composta por duas camadas ocultas com blocos LSTM, cada uma com 256 neurônios. Depois de cada camada LSTM é realizado um *dropout* em 20 % dos neurônios, processo que consiste em uma exclusão aleatória de um determinado número de neurônios, visando minimizar o *overfitting*. Uma camada *Fully Connected* com função de ativação *softmax* é responsável pela saída da rede.

A rede descrita acima foi treinada em um projeto real com mais de dez mil linhas de códigos, desenvolvido em C#. O treinamento foi executado durante muitas épocas, também de forma recorrente, isto é, a saída da primeira camada é a entrada da segunda e assim por diante.

Como a análise de estilo de programação envolve várias variáveis, agravadas pelo fato de que a abordagem proposta leva em conta aspectos que não estão claros a princípio, o primeiro passo foi definir um conjunto de caracteres alvos que teriam seus comportamentos analisados. De forma empírica foram escolhidos cinco caracteres que indicam início e fim de blocos de código, são eles: “{”, “}”, “(”, “)” e “;”. Portanto, sempre

que a rede encontrar um desses caracteres, o seu comportamento será analisado. É importante ressaltar que estes não são os únicos aspectos analisados pela rede, mas aqueles monitorados por esse experimento.

Para dar credibilidade à análise das respostas dadas pela rede neural, foi realizada uma análise de frequência no código do projeto utilizado para treinamento. A ideia é garantir que as sugestões dadas pela rede estejam de acordo com o estilo presente no projeto. Isso foi feito contando os caracteres que aparecem após os escolhidos para validação do modelo. Por exemplo, em um projeto, foi analisado quais caracteres mais se repetem após "{", assim, se a rede sugere algo não está presente no código, isso significa que ela não conseguiu aprender o estilo de programação presente no projeto. Lembrando que essa etapa foi realizada apenas para mostrar a eficiência da abordagem, que uma vez comprovada, eliminará a necessidade desse passo.

Por fim, as classes selecionadas para os testes foram submetidas à rede treinada. A classe é analisada caractere por caractere, e sempre que qualquer um dos caracteres alvo escolhidos for encontrado e passado para a rede, sua resposta será analisada. Destacando que os caracteres definidos envolvem diferentes dificuldades de sugestão, como será mostrado na análise dos resultados, presente na próxima sessão.

4. Resultados

Nesta seção, discutiremos os resultados das duas análises realizadas para validar a abordagem proposta. Começando com a análise de frequência, seguida da própria análise de estilo de programação.

4.1. Análise de Frequência

A Tabela 1 mostra o resultado da análise de frequência do projeto. As colunas representam os caracteres alvo e as linhas as estruturas às quais eles pertencem. As células de interseção indicam, quando elas existem, os três caracteres mais recorrentes.

Tabela 1. Análise de Frequência

Estrutura	{	}	(	)	;
if	\n, " " , \t	\n, " ",)	alpha,), (	\n, " ", ;	
else	\n, " "	\n, " ", ;			
for	\n, " "	\n,), " "	alpha,), " "	;,], \n	
while	\n	\n	alpha,), (	\n,), ;	
try	\n, " "	\n, " ",)			
catch	\n	\n, " "	alpha, " ",)	\n, ;, " "	
função			alpha,), \n	\n,], " "	\n, " ", \t
atribuição			alpha,), " "	;,),]	\n, " ", \t

Como pode ser visto, a maioria das vezes que um "{" ou "}" é escrito, o próximo caractere é uma quebra de linha, chegando em alguns casos a 100%. Por outro lado, quando um "(" é escrito, a tendência é que o próximo seja um caractere alfanumérico, enquanto que depois de um ") ", geralmente vem um espaço em branco. Ao analisar "; ", a frequência é repetida tanto na criação da função quanto na atribuição da variável, quebrando a linha na maioria dos casos.

