



Congresso Brasileiro
de Software: Teoria e Prática **BRASILIA.2013**

Congresso Brasileiro de Software: **Teoria e Prática**

29 de setembro a 04 de outubro de 2013



Brasília-DF

Anais

WESB 2013

IV WORKSHOP DE ENGENHARIA DE SOFTWARE BASEADA EM BUSCA



WESB 2013

IV Workshop de Engenharia de Software Baseada em Busca

29 de setembro de 2013
Brasília-DF, Brasil

ANAIIS

Volume 01
ISSN: 2178-6097

COORDENAÇÃO DO WESB 2013

Arielo Claudio Dias Neto
Gledson Elias
Jerffeson Teixeira de Souza
Silvia Regina Vergilio

COORDENAÇÃO DO CBSOFT 2013

Genaina Rodrigues – UnB
Rodrigo Bonifácio – UnB
Edna Dias Canedo – UnB

REALIZAÇÃO

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (DIMAp/UFRN)

PROMOÇÃO

Sociedade Brasileira de Computação (SBC)

PATROCÍNIO

CAPES, CNPq, Google, INES, Ministério da Ciência, Tecnologia e Inovação, Ministério do Planejamento, Orçamento e Gestão e RNP

APOIO

Instituto Federal Brasília, Instituto Federal Goiás, Loop Engenharia de Computação, Secretaria de Turismo do GDF, Secretaria de Ciência Tecnologia e Inovação do GDF e Secretaria da Mulher do GDF



WESB 2013

4th Brazilian Workshop on Search Based Software Engineering

September 29, 2013
Brasília-DF, Brazil

PROCEEDINGS

Volume 01
ISSN: 2178-6097

WESB 2013 CHAIRS

Arielo Claudio Dias Neto
Gledson Elias
Jerffeson Teixeira de Souza
Silvia Regina Vergilio

CBSOFT 2013 GENERAL CHAIRS

Genaína Rodrigues – UnB
Rodrigo Bonifácio – UnB
Edna Dias Canedo – UnB

ORGANIZATION

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (DIMAp/UFRN)

PROMOTION

Brazilian Computing Society (SBC)

SPONSORS

CAPES, CNPq, Google, INES, Ministério da Ciência, Tecnologia e Inovação, Ministério do Planejamento, Orçamento e Gestão e RNP

SUPPORT

Instituto Federal Brasília, Instituto Federal Goiás, Loop Engenharia de Computação, Secretaria de Turismo do GDF, Secretaria de Ciência Tecnologia e Inovação do GDF e Secretaria da Mulher do GDF

Autorizo a reprodução parcial ou total desta obra, para fins acadêmicos, desde que citada a fonte

APRESENTAÇÃO

O Workshop de Engenharia de Software Baseada em Busca (WESB) constitui um fórum de discussão sobre a aplicação de técnicas de busca para solucionar problemas da Engenharia de Software. No contexto do workshop, técnicas de busca englobam tanto técnicas tradicionais, como força bruta ou branch-and-bound, quanto meta-heurísticas, como algoritmos genéticos e outros algoritmos bio-inspirados.

O WESB é um workshop sobre fundamentos teóricos e experiências práticas da Engenharia de Software Baseada em Busca (SBSE - Search Based Software Engineering) em projetos acadêmicos e industriais. O objetivo é a troca de experiências, opiniões e debates entre os participantes.

Os trabalhos submetidos para esta quarta edição foram cuidadosamente revisados por três avaliadores do comitê de programa, que contou com pesquisadores de diferentes regiões do país, e com a colaboração de alguns revisores externos. Estes anais contêm os trabalhos selecionados dentre todas estas submissões. São 9 artigos completos e 3 resumos estendidos (RE). Os principais temas abordados incluem: otimização de conflitos de merge, teste de software, projeto de software, Linha de Produto de Software, planejamento do projeto de software, requisitos, e o mapeamento da área de SBSE no Brasil.

Gostaríamos de parabenizar a todos os autores dos trabalhos selecionados e também agradecer aos autores de todas as submissões realizadas.

Agradecemos especialmente a todos os membros do comitê de programa e aos eventuais revisores por eles elencados. Sabemos que sem esta valiosa colaboração não teríamos conseguido concluir o processo de revisão no tempo previsto. Agradecemos também aos organizadores do CBSoft 2013 pela infraestrutura disponibilizada e pela oportunidade oferecida.

Desejamos um excelente evento a todos e que o WESB-2013 contribua para ampliar e consolidar a área de Engenharia de Software Baseada em Busca no Brasil.

Silvia R. Vergilio

DInf -UFPR, Coordenadora Geral

BIOGRAFIA DOS COORDENADORES / FEES 2013

CHAIRS SHORT BIOGRAPHIES

ARILO CLAUDIO DIAS NETO, UFAM

Bacharel em Ciência da Computação pela UFAM em 2004, Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ em 2006, Doutor em Engenharia de Sistemas e Computação pela COPPE/UFRJ em 2009. Atualmente, é Professor Adjunto do Instituto de Computação da Universidade Federal do Amazonas (IComp/UFAM) e Professor Permanente do Programa de Pós-Graduação em Informática (PPGI) da UFAM. Tem atuado nos últimos 9 anos em pesquisas científicas e projetos de desenvolvimento na área de Qualidade de Software, principalmente Teste de Software, Engenharia de Software Experimental, Engenharia de Software Baseada em Busca e Desenvolvimento Distribuído de Software. Atua como revisor do Journal of the Brazilian Computer Society (JBCS), do International Journal of Information and Software Technology (IST).

GLÊDSON ELIAS, UFPB

Doutor em Ciência da Computação na área de Engenharia de Software pelo CIn/UFPE (2002). Iniciou a carreira acadêmica em 1993, e, atualmente, é Professor do Programa de Pós-Graduação em Informática do DI/UFPB. Suas principais áreas de interesse são: Engenharia de Software Baseada em Buscas, Desenvolvimento Distribuído de Software, Reutilização de Software, Middleware e Computação Móvel. Atualmente coordena e participa de projetos de pesquisa relacionados a sistemas de recomendação baseados em buscas meta-heurísticas para agrupamentos de componentes e alocação de equipes de desenvolvimento distribuídas. Foi coordenador geral do WDBC 2004, SBES 2007 e WDDS 2007; coordenador do comitê de programa do WDBC 2004, WDDS 2008, e Sessão de Ferramentas SBES 2009; e membro do comitê diretivo do SBCARS e WDDS. Tem participado como membro do comitê de programa de diversos eventos nacionais e internacionais.

JERFFESON TEIXEIRA DE SOUZA, UECE

Recebeu o título de Ph.D. em Ciência da Computação pela School of Information Technology and Engineering (SITE) da University of Ottawa, Canadá. É professor adjunto da Universidade Estadual do Ceará (UECE), já tendo trabalhado como Coordenador do Mestrado Acadêmico em Ciência da Computação da UECE (MACC). Atualmente é Coordenador do Grupo de Otimização em Engenharia de Software da UECE (GOES.UECE). É ainda membro nominado do Hillsides Group e membro do International Editorial Board do International Journal of Patterns (IJOP). Trabalhou ainda como Co-presidente da SugarLoafPLoP 2004, Co-presidente do Comitê de Programa do SugarLoafPLoP 2007 e Presidente do SugarLoafPLoP 2008. Seus interesses de pesquisa são: Otimização em Engenharia de Software, Documentação e Aplicação de Padrões de Software e Estudo de Técnicas e Aplicação de Algoritmos de Mineração de Dados.

SILVIA REGINA VERGILIO, UFPR

Bacharel em Ciência da Computação pelo ICMSC/USP em 1989, Mestre em Engenharia de Sistemas e Computação pela DCA/Unicamp em 1991, Doutora em Engenharia de Sistemas e Computação pelo DCA/Unicamp em 1997. Atualmente é Professora Associada da Universidade Federal do Paraná (UFPR). Seus interesses de pesquisa em Engenharia de Software são: Otimização em Engenharia de Software, Computação Evolutiva, Teste e Validação de Software, Aplicações Web. Participa em comitês de diferentes eventos nacionais e internacionais.

COMITÊS TÉCNICOS / TECHNICAL COMMITTEES

COMITÊ DE PROGRAMA / PROGRAM COMMITTEE

Adriana Cesário de Faria Alvim, UNIRIO
Arilo Claudio Dias Neto, UFAM
Auri Vincenzi, UFG
Celso G. Camilo-Junior, UFG
Eliane Martins, IC/Unicamp
Geraldo Robson Mateus, UFMG
Glêdson Elias, UFPB
Gustavo Augusto Lima de Campos, UECE
Jerffeson Teixeira de Souza, UECE
Leila Silva, UFS
Maria Claudia Figueiredo Pereira Emer, UTFPR
Márcio de Oliveira Barros, UNIRIO
Mariela Inés Cortés, UECE
Mel Ó Cinnéide, University College Dublin, IE
Pedro de Alcântara dos Santos Neto, UFPI
Phil McMinn, University of Sheffield, UK
Rosiane de Freitas Rodrigues, UFAM
Silvia Regina Vergilio, UFPR

REVISORES DE TRABALHOS / REVIEWERS

Adriana C. F. Alvim
Arilo Claudio Dias Neto
Auri Vincenzi
Cassio Leonardo Rodrigues
Celso G. Camilo-Junior
Daniela Cascini
Eliane Martins
Fabio de Almeida Farzat
Glêdson Elias
Gustavo Augusto Lima de Campos
Jackson Barbosa
Jerffeson Teixeira de Souza
Leila Silva
Maria Claudia Figueiredo Pereira Emer
Márcio de Oliveira Barros
Mariela Inés Cortés
Pedro de Alcântara dos Santos Neto
Ricardo de Souza Brito
Ricardo Rabêlo
Rosiane de Freitas Rodrigues
Silvia Regina Vergilio

COMITÊ ORGANIZADOR / ORGANIZING COMMITTEE

COORDENAÇÃO GERAL

Genaína Nunes Rodrigues, CIC, UnB

Rodrigo Bonifácio, CIC, UnB

Edna Dias Canedo, CIC, UnB

COMITÊ LOCAL

Diego Aranha, CIC, UnB

Edna Dias Canedo, FGA, UnB

Fernanda Lima, CIC, UnB

Guilherme Novaes Ramos, CIC, UnB

Marcus Vinícius Lamar, CIC, UnB

George Marsicano, FGA, UnB

Giovanni Santos Almeida, FGA, UnB

Hilmer Neri, FGA, UnB

Luís Miyadaira, FGA, UnB

Maria Helena Ximenis, CIC, UnB

PALESTRA CONVIDADA / INVITED KEYNOTE

APLICAÇÕES DE ALGORITMOS EVOLUTIVOS MULTIOBJETIVOS EM ENGENHARIA DE SOFTWARE

Muitos problemas da área de Engenharia de Software podem ser formulados como problemas de otimização multiobjetivo. Por exemplo, o projeto de um software enfrenta demandas de qualidade, baixo custo, maior segurança, e assim por diante. Portanto, nestes problemas, muitos objetivos precisam ser tratados simultaneamente. Esta palestra tem como objetivo discutir a evolução dos principais desenvolvimentos no campo dos Algoritmos Evolutivos Multiobjetivos desde a sua criação no início dos anos noventa, rever os algoritmos básicos, e apresentar aplicações destes algoritmos em Engenharia de Software.

AURORA TRINIDAD RAMIREZ POZO

Possui graduação em Engenharia Elétrica - Universidad de Concepcion (1985), mestrado em Engenharia Elétrica pela Universidade Federal de Santa Catarina (1991) e doutorado em Engenharia Elétrica pela Universidade Federal de Santa Catarina (1996). Atualmente é professora Associado da Universidade Federal do Paraná (desde 1997). Tem experiência na área de Ciência da Computação, com ênfase em Inteligência Computacional, atuando principalmente nos seguintes temas: Vida Artificial (Computação Evolutiva, Nuvem de Particula, etc..) e aplicações em Engenharia de Software e Mineração de Dados. Bolsista de Produtividade em Pesquisa do CNPq - Nível 2.

ÍNDICE DE ARTIGOS / TABLE OF CONTENTS

PALESTRA CONVIDADA

APLICAÇÕES DE ALGORITMOS EVOLUTIVOS MULTIOBJETIVOS EM ENGENHARIA DE SOFTWARE

9

Aurora Trinidad Ramirez Pozo

SESSÃO TÉCNICA 1: PLANEJAMENTO DO PROJETO DE SOFTWARE E REQUISITOS

MODELANDO O PROBLEMA DA PRÓXIMA RELEASE SOB A PERSPECTIVA DA ANÁLISE DE PONTOS DE FUNÇÃO

12

Vitor Gonçalves, Márcio Barros

UMA ADAPTAÇÃO DO ALGORITMO GENÉTICO PARA O PROBLEMA DO PRÓXIMO RELEASE COM INTERDEPENDÊNCIAS ENTRE REQUISITOS

22

Italo Bruno, Matheus Paixão, Jerffeson Souza

UMA ABORDAGEM MULTIOBJETIVO PARA O PROBLEMA DE DESENVOLVIMENTO DE CRONOGRAMAS DE PROJETOS DE SOFTWARE

32

Sophia Nobrega, Sérgio R. de Souza, Marcone Souza

A FRAMEWORK FOR SELECTION OF SOFTWARE TECHNOLOGIES USING SEARCH-BASED STRATEGIES (RE)

42

Aurélio Grande, Rosiane de Freitas, Arilo Dias Neto

SESSÃO TÉCNICA 2: TESTE DE SOFTWARE, MAPEAMENTO DA COMUNIDADE DE SBSE, ESTUDOS EXPERIMENTAIS

MAPEAMENTO DA COMUNIDADE BRASILEIRA DE SBSE

46

Wesley Klewerton Guez Assunção, Márcio Barros, Thelma Colanzi, Arilo Dias Neto, Matheus Paixão, Jerffeson Souza, Silvia Regina Vergilio

UM ALGORITMO GENÉTICO COEVOLUCIONÁRIO COM CLASSIFICAÇÃO GENÉTICA CONTROLADA APLICADO AO TESTE DE MUTAÇÃO

56

André Oliveira, Celso Camilo-Junior, Auri Marcelo Rizzo Vincenzi

SELEÇÃO DE PRODUTO BASEADA EM ALGORITMOS MULTIOBJETIVOS PARA O TESTE DE MUTAÇÃO DE VARIABILIDADES	66
Édipo Luis Féderle Féderle, Giovani Guizzo, Thelma Colanzi, Silvia Regina Vergilio, Eduardo Spinosa	
UM ESTUDO COMPARATIVO DE HEURÍSTICAS APLICADAS AO PROBLEMA DE CLUSTERIZAÇÃO DE MÓDULOS DE SOFTWARE (RE)	76
Alexandre Pinto, Márcio Barros, Adriana Cesário de Faria Alvim	
SESSÃO TÉCNICA 3: PROJETO DE SOFTWARE E LINHA DE PRODUTO DE SOFTWARE	
MODELO PARA APOIAR A CONFIGURAÇÃO E GERÊNCIA DE VARIABILIDADE EM LPS	80
Juliana Pereira, Eduardo Figueiredo, Thiago Noronha	
OTIMIZANDO ARQUITETURAS DE LPS: UMA PROPOSTA DE OPERADOR DE MUTAÇÃO PARA A APLICAÇÃO AUTOMÁTICA DE PADRÕES DE PROJETO	90
Giovani Guizzo, Thelma Colanzi, Silvia Regina Vergilio	
A CAMINHO DE UMA ABORDAGEM BASEADA EM BUSCAS PARA MINIMIZAÇÃO DE CONFLITOS DE MERGE	100
Gleiph Menezes, Leonardo Murta, Márcio Barros	
USANDO HILL CLIMBING PARA IDENTIFICAÇÃO DE COMPONENTES DE SOFTWARE	110
Similares (RE)	
Antonio Ribeiro dos Santos Júnior, Leila Silva, Glêbson Elias	

Modelando o Problema da Próxima Release sob a Perspectiva da Análise de Pontos de Função

Vitor Padilha Gonçalves, Márcio de O. Barros

Programa de Pós-Graduação em Informática – Universidade Federal do Estado do Rio de Janeiro (UNIRIO) – Rio de Janeiro – RJ – Brazil

{vitor.padilha,marcio.barros}@uniriotec.br

Abstract. *The next release problem (NRP) has been widely discussed, considering that software requirements have fixed value. As the analysis of function points is a nonlinear method for measuring the effort required to implement the requirements and is used as a basis for estimating costs and time, which are the main variables for the distribution of releases of a software project, this paper presents a new solution to the NRP based on this method. Thus, concepts of the method will be shown, the problem will be formally described and a study to validate the proposal will be shown.*

Resumo. *O problema da próxima release (NRP) tem sido bastante discutido, porém com uma proposta na qual os requisitos de software possuem um valor fixo e indivisível. Como a análise de pontos de função é um método não linear para medição dos valores dos requisitos e é utilizada como base para estimativas de custos e prazos, sendo estas as principais variáveis para distribuição das releases de um projeto de software, este trabalho apresentará uma nova proposta de solução do NRP com base neste método. Desta forma, conceitos do método serão mostrados, o problema será formalizado e um estudo feito para validação da proposta será apresentado.*

1. Introdução

No contexto de contratação de serviços para desenvolvimento de software, usualmente o custo e o tempo necessários para entrega do produto são fatores relevantes. Estes fatores estão diretamente ligados ao esforço necessário para construção do sistema. Este esforço, por outro lado, é dependente dos requisitos desejados pelos patrocinadores do projeto. Muitas vezes, o esforço requerido para desenvolver um software é suficientemente grande para justificar seu particionamento em diversas versões (*releases*), de modo a atender às restrições financeiras e a data de entrega de alguns dos requisitos [Del Sagrado et al. 2011]. Com o objetivo de aumentar a satisfação dos patrocinadores, os requisitos mais importantes devem ser priorizados. A escolha dos requisitos que serão incluídos em uma versão do software é comumente chamada de “problema da próxima release” (NRP, sigla em inglês para *next release problem*) [Bagnall et al. 2001; Del Sagrado et al. 2011; Durillo et al. 2010; Zhang et al. 2013].