É importante ressaltar que os caracteres alvo “{”, “}” pertencem a funções, assim, a análise de recorrência encontrou um máximo de três caracteres possíveis, enquanto com “(”, “)” e “;” que são pertencentes a atribuições de variáveis entre outros, geram maiores as possibilidades. Isso se deve ao fato de que o número de usos deles é diverso, aumentando a complexidade de entender seu comportamento. Inclusive, dificultando a criação de regras de como usá-las, como são feitas em outras abordagens.

4.2. Análise de Estilo de Programação

A Tabela 2 mostra os resultados obtidos e está estruturado da seguinte forma: As colunas representam os caracteres alvo escolhidos, enquanto as linhas são as estruturas às quais esse caractere pertence. O valor percentual representa o número de vezes que a rede sugeriu um caractere que faz parte dos três definidos como com mais frequentes na análise anterior. Assim, a célula destacada em amarelo indica que em 100% das vezes que a rede recebeu um caractere “{” pertencente a um “if”, sugeriu como o próximo caractere um dos mais frequentes. Os caracteres alvo são analisados de acordo com um determinado critério, por exemplo, não faz sentido analisar um “{” em uma chamada de função ou atribuição de variável, ou “;” em um “try”.

Tabela 2. Resultado da análise de estilo de programação

Estrutura	{	}	(	)	;
if	100%	100%	91%	86%	
else	100%	100%			
for	100%	100%	100%	84%	
while	100%	100%	100%	100%	
try	100%	100%			
catch	100%	100%	100%	100%	
função			80%	80%	92%
atribuição			94%	93%	100%

Os resultados mostram que para os caracteres “{” e “}” a rede neural sugere os caracteres mais frequentes no projeto em 100% das vezes, lembrando que estes alvos eram os de menores dificuldades. Em relação aos caracteres “(” e “)” a rede não atinge acurácia máxima em todos os casos, porém ficando sempre acima dos 80%. Quando analisamos “;”, os resultados são melhores que o anterior, com 92% em chamadas de função, e 100% em atribuições.

Os casos em que a RNA não atinge o máximo acontecem nos casos em que a quantidade de possibilidades é muito grande, o que significa que a rede não está necessariamente errada, apenas o que ela sugere não é uma das três ocorrências mais frequentes.

5. Trabalhos Relacionados

Man e Cook (1989) propuseram uma técnica de análise de autoria por meio do estilo de programação. A técnica baseou-se no princípio de que os programadores deixam “impressões digitais” em seus códigos, similar à análise de textos literários, em que os autores costumam usar palavras e frases específicas. Os métodos que usam tais técnicas são geralmente empregados para identificar plágios e são centrados na comparação da

decomposição estrutural de sistemas e na aplicação de uma bateria de métricas de complexidade de software para identificar programas suspeitos. Os testes foram realizados em códigos Pascal, utilizando aspectos como uso de comentários, número de espaços utilizados para recuo, uso de letras maiúsculas e minúsculas.

No trabalho de Lake e Cook (1990), uma ferramenta chamada STYLE foi proposta, também para a linguagem Pascal, que realiza uma análise automática do código-fonte do programa submetido a ela. O objetivo da ferramenta é produzir mensagens de advertência ou de congratulações para os usuários, levando em consideração economia, modularidade, simplicidade, estrutura e layout. A forma como a análise é feita é também através de regras definidas, e os autores mencionam as dificuldades de encontrar regras consistentes para serem usadas.

Ala-Mutka et al. (2004) criaram uma ferramenta de análise de estilo de programa para auxiliar estudantes de programação na linguagem C++. Como na concepção dos autores, o estilo de programação não é considerado pelos programadores iniciantes, a ferramenta os ajudaria a produzir códigos de melhor qualidade, atendendo a um conjunto de regras previamente definidas. As regras foram definidas por um conjunto de professores de programação, reunindo o que eles consideravam boas práticas de programação. A ferramenta considera, por exemplo, o número de linhas em branco, comentários e recuo.