A impossibilidade de investigar todas as possíveis combinações de requisitos, mesmo em um problema de pequena escala, levou diversos autores a propor abordagens baseadas em buscas heurísticas para encontrar boas soluções para este problema [Bagnall et al. 2001; Durillo et al. 2010; Li et al. 2010; Souza et al. 2011; Tonella et al. 2013]. Estas soluções indicam um conjunto de requisitos que tenta maximizar a satisfação dos patrocinadores, enquanto atendem a restrições de custo e\ou prazo. Em

grande parte destas propostas, os autores atribuem um valor fixo ao esforço (ou custo) necessário para o desenvolvimento de cada requisito. Alguns trabalhos [Del Sagrado et al. 2011; Zhang et al. 2013] consideram relações de interdependência entre requisitos, de modo que determinado requisito pode afetar e/ou ter seu esforço (ou custo) de desenvolvimento afetado pela presença (ou ausência) de outros requisitos na mesma versão do software. A seleção do subconjunto ideal de requisitos a ser implementado na próxima versão de um software é um problema NP-completo mesmo quando não existem interdependências entre os requisitos [Bagnall et al. 2001].

Conforme o Guia Prático para Contratação de Soluções de Tecnologia da Informação [SLTI 2011] do Ministério do Planejamento, Orçamento e Gestão do Governo brasileiro, a contratação de serviços de TI por empresas e instituições do Governo depende de uma análise de custos e prazos apresentados em uma licitação. Como o esforço para desenvolvimento de um sistema é proporcional a estes dois fatores, a contratação de serviços de TI pelo Governo dependerá, em última análise, de uma estimativa do esforço. O esforço necessário para o desenvolvimento de um software pode ser estimado através de diversas técnicas [Boehm et al. 2000; COSMIC 2003; IFPUG 2009], entre elas a análise pontos de função (APF) [Ahn et al. 2003; Ferreira e Hazan 2010; Matson et al. 1994]. O governo brasileiro, através de recomendação do TCU, tem utilizado a APF [IFPUG 2009] como base para contratação de serviços de desenvolvimento de software para empresas e instituições públicas [Ferreira e Hazan 2010]. Uma vantagem de utilizar a APF é a flexibilidade de mudar o escopo ao longo do projeto: as empresas compram um volume de esforço para desenvolvimento de sistemas e não os sistemas propriamente ditos, permitindo mudanças nos requisitos e não ficando limitadas a um conjunto de necessidades iniciais.

O cálculo de esforço com a APF depende de informações que podem ser extraídas dos requisitos funcionais do sistema. O esforço é proporcional ao número de uma unidade abstrata, chamada de *ponto de função*. O cálculo do número de pontos de função depende do número e complexidade de transações e agrupamentos de dados de um software, sendo esta complexidade determinada por tabelas de referência baseadas em dados que descrevem as transações e agrupamentos. Estas tabelas são caracterizadas por intervalos discretos, de modo que um pequeno aumento na complexidade de uma transação não necessariamente aumenta a estimativa de esforço para o seu desenvolvimento. O uso destes intervalos discretos cria não-linearidades na estimativa de esforço, que podem ser exploradas pela busca heurística usada para resolver o NRP.

O objetivo deste trabalho é utilizar da Engenharia de Software baseada em Buscas juntamente com APF como proposta de solução ao NRP. Com isto, será definida uma metodologia para a definição das *releases* de um projeto de software, a fim de atender restrições de custo e prazo. Para validação do método, um estudo será feito para avaliar o impacto da não-linearidade apresentada pela APF no NRP. Sendo assim, esperamos desenvolver um método de apoio à contratação de serviços de desenvolvimento de software por empresas públicas, permitindo eventual contenção de gastos e uma maior autonomia nas mudanças de requisitos no decorrer do projeto. Para isto, o NRP baseado em APF será formalizado e um estudo do problema com base em um sistema de gestão de pessoal será apresentado para análise dos resultados.

Este artigo está dividido nos seguintes tópicos: a seção 2 apresentará os conceitos da análise de pontos de função (APF); a seção 3 formalizará o problema da

NRP no contexto da APF; a seção 4 mostrará um exemplo de aplicação do problema em um sistema de recursos humanos, e; na seção 5 será apresentado a conclusão e trabalho futuros.

2. Análise de Pontos de Função

A contagem de pontos de função consiste em contar funções de dados e de transações. O manual do IFPUG (2009) define as funções de dados como “funcionalidade de dados que satisfaz os Requisitos Funcionais do Usuário referentes a armazenar e/ou referenciar dados” e as funções de transação como “processo elementar que fornece funcionalidade ao usuário para processar dados”. Neste sentido, é possível observar que uma função de dados é decorrente de uma necessidade das funções de transação para processamento. Com isto, só faz sentido determinada função de dados existir se ela for utilizada por alguma função de transação. Caso contrário, estaríamos contabilizando um elemento que não possui nenhuma utilidade aos usuários finais.

As funções de dados podem ser classificadas como ALI (arquivo lógico interno) ou AIE (arquivo de interface externa). Um ALI é definido como “grupo de dados ou informações de controle logicamente relacionados, reconhecido pelo usuário, mantido dentro da fronteira da aplicação”, ou seja, refere-se a informações que podem ser incluídas ou alteradas pela própria aplicação. Já um AIE é “grupo de dados ou informações de controle logicamente relacionados, reconhecido pelo usuário, referenciado pela aplicação, porém mantido dentro da fronteira de outra aplicação”, ou seja, são dados ou informações que são somente lidas pela aplicação a ser desenvolvida.

Cada função de dados é caracterizada por dois elementos: dados elementares referenciados (DER) e registros lógicos referenciados (RLR). Segundo IFPUG, um DER é “atributo único, reconhecido pelo usuário e não repetido” e um RLR é “um subgrupo de dados elementares referenciados, reconhecido pelo usuário, contido em uma função de dados”. Ou seja, um DER está contido em um RLR, que está contido em uma função de dados. A complexidade da função de dados é proporcional ao número de DER e RLR que ela possui. A Tabela 1 apresenta esta relação.

Tabela 1. Complexidade das funções de dados

DER RLR	1 - 19	20 – 50	> 50
1	Baixa	Baixa	Média
2 – 5	Baixa	Média	Alta
> 5	Média	Alta	Alta

O número de funções de dados atribuídos a uma função de dados dependerá da complexidade da função e do seu tipo (AIE ou ALI), como apresentado na Tabela 2.

Tabela 2. Tamanho das funções de dados

Complexidade	Tipo	
	ALI	AIE
Baixa	7	5
Média	10	7
Alta	15	10

A contagem das funções de transação segue o mesmo procedimento das funções de dados, definindo os tipos e calculando a complexidade das mesmas. As funções transacionais são classificadas como CE (consulta externa), EE (entrada externa) ou SE

(saída externa). Para definição do tipo de uma função transacional é preciso observar: (1) se é um processo que apenas consulta uma ou mais funções de dados sem nenhum processamento; (2) se é um processo de consulta com algum processamento; ou (3) se é um processo que processa dados para alimentar um ou mais ALI. No primeiro caso, a função transacional é denominada consulta externa (CE); no segundo, ela é uma saída externa (SE) e, no último caso, é uma entrada externa (EE).

Tabela 3. Complexidade das funções de dados

EE				SE e CE			
DER \ ALR	1 - 4	5 - 15	> 15	DER \ ALR	1 - 5	6 - 19	> 19
0 - 1	Baixa	Baixa	Média	0 - 1	Baixa	Baixa	Média
2	Baixa	Média	Alta	2 - 3	Baixa	Média	Alta
> 2	Média	Alta	Alta	> 3	Média	Alta	Alta

O cálculo da complexidade de uma função de transação é feito a partir do número de arquivos lógicos referenciados (ALR) e de dados elementares referenciados (DER) pela mesma, conforme mostra a Tabela 3. Um ALR é definido como “função de dados lida e/ou mantida por uma função transacional”. O DER segue a mesma definição usada nas funções de dados. Com a complexidade e o tipo de uma função de transação, é possível definir o seu número de pontos de função, como apresentado na Tabela 4.

Tabela 4. Tamanho das funções de transação

Complexidade \ Tipo	EE	SE	CE
Baixa	3	4	3
Média	4	5	4
Alta	6	7	6

Com as funções de dados e de transação medidas, faz-se o somatório dos pontos atribuídos as mesmas para obter o número de pontos de função do sistema. Este pode ser utilizado como medida de esforço para desenvolvimento do projeto de software.

3. Formalizando a Seleção das Funções de Dados e de Transação a Partir de um Subconjunto de Transações Escolhidas

Diversos artigos que tratam o problema da priorização de requisitos (Durillo et al., 2010; Li et al., 2010; Sagrado et al., 2011; Tonella et al., 2013) mostram que é preciso administrar a relação entre satisfação dos patrocinadores e requisitos atendidos. Neste sentido, é necessário priorizar os requisitos que são mais importantes para determinado conjunto de patrocinadores a fim de maximizar a sua satisfação. As diferenças entre trabalhos relacionados à NRP dependem do objetivo a ser otimizado, das características do problema a serem estudadas e do tipo de dependências entre os requisitos de software. Para formalização de nossa visão do problema, adotaremos as definições propostas por Bagnall et al. (2001), Zhang et al. (2013) e Durillo et al. (2010).

A definição clássica do NRP está apresentada a seguir. Considerando um conjunto de requisitos $R = \{r_1, r_2, \dots, r_n\}$, cada requisito r_i é descrito por um custo c_i , positivo e maior que zero, e possui um conjunto de precedências, que são os requisitos que devem ser implementados antes de r_i . Considerando $S = \{s_1, s_2, \dots, s_m\}$ o conjunto de todos os patrocinadores do sistema, cada patrocinador s_j possui um grau de interesse v_{ij} sobre cada requisito $r_i \in R$ e está associado a um peso p_j , que retrata a importância do

patrocinador na empresa. Neste sentido, é preciso encontrar um subconjunto de requisitos $R' \subseteq R$ que maximize a satisfação dos patrocinadores e minimize o custo de desenvolvimento da *release*. Sendo assim, as funções que definem o problema são:

$$\text{Minimizar } f(1) = \sum_{i \in R} c_i \quad (1)$$

$$\text{Maximizar } f(2) = \sum_{j=1}^m p_j \sum_{i \in R'} v_{ij} \quad (2)$$

Os diferenciais da nossa proposta em relação à formulação clássica do NRP são: (1) considerar as dependências entre os elementos da APF, conforme mostrado na seção 2; (2) utilizar a APF como base para o cálculo do custo dos requisitos; e (3) considerar as funções de dados e transacionais de acordo com os interesses dos patrocinadores pelos requisitos (v_{ij} , na definição padrão) no contexto da APF. Neste sentido, a seleção de requisitos começa com um conjunto de funções de dados e transação representando todo o escopo de software, ou seja, todos os requisitos desejados pelos patrocinadores. Para isto, a Tabela 5 sumariza a notação dos elementos envolvidos na APF.

Tabela 5. Tabela de Notações dos Elementos de APF

Notação	Tam.	Definição	Representação
S	k	Conjunto de patrocinadores envolvidos no projeto	$\{s_1, s_2, \dots, s_k\}$
T	n	Conjunto das funções de transação do software	$\{t_1, t_2, \dots, t_n\}$
D	m	Conjunto das funções de dados do software	$\{d_1, d_2, \dots, d_m\}$
$tipo(d_i)$	-	Tipo da função de dados d_i	ALI ou AIE
$RLR(d_i)$	p_i	Conjunto das RLR de uma função de dados d_i	$\{rl_{i1}, \dots, rl_{ip_i}\}$
$DER(rl_i)$	q_i	Conjunto dos DER de um RLR rl_i	$\{dr_{i1}, \dots, dr_{iq_i}\}$
$tipo(t_i)$	-	Tipo da função de transação t_i	EE ou SE ou CE
$DER(t_i)$	r_i	Conjunto de DET referenciados por t_i	$\{dr_{i1}, \dots, dr_{ir_i}\} \therefore dr_j \in \bigcup_{d \in D} \bigcup_{rl \in RLR(d)} DER(rl)$
$PR(t_i)$	-	Conjunto de transações que precedem a transação t_i	$\{t_x, \dots, t_y\}$
v_{ij}	-	Interesse de um patrocinador s_j na transação t_i	-
p_i	-	Importância de um patrocinador s_i na empresa	-

A partir das notações acima é possível representar o número de pontos de função de uma função de dados $d_i \in D$ com base no seu tipo, seus RLR e os DER mantidos por estes RLR. O conjunto dos DER de d_i é calculado pela equação abaixo.

$$DER(d_i) = \bigcup_{rl \in RLR(d_i)} DER(rl)$$

O número de pontos de função de d_i é representado por $PFD(tipo(d_i), RLR(d_i), DER(d_i))$ e calculado conforme as tabelas apresentadas na seção 2. A fórmula abaixo representa o número total de pontos de função do software.

$$PDF = \sum_{d_i \in D} PFD(tipo(d_i), RLR(d_i), DER(d_i))$$

De forma similar, é possível representar o número de pontos de função de uma função de transação $t_i \in T$ com base no seu tipo, as funções de dados referenciadas por ela (chamada ALR) e os DER usados por esta função de transação. O conjunto de funções de dados referenciadas por t_i , $ALR(t_i)$, é calculado pela equação abaixo.

$$ALR(t_i) = \{d_i \in D : \exists rl_i \in RLR(d_i) \rightarrow DET(rl_i) \cap DET(t_i) \neq \emptyset\}$$

Com isto, o número de pontos de função de t_i é representado por $PFT(tipo(t_i), ALR(t_i), DER(t_i))$ e também é calculado conforme a tabelas apresentadas na seção 2. O número de pontos de função das funções de transação do software é dado abaixo.

$$PFT = \sum_{t_i \in T} PFT(tipo(t_i), ALR(t_i), DER(t_i))$$

Portanto, o número de pontos de função do software é dado pela fórmula abaixo.

$$PF = PDF + PFT$$

Conforme apresentado na seção 2, na modelagem da APF o elemento funcional para os patrocinadores é a função de transação, ou seja, é nela que o patrocinador tem interesse, sendo a função de dados decorrente da necessidade de armazenamento e/ou leitura de dados por uma ou mais funções de transação. Assim, para o particionamento do software em *releases* tomaremos um subconjunto de T , representado por T^i , referente às funções de transação que serão consideradas na *release*.

Na modelagem da APF, as dependências podem ocorrer entre duas funções de transação e entre uma função de dados e uma função de transação. O primeiro tipo de dependência tem como efeito, que ao se escolher o subconjunto T^i de funções de transação para uma *release*, é preciso identificar que outras funções de transação devem ser incluídas nesta *release* através de uma análise de precedência. Isto acontece porque uma função de transação pode ser funcionalmente dependente de uma ou mais funções de transações, ou seja, ela pode utilizar recursos providos por outras transações e ficar inviável sem que elas sejam previamente (ou conjuntamente) desenvolvidas. Por exemplo, a função de transação “Realizar Venda de Produto” é dependente da função de transação “Cadastrar Produto”, pois não é possível vender produtos que não sejam reconhecidos pelo sistema. Este é o tipo mais comum de dependência entre requisitos, abordado em trabalhos anteriores [Bagnall et al. 2001; Zhang et al. 2013].

Quanto à dependência entre uma função de dados e uma função de transação, o manual de contagem de pontos de função [IFPUG 2009] indica que é necessário contabilizar quais funções de dados (ALR) e quais de seus dados (DER) uma função de transação utiliza para determinar seu tamanho em pontos de função. Neste caso, a escolha das funções de dados para uma *release* depende das funções de transação escolhidas, ou seja, somente as funções de dados referenciadas por pelo menos uma função de transação escolhida serão consideradas. O mesmo ocorre com os DER de uma função de dados: somente os DER referenciados por alguma transação serão considerados como elementos da função de dados na *release*.

Por conta dos dois tipos de dependência, ao selecionar o subconjunto T^i é preciso: (1) adicionar as funções de transação que precedem, diretamente e indiretamente, as transações selecionadas; (2) encontrar os DER referenciados pelas funções de transação selecionadas e adicionadas; (3) encontrar as funções de dados e os RLR nas quais estes DER estão contidos; (4) contar o número de pontos de função das funções de dados considerando apenas os DER e RLR utilizados pelas funções de transação selecionadas; e (5) calcular o número de pontos de função final para a *release*.

Após a etapa (5), teremos o número total de pontos de função do software que, multiplicado pelo custo unitário do ponto de função, gerará o custo total da *release*, que pode substituir a equação (1) da formulação clássica do NRP. É importante perceber que a otimização deste custo não é simples – uma transação retirada do escopo pode tornar diversos DER desnecessários, fazendo com que a complexidade da sua função de dados seja reduzida e, consequentemente, o custo da *release* também o seja. Assim, consideraremos as não-linearidades do método de pontos de função, calculando o custo dos requisitos como uma caixa-branca.