Dada a dificuldade de definir as regras do estilo de programação, alguns trabalhos foram desenvolvidos apenas para sua definição. Neste contexto, tanto Oman e Cook (1990) quanto Dromey (1995) pretendem definir uma taxonomia para o estilo de programação e quais aspectos as regras devem cobrir.

Como pode ser visto nos trabalhos apresentados, existe uma preocupação em ajudar os programadores a desenvolver códigos de melhor qualidade. Todavia, as técnicas geralmente realizam isso apenas analisando regras previamente estabelecidas. O que este trabalho se propõe a fazer é realizar a validação de estilo considerando além destas regras, que são naturalmente aprendidas por uma RNA, fatores subjetivos que não podem ser descritos em um conjunto de regras.

6. Considerações Finais

Este trabalho propôs uma nova abordagem para realizar a análise do estilo de programação, uma vez que o modo atual de realizar tal tarefa considera apenas aspectos objetivos, que precisam ser previamente definidos através de regras. A abordagem proposta utiliza uma RNA recorrente que aprende a maneira como o código é escrito e extrai o padrão utilizado considerando os aspectos objetivos e subjetivos.

A rede mostrou-se eficiente para o aprendizado do estilo de programação presente no projeto alvo. Conseguindo atingir 100% de sucesso na maioria dos casos e bons resultados, mesmo nos pontos mais difíceis, revelando a capacidade da rede neural recorrente de aprender e replicar o padrão presente no código.

Como trabalho futuro, espera-se encontrar uma maneira mais eficiente de analisar a precisão do modelo, possivelmente incluindo também algum aspecto subjetivo. Além disso, pretende-se expandir os experimentos analisando projetos desenvolvidos em outras linguagens de programação, e aplicar o estilo aprendido em um projeto um projeto distinto, demonstrar a eficiência da abordagem na extração de estilo de programação.

7. Agradecimentos

O presente trabalho foi realizado com apoio de Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001 e da Fundação de Amparo à Pesquisa do Estado de Goiás (FAPEG) - Processo 201710267000615.

Referências

Ala-Mutka, K., Uimonen, T., and Jarvinen, H.-M. (2004). Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research*, 3:245–262.

Devanbu, P. (2013). On the naturalness of software. In *Proceedings of the 6th India Software Engineering Conference*, ISEC '13, pages 61–61, New York, NY, USA. ACM.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on software engineering*, 21(2):146–162.

Gousios, G., Pinzger, M., and Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM.

Gousios, G., Zaidman, A., Storey, M.-A., and Van Deursen, A. (2015). Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2017). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232.

Lake, A. and Cook, C. (1990). Style: an automated program style analyzer. *ACM SIGCSE Bulletin*, 22(3):29–33.

Oman, P. W. and Cook, C. R. (1989). Programming style authorship analysis. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326. ACM.

Oman, P. W. and Cook, C. R. (1990). A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 244–250. ACM.

Rodríguez-Bustos, C. and Aponte, J. (2012). How distributed version control systems impact open source software projects. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 36–39. IEEE.

Silva, I. d., Spatti, D. H., and Flauzino, R. A. (2010). Redes neurais artificiais para engenharia e ciências aplicadas. *São Paulo: Artliber*, pages 33–111.

Tsoi, A. C. and Back, A. (1997). Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15(3-4):183–223.

Yu, Y., Wang, H., Yin, G., and Ling, C. X. (2014). Reviewer recommender of pull-requests in github. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 609–612. IEEE.

Lista de Autores | *Author Index*

A
Alvim, Adriana Cesário de Faria 11
Andrade, Andre 11
Araújo, Allysson Allex 41

B
Barros, Márcio 11

C
Camilo-Junior, Celso 51
Campos, Gustavo 41
Cecílio, Matheus 31
Colanzi, Thelma 31

D
Dantas, Altino 51

K
Kessentini, Marouane 1

L
Lima, Jackson Prado 21

M
Mariani, Thainá 1

R
Roque, Lucas 51

S
Saraiva, Raphael 41
Silva, Diego 31
Soares, Pamella 41
Souza, Jerffeson 41

V
Vergilio, Silvia 1, 21