Considerando o conjunto de funções de transação selecionadas T^i , o conjunto de funções de transação que precedem diretamente T^i é calculado pela equação abaixo.

$$precDiretos(T^i) = \bigcup_{t \in T^i} PR(t)$$

O conjunto de todas as funções de transação precedentes do conjunto T^i é definido recursivamente pela fórmula abaixo.

$$precedentes(T^i) = precDiretos(T^i) \cup precedentes(precDiretos(T^i))$$

Finalmente, T^f , que representa o conjunto de todas as funções de transação que devem ser consideradas na *release* quando escolhido o subconjunto T^i , é:

$$T^f = T^i \cup precedentes(T^i)$$

Para melhor entendimento, a Figura 1 ilustra três subconjuntos, onde cada elipse representa uma transação, sendo que as tracejadas não estão selecionadas. Sabendo que as setas (\rightarrow) representam a precedência das transações, temos: (a) o lado esquerdo representa o conjunto T^i ; (b) o meio, $precedentes(T^i)$; e (c) o direito, T^f . Sendo assim, $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, $PR(t_1) = \{t_6\}$, $PR(t_2) = \{t_6\}$, $PR(t_3) = \emptyset$, $PR(t_4) = \{t_6\}$, $PR(t_5) = \emptyset$, $PR(t_6) = \emptyset$, $T^i = \{t_2, t_4\}$, $precedentes(T^i) = \{t_1, t_4, t_5, t_6\}$ e $T^f = \{t_1, t_2, t_4, t_5, t_6\}$.

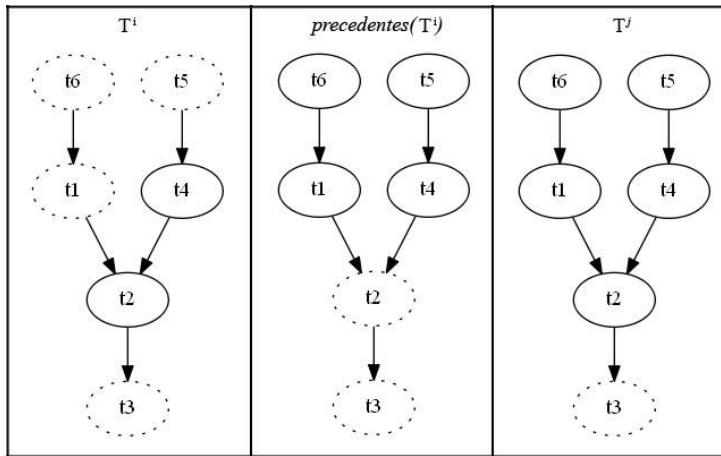


Figura 1. Conjuntos T^i , $precedentes(T^i)$ e T^f

Selecionado o subconjunto de transações T^f , é preciso encontrar os DER utilizados por este subconjunto (DR), conforme a equação abaixo.

$$DR'(T^f) = \bigcup_{t_i \in T^f} DER(t_i)$$

Em seguida, é preciso selecionar os RLR que mantém os DER referenciados pelas funções de transação. Estes RLR são representados pela equação RL abaixo. Estes RLR vão manter um subconjunto dos seus DER originais (DER'), uma vez que DER não utilizados pelas funções de transação devem ser desconsiderados.

$$RL(T^f) = \{d_i \in D : \exists rl_i \in RLR(d_i) \rightarrow DER(rl_i) \cap DR(T^f) \neq \emptyset\}$$

$$DER'(rl_i, T^f) = DER(rl_i) \cap DR(T^f)$$

Finalmente, precisamos selecionar as funções de dados que mantém os RLR que compõem o conjunto $RL(T^f)$. Isto é feito pela equação D' abaixo. Estas funções de dados vão manter um subconjunto dos seus RLR originais (RLR'), uma vez que RLR que não mantenham dados utilizados pelas funções de transação devem ser descartados.

$$D'(T^f) = \{d_i \in D : RLR(d_i) \cap RL(T^f) \neq \emptyset\}$$

$$RLR'(d_i, T^f) = RLR(d_i) \cap RL(T^f)$$

$$DER'(d_i, T^f) = DER(d_i) \cap DR(T^f)$$

Com T^f , $D'(T^f)$, $RL(T^f)$ e $DR(T^f)$ é possível obter o total de pontos de função para a *release*. Para isto, é preciso calcular PFD' e PFT' referentes, respectivamente, ao número de pontos de função das funções dados e das funções de transação consideradas na *release*. Abaixo, apresentamos as representações destes cálculos.

$$PFD' = \sum_{d_i \in D'(T^f)} PFD(tipo(d_i), RLR'(d_i, T^f), DER'(d_i, T^f))$$

$$PFT' = \sum_{t_i \in T^f} PFT(tipo(t_i), ALR(t_i), DER(t_i))$$

A Figura 2 representa a utilização de uma função de dados d_1 (neste caso, um ALI), pelas funções de transação t_2 e t_3 e considera $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ e $T^f = \{t_1, t_2, t_4, t_5, t_6\}$. Sendo assim, $PFD(tipo(d_1), RLR(d_1), DER(d_1)) = 15$, pois $|RLR(d_1)| = 4$ e $|DER(d_1)| = 23$, e, $PFD(tipo(d_1), RLR'(d_1, T^f), DER'(d_1, T^f)) = 10$, pois $|RLR'(d_1, T^f)| = 2$ e $|DER'(d_1, T^f)| = 9$. Este exemplo demonstra que a retirada de uma função de transação pode provocar uma redução não-linear no custo da *release*.

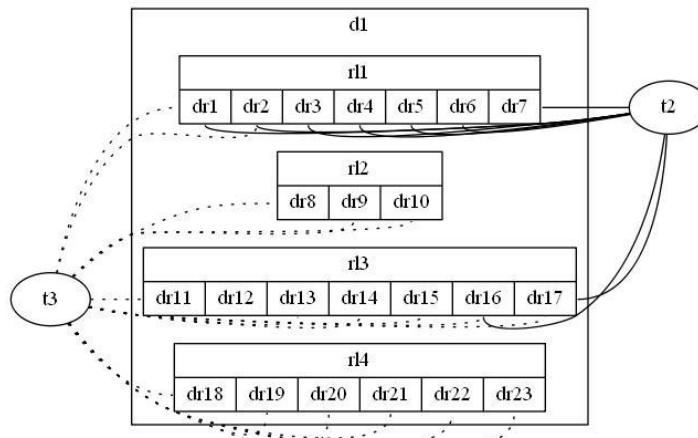


Figura 2. Exemplo: Utilização de d_1

A partir de PFD' e PFT' é possível obter o custo total, em pontos de função, para a *release*. Neste sentido, o problema da NRP sob a perspectiva da APF é representado pelas funções a seguir:

$$\text{Minimizar } f(1) = PFD' + PDT' \quad (1)$$

$$\text{Maximizar } f(2) = \sum_{j=1}^k p_j \sum_{i \in T^f} v_{ij} \quad (2)$$

4. Exemplo Prático

Este artigo apresenta um trabalho em andamento. Para avaliação da proposta apresentada na seção 3, foi realizada uma prova de conceito baseada em um sistema para gestão de pessoas. A contagem inicial do projeto consiste em 65 funções de transação (221 PF) e 9 funções de dados (67 PF). Sendo assim, o sistema apresenta um total de 288 pontos de função. Considerando quatro *releases* que poderão consistir em no máximo 96 pontos de função cada, um algoritmo genético mono-objetivo foi

programado para maximizar a satisfação dos patrocinadores. A partir dos resultados obtidos faremos uma avaliação inicial da proposta.

Para o algoritmo, foram feitas 50 execuções independentes. A execução foi configurada para uma população inicial de 130 indivíduos, um número máximo de 16.900 avaliações (o número de indivíduos ao quadrado) e com os seguintes operadores: *single-point crossover* com probabilidade de 80%, mutação *bit-flip* com probabilidade de 2% e seleção por torneio. Os dados obtidos após as execuções são apresentados na Tabela 6.

Tabela 6. Resultados Execuções do Algoritmo Genético

<i>Release</i>	Informação	Media	Mínimo	Máximo
1 ^a	Pontos de função	94,72 ± 1,21	93,0	96,0
	Satisfação (%)	44,90 ± 0,60	44,6	45,9
2 ^a	Pontos de função	190,3 ± 1,34	187,0	192,0
	Satisfação (%)	75,51 ± 0,40	74,6	75,9
3 ^a	Pontos de função	285,38 ± 1,70	281,0	288,0
	Satisfação (%)	99,64 ± 0,22	99,09	100,0
4 ^a	Pontos de função	288,0 ± 0	288,0	288,0
	Satisfação (%)	100,0 ± 0	100,0	100,0
	Tempo de execução (seg)	44,4 ± 0,37	43,8	46,0

É possível observar, a partir da tabela acima, que a primeira *release*, apesar de medir 1/3 dos pontos de função inicial, atende em média a quase 45% da satisfação dos patrocinadores. Observa-se também que para a terceira *release*, as funções de transação escolhidas atenderam, em média, apenas 24% da satisfação do cliente (satisfação acumulada da terceira *release* menos a satisfação acumulada da segunda *release*).

Em determinadas execuções é necessária uma quarta *release*. Isto ocorre porque nas *releases* anteriores o escopo, que correspondia uma maior expectativa aos interessados, era menor que o limite de 96 pontos de função. Com isto, esta diferença vai acumulando para as *releases* posteriores. É possível também que os interessados, após uma análise do escopo, concluam que alguns dos requisitos iniciais não são importantes (geralmente estes requisitos são considerados apenas nas últimas *releases*) e, consequentemente, os excluam do projeto. Com isto, diminui-se o custo e tempo de desenvolvimento do sistema. Além disto, o tempo de execução das buscas de soluções é quase insignificante em relação a uma análise sistemática e intuitiva de todo escopo do software de maneira não automatizada.

Conclui-se então que existe um ganho significativo em relação às expectativas dos interessados nas *releases* iniciais e no tempo para fazer a priorização dos requisitos funcionais do sistema.

5. Conclusões e Trabalhos Futuros

O estudo apresentado na seção 4 mostra que a proposta é eficiente, embora sejam necessários outros estudos que comprovem a validação da proposta. Neste sentido, dois estudos para explorar outras características da solução. O primeiro com a finalidade de comparar a proposta da seção 3 com trabalhos relacionados ao NRP, que utilizam uma abordagem de valor fixo para medir o esforço de desenvolvimento de um requisito. Neste caso, faremos uma avaliação dos efeitos da não-linearidade, característica da APF, no contexto do NRP. O segundo tem objetivo de explorar a característica bi-objetiva do problema (satisfação dos patrocinadores x esforço) a fim de propor boas

soluções aos patrocinadores e, sendo assim, auxiliando-os nas tomadas de decisões quanto a prazo e custo de projetos de software.

Referências

- Ahn, Y.; Suh, J.; Kim, S.; Kim, H. (2003). The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice*, v. 15, n. 2, p. 71–85.
- Bagnall, A. J.; Rayward-Smith, V. J.; Whittle, I. M. (2001). The next release problem. *Information and Software Technology*, v. 43, n. 14, p. 883–890.
- Boehm, B. W.; Abts, C.; Brown, A. W.; et al. (2000). *Software Cost Estimation with Cocomo II*. Prentice Hall.
- COSMIC (2003). *COSMIC-FFP Measurement Manual: Versão 2.2*. Common Software Measurement International Consortium: .
- Del Sagrado, J.; Águila, I. M.; Orellana, F. J. (2011). Requirements interaction in the next release problem. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*.
- Durillo, J. J.; Zhang, Y.; Alba, E.; Harman, M.; Nebro, A. J. (2010). *A study of the bi-objective next release problem*. v. 16p. 29–60
- Ferreira, R. C. da C.; Hazan, C. (2010). Uma Aplicação da Análise de Pontos de Função no Planejamento e Auditoria de Custos de Projetos de Desenvolvimento de Sistemas. *Workshop de Computação Aplicada em Governo Eletrônico*, n. 2010.
- IFPUG (2009). *Function Point Counting Practices Manual (Release 4.3)*.
- Li, C.; Akker, M.; Brinkkemper, S.; Diepen, G. (2010). An integrated approach for requirement selection and scheduling in software release planning. *Requirements Engineering*, v. 15, n. 4, p. 375–396.
- Matson, J. E.; Barrett, B. E.; Mellichamp, J. M. (1994). Software development cost estimation using function points. *IEEE Transactions on Software Engineering*, v. 20, n. 4, p. 275 –287.
- SLTI, S. de L. e T. da I. (2011). Guia Prático para Contratação de Soluções de Tecnologia da Informação.
- Souza, J. T. De; Maia, C. L. B.; Ferreira, T. do N.; Carmo, R. A. F. Do; Brasil, M. M. A. (2011). An Ant Colony Optimization Approach to the Software Release Planning with Dependent Requirements. In: Cohen, M. B.; Cinnéide, M. Ó.[Eds.]. *Search Based Software Engineering*. Springer Berlin Heidelberg. p. 142–157.
- Tonella, P.; Susi, A.; Palma, F. (2013). Interactive requirements prioritization using a genetic algorithm. *Information and Software Technology*, v. 55, n. 1, p. 173–187.
- Zhang, Y.; Harman, M.; Lim, S. L. (2013). Empirical evaluation of search based requirements interaction management. *Information and Software Technology*, v. 55, n. 1, p. 126–152.

Uma Adaptação de Algoritmo Genético para o Problema do Próximo Release com Interdependências entre Requisitos

Italo Yeltsin¹, Matheus Paixão¹, Jerffeson Souza¹

¹Grupo de Otimização em Engenharia de Software da UECE

Universidade Estadual do Ceará (UECE)

Avenida Paranjana, 1700 – Fortaleza – CE – Brasil

{br.yeltsin, mhepaixao}@gmail.com, jeff@larces.uece.br

Abstract. *The Next Release Problem consists in select a set of requirements to be developed in the system's next release, aiming to maximize the overall importance value. When considering the requirements' interdependencies, more constraints are added and the requirements' characteristics become dynamic, so that the algorithms have to be better suited for this new problem. This paper presents a genetic algorithm proposal to handle the requirements' relations in this problem, focusing in the initial population generation method and the repairing procedure. Empirical evaluations state that the proposed algorithm presents better solutions when compared with the usually used genetic algorithms.*

Resumo. *O Problema do Próximo Release consiste em selecionar um conjunto de requisitos para serem entregues na próxima release do sistema, maximizando a importância. Ao se considerar as interdependências entre requisitos, mais restrições são adicionadas e as características dos requisitos tornam-se dinâmicas, de forma que os algoritmos têm de ser melhor adaptados para esse novo problema. Este trabalho apresenta uma adaptação de algoritmo genético para tratar as relações entre requisitos na resolução deste problema, tendo como foco a geração da população inicial e a reparação de indivíduos inválidos. Avaliações empíricas mostram que o algoritmo proposto apresenta soluções melhores em comparação com os algoritmos genéticos utilizados normalmente.*

1. Introdução

No modelo de desenvolvimento de software iterativo e incremental, versões funcionais do sistema são entregues ao(s) cliente(s) ao longo de ciclos, ou iterações. Tais etapas também são chamadas de *releases*. Uma *release* é desenvolvida durante um período de tempo definido e tem um certo orçamento disponível, de forma que é normalmente impossível selecionar todos os requisitos do sistema para serem desenvolvidos em uma única *release*. É de conhecimento geral que quanto maior a presença do cliente no processo de desenvolvimento, menor é a probabilidade de falhas ao longo do projeto. Assim, é necessário que a seleção dos requisitos para serem incluídos na próxima *release* do sistema seja feita de forma a manter o interesse do cliente no projeto.

Na área de Engenharia de Software Baseada em Busca, esse problema é chamado de Problema do Próximo Release, ou *The Next Release Problem* (NRP). Foi primeiramente modelado em [Bagnall et al. 2001]. Neste trabalho cada cliente solicita um

subconjunto de requisitos para serem incluídos na próxima *release*, caso todos os requisitos de um certo cliente sejam selecionados, este cliente é considerado satisfeito. O objetivo é exatamente selecionar um subconjunto de requisitos para serem implementados na próxima release, maximizando a satisfação dos clientes e respeitando o orçamento disponível. O problema foi solucionado utilizando tanto técnicas exatas como metaheurísticas. Por conta do problema ser NP-completo, algoritmos exatos mostraram-se inviáveis na resolução de instâncias maiores, sendo mais apropriado a utilização de metaheurísticas.

Outra abordagem, proposta inicialmente em [Baker et al. 2006], considera que os próprios requisitos apresentam um certo valor de importância e um custo de implementação. Neste, a importância da *release* é dada a partir dos valores de importância dos requisitos incluídos na mesma. O objetivo é selecionar um subconjunto de requisitos que maximize essa importância, respeitando ainda o orçamento. Neste trabalho foram utilizadas somente metaheurísticas na resolução.

Um dos aspectos mais importantes para a atividade de seleção de requisitos é a análise das relações, ou interdependências, entre os requisitos, como pode ser visto em [Carlshamre et al. 2001] e [Dahlstedt and Persson 2005]. As interdependências entre requisitos podem ser classificadas em dois grupos distintos: *interdependências funcionais* e *interdependências de valor* [Del Sagrado et al. 2011]. As primeiras determinam relações onde a implementação de certo requisito depende diretamente da implementação de outro. Nas últimas, a implementação de um requisito pode influenciar nas características de outro requisito, como aumentar a importância ou diminuir seu custo. Apesar de importantes, as interdependências entre requisitos têm sido pouco exploradas na área de SBSE, fazendo com que os modelos matemáticos fiquem muito distantes de um ambiente real de planejamento de *releases*. Acredita-se que este seja um dos principais motivos para a não adoção de abordagens baseadas em busca para o NRP em projetos reais.

O trabalho em [Del Sagrado et al. 2011] mostra as diferenças no processo de busca da solução quando as interdependências são consideradas. Basicamente, as interdependências funcionais adicionam mais restrições ao problema de seleção, diminuindo e dificultando a exploração do espaço de busca. Com relação às interdependências de valor, as mesmas tornam dinâmicos os valores de importância e custo dos requisitos, dificultando seu cálculo. Dessa forma, faz necessário uma melhor adaptação das metaheurísticas para solução deste problema.

Mais especificamente para os algoritmos genéticos [John 1975], as interdependências funcionais dificultam a geração da população inicial de forma aleatória. Como o espaço de busca torna-se mais limitado, a dificuldade em gerar soluções válidas de forma aleatória aumenta, e dependendo da quantidade de interdependências funcionais, torna-se praticamente impossível. As interdependências também aumentam a probabilidade das operações de crossover e mutação gerarem indivíduos inválidos. Neste caso, faz-se necessário um operador de reparação para tornar válido certo indivíduo que esteja quebrando alguma restrição. As estratégias mais comuns para reparação são totalmente aleatórias o que piora muito a qualidade das soluções reparadas.

Este artigo apresenta uma nova proposta de algoritmo genético para resolução do Problema do Próximo Release considerando as interdependências entre requisitos,

com foco na criação da população inicial e no operador de reparação de indivíduos inválidos. Avaliações empíricas realizadas são consistentes em mostrar que o novo algoritmo genético apresenta resultados melhores que as atuais abordagens para resolução do NRP.

Além de [Del Sagrado et al. 2011], outro trabalho relacionado pode ser visto em [Nascimento and Souza 2012], onde é proposta uma adaptação do algoritmo de otimização por colônia de formigas (ACO) para a resolução do NRP com interdependências. A adaptação do ACO é comparada à têmpera simulada e ao algoritmo genético, sendo que nenhuma dessas metaheurísticas foi adaptada para tratar as interdependências.

O restante deste artigo é organizado da seguinte forma: Na Seção 2 são mostradas com mais detalhes as possíveis interdependências entre requisitos e na Seção 3 o NRP com interdependências é matematicamente formulado. Na Seção 4 é apresentado o novo algoritmo proposto, enquanto a Seção 5 apresenta as avaliações empíricas realizadas. Finalmente, a Seção 6 apresenta as conclusões e trabalhos futuros.

2. Interdependências entre Requisitos

Os requisitos relacionam-se entre si de várias formas diferentes e complexas, sendo esses relacionamentos chamados de interdependências. Em um ambiente de planejamento da próxima *release* essas interdependências ganham mais importância, pois o fato dos requisitos se relacionarem torna difícil, e eventualmente impossível, selecionar um conjunto de requisitos baseado somente em prioridades do cliente [Carlshamre et al. 2001].

Como citado anteriormente, as interdependências podem ser *funcionais* ou *de valor*, sendo que as interdependências que influenciam no planejamento da próxima *release* são mostradas com mais detalhes a seguir [Carlshamre et al. 2001]:

- **Funcionais**

- R_1 **AND** R_2 , quando o requisito R_1 depende do requisito R_2 para funcionar e vice-versa. Em outras palavras, os dois requisitos devem ser implementados na mesma *release*.
- R_1 **REQUIRES** R_2 , neste caso o requisito R_1 depende do requisito R_2 , mas não vice-versa. O requisito R_1 só pode ser incluído na *release* se R_2 também o for, mas R_2 pode ser selecionado sozinho.

- **Valor**

- R_1 **CVALUE** R_2 , a implementação de R_1 influencia no valor de importância do requisito R_2 .
- R_1 **ICOST** R_2 , a implementação do requisito R_1 influencia no custo de desenvolvimento de R_2 .

Com relação às interdependências funcionais é importante distinguir a diferença entre dependências e requisitos dependentes. As dependências de um certo requisito r_i consiste no conjunto de todos os requisitos que r_i necessita para ser incluído na *release*, incluindo as dependências das dependências, de forma recursiva. Requisitos dependentes de r_i são todos aqueles que precisam que r_i esteja na *release* para que ele mesmo possa ser selecionado para a *release*.

Vale ressaltar ainda que as influências de valor CVALUE e ICOST podem ser tanto positivas quanto negativas.

3. Formulação Matemática do Problema

O conjunto de requisitos é representado por $R = \{r_1, r_2, \dots, r_N\}$, onde N é a quantidade de requisitos. Os valores w_i e c_i indicam, respectivamente, a importância e custo de implementação do requisito r_i . O orçamento disponível para a *release* é dado por b .

O Problema do Próximo Release é então formulado a seguir:

$$\text{maximizar: } \sum_{i=1}^N w_i x_i \quad (1)$$

$$\text{sujeito a: } \sum_{i=1}^N c_i x_i \leq b \quad (2)$$

onde, a variável de decisão $X = \{x_1, x_2, \dots, x_N\}$ indica quais requisitos foram selecionados para a próxima *release*. O requisito r_i é incluído na *release* quando $x_i = 1$, caso contrário $x_i = 0$.

As interdependências funcionais são representadas a partir de uma única matriz com dimensão $N \times N$, chamada de *functional*. Se $functional_{ij} = 1$ e $functional_{ji} = 0$, têm-se uma interdependência **REQUIRES** entre os requisitos r_i e r_j . Assim, r_i só pode ser incluído na *release* caso r_j também o seja, mas r_j pode ser selecionado sem a presença de r_i . No caso em que $functional_{ij} = functional_{ji} = 1$, têm-se uma relação **AND** entre os requisitos r_i e r_j , ou seja, os dois requisitos têm que ser incluídos juntos na *release*.

As interdependências de valor **CVALUE** e **ICOST** são representadas com matrizes de dimensão $N \times N$, *cvalue* e *icost* respectivamente. São compostas por valores contidos no intervalo $[-1, 1]$ que indicam a porcentagem de influência na importância e custo entre os requisitos. Por exemplo, quando $cvalue_{ij} = 0.2$, a inclusão do requisito r_i na *release* resulta em um aumento da importância do requisito r_j em 20%. Da mesma forma, $icost_{ij} = -0.4$ indica que, caso o requisito r_i seja selecionado para a *release*, o custo de desenvolvimento de r_j diminui em 40%.

4. Algoritmo Genético Proposto

Um pseudo-código simplificado de algoritmo genético pode ser visto no Algoritmo 1. Ressalta-se a necessidade da função de reparação caso as operações de crossover ou mutação gerem um indivíduo inválido.

O algoritmo genético canônico, abreviado neste trabalho como AG_{can} , é aquele onde tanto a geração da população inicial como a função de reparo são realizadas de forma aleatória. Nesta estratégia de geração da população inicial, indivíduos aleatórios são criados, caso este seja inválido, é reparado. Esta reparação na população inicial é necessária, pois como dito anteriormente, dependendo da quantidade de interdependências, uma geração totalmente aleatória pode nunca encontrar uma população inicial válida. A função de reparo simplesmente escolhe um requisito aleatório e o remove do indivíduo, repetindo esse procedimento até que o indivíduo não quebre mais nenhuma restrição.

No Algoritmo 2 pode ser vista a proposta de geração da população inicial considerando as interdependências. O algoritmo consiste em selecionar requisitos de forma

Algoritmo 1 Pseudo-código de um Algoritmo Genético

```
População ← CriarPopulaçãoInicial()
while Critério de Parada do
    CalcularFitness();
    for each indivíduo na População do
        pais ← SelecionarPais()
        indivíduo ← RealizarCrossover(pais)
        RealizarMutação(indivíduo)
        if indivíduo é inválido then
            Reparar(indivíduo)
        end if
    end for
end while
```

aleatória e adicioná-lo ao indivíduo junto com todas as suas dependências funcionais, inclusive as recursivas. Este procedimento garante que o indivíduo não vai quebrar nenhuma restrição funcional. Essa adição de requisitos aleatórios juntamente com suas dependências se repete até que o orçamento seja ultrapassado e o indivíduo torne-se inválido, então basta retirar todos os requisitos que acabaram de ser adicionados para tornar o indivíduo válido novamente. No momento da retirada das dependências, é importante que somente as dependências que foram adicionadas na atual iteração sejam retiradas. Uma retirada de todas as dependências de certo requisito pode resultar em uma quebra de restrição funcional de algum outro requisito na *release*. Ao final ainda é aplicado um operador de busca local para aumentar ainda mais a qualidade do indivíduo.

Algoritmo 2 CriarPopulaçãoInicial()

```
for each indivíduo na população do
    parada ← false
    while parada = false do
        requisitoAleatório ← SelecionaRequisitoAleatorio()
        AdicioneRequisito(requisitoAleatório)
        AdicioneDependencias(requisitoAleatório)
        if Custo(indivíduo) > orçamento then
            RemovaRequisito(requisitoAleatório)
            RemovaDependenciasAdicionadaNestaIteracao(requisitoAleatório)
            parada ← true
        end if
    end while
    BuscaLocal(indivíduo);
end for
```

Uma função de reparação que leva em consideração as interdependências pode ser vista no Algoritmo 3. O princípio do algoritmo é solucionar primeiramente a restrição de orçamento e depois as restrições funcionais, caso as duas sejam necessárias. Primeiramente é verificado se o indivíduo está acima do orçamento. Caso esteja, um requisito contido na release é selecionado aleatoriamente e é removido juntamente com todos os seus dependentes. Remover os dependentes neste contexto é necessário, pois retirar so-

mente o requisito aleatório pode fazer com que alguma restrição funcional seja quebrada com relação a algum outro requisito na *release*. Este procedimento é repetido até que o custo do indivíduo torne-se menor que o orçamento, sendo o momento de solucionar as restrições funcionais. Um novo requisito presente na *release* é selecionado aleatoriamente para ter suas dependências adicionadas, dessa forma é possível solucionar essa restrição aumentando a importância total da *release*. Caso essa adição de dependências quebre a restrição de orçamento, a única solução é remover o requisito aleatório bem como todos os seus dependentes e suas dependências que acabaram de ser adicionadas. É importante frisar que somente os requisitos adicionados na iteração atual devem ser removidos, como explicado no Algoritmo anterior.

Algoritmo 3 Reparar(indivíduo)

```

while indivíduo inválido do
    if Custo(indivíduo) > orçamento then
        requisitoAleatório ← SelecionaRequisitoAleatorioNoIndivíduo()
        RemovaRequisito(requisitoAleatório);
        RemovaDependentes(requisitoAleatório);
    else
        requisitoAleatório ← SelecionaRequisitoAleatorioNoIndivíduo()
        AdicioneDependencias(requisitoAleatório);
        if Custo(Indiviuo) > orçamento then
            RemovaRequisito(requisitoAleatório);
            RemovaDependentes(requisitoAleatório);
            RemovaDependenciasAdicionadasNestaIteracao(requisitoAleatório);
        end if
    end if
end while

```

5. Avaliação Empírica

5.1. Configuração

O conjunto de instâncias utilizado foi o mesmo de [Nascimento and Souza 2012]. São 5 instâncias geradas aleatoriamente, variando a quantidade de requisitos de 10 a 500. O nome da instância está no formato *I_R*, onde *R* é a quantidade de requisitos. A instância *I_100*, por exemplo, tem 100 requisitos. O valor de importância do requisito pode variar entre 1 e 10 enquanto o custo de implementação está no intervalo entre 1 e 5. Para garantir que não é possível adicionar todos os requisitos na *release*, o orçamento foi configurado como 60% da soma dos custos de todos os requisitos, como em [Nascimento and Souza 2012].

Para efeito de comparação, cada instância foi solucionada por 4 algoritmos genéticos diferentes. São eles o AG canônico (AG_{can}) que utiliza as estratégias de geração da população inicial e reparação aleatórios, um AG utilizando a abordagem de população inicial como mostrada no Algoritmo 2 e reparação aleatória (AG_{pi}), um AG com população inicial aleatória e a função de reparação mostrada no Algoritmo 3 (AG_{re}) e finalmente o AG utilizando as abordagens de criação da população inicial e reparação propostas neste artigo (AG_{prop}). A utilização de diferentes AGs tem como objetivo mostrar o impacto da proposta de população inicial e da proposta de reparo separadamente,

bem como indicar a melhora nos resultados quando as duas propostas são usadas em conjunto.

Todos os algoritmos foram executados com os seguintes parâmetros: taxa de crossover igual 0.9 e taxa de mutação igual a $1/(10N)$, onde N é a quantidade de requisitos da instância. Número de indivíduos na população igual a 100 e número de gerações igual a 100. A seleção dos pais é feita pelo método da roleta. Foi ainda empregada a estratégia de elitismo, onde ao final de cada iteração, 20% dos melhores indivíduos da população são automaticamente adicionados na próxima geração.

Para o algoritmo de geração da população inicial proposto, o operador de busca local utilizado foi um simples *Hill Climbing*. É considerada uma solução vizinha toda aquela que for válida e diferir em somente um requisito. Define-se então aleatoriamente um número máximo de buscas locais realizadas, variando de 1 a 10% do número de requisitos. Essa aleatoriedade na quantidade de buscas locais garante que a população inicial ainda terá uma boa diversidade.

Para cada instância, cada algoritmo foi executado um total de 30 vezes, obtendo-se média e desvio padrão tanto do valor de fitness como do tempo de execução em milissegundos.

5.2. Resultados e Análises

Na Tabela 1 são apresentados os resultados do AG canônico e do AG com a população inicial proposta e reparo aleatório. São destacados os melhores resultados tanto de fitness como de tempo.

Tabela 1. Resultados dos algoritmos AG_{can} e AG_{pi}

Instância	Métrica	Algoritmo	
		AG_{can}	AG_{pi}
I_10	Fitness	174.62 ± 7.25	219.10 ± 0.34
	Tempo	6.33 ± 4.81	6.33 ± 4.81
I_25	Fitness	875.01 ± 107.56	1054.08 ± 22.07
	Tempo	26.33 ± 4.81	35 ± 5
I_50	Fitness	3419.74 ± 410.32	7515.05 ± 126.93
	Tempo	57 ± 7.81	195.33 ± 26.29
I_100	Fitness	6014.93 ± 758.7	20795.8 ± 331.24
	Tempo	146.66 ± 13.24	1192.33 ± 21.08
I_500	Fitness	14035.40 ± 2123.96	218250 ± 1187.97
	Tempo	919 ± 76.34	32504.6 ± 1297.83

Percebe-se que a qualidade dos indivíduos na população inicial influencia consideravelmente o valor final de fitness. O algoritmo genético utilizando a população inicial proposta chega a ter em média, um valor de fitness 373% maior. Pelo fato de gerar indivíduos de melhor qualidade na população inicial, o AG_{pi} torna-se mais constante ao longo das execuções, apresentando em média, um desvio padrão 69% menor que o AG canônico. Devido principalmente ao operador de busca local, o AG_{pi} apresenta um tempo de execução maior em comparação com o AG_{can} . Mesmo assim, ainda pode ser considerado um tempo de execução baixo, com a maior instância demorando pouco mais de 30 segundos, em média.

Na Tabela 2 são mostrados os valores de fitness e tempo de execução para o AG canônico e o AG com população inicial aleatória e função de reparo proposta. Os melhores resultados são novamente destacados.

Tabela 2. Resultados dos algoritmos AG_{can} e AG_{re}

Instância	Métrica	Algoritmo	
		AG_{can}	AG_{re}
I_10	Fitness	174.62 ± 7.25	177.87 ± 7.24
	Tempo	6.33 ± 4.81	7.33 ± 4.42
I_25	Fitness	875.01 ± 107.56	881.45 ± 94.68
	Tempo	26.33 ± 4.81	28 ± 5.41
I_50	Fitness	3419.74 ± 410.32	3812.68 ± 852.52
	Tempo	57 ± 7.81	62.33 ± 15.63
I_100	Fitness	6014.93 ± 758.7	6338.76 ± 854.59
	Tempo	146.66 ± 13.24	154 ± 18.36
I_500	Fitness	14035.40 ± 2123.96	14037.3 ± 1732.07
	Tempo	919 ± 76.34	921.66 ± 67.13

Ao comparar os valores de fitness encontrados pelos dois algoritmos, constata-se uma pequena melhora com a utilização da função de reparo proposta. Em média os resultados de AG_{re} para fitness são 3.8% melhores. Percebe-se então que o operador de reparação proposto não auxilia na melhora das soluções, mas somente repara os indivíduos inválidos com mais qualidade. Com relação ao desvio padrão e ao tempo de execução, a diferença dos resultados é relativamente pequena, com 17% e 7% respectivamente.

São mostrados na Tabela 3 os resultados dos algoritmos AG_{pi} e AG_{re} , juntamente com os resultados alcançados pela adaptação de AG proposta neste artigo, onde os procedimentos para geração da população inicial e função de reparo levam em consideração as interdependências entre requisitos (AG_{prop}).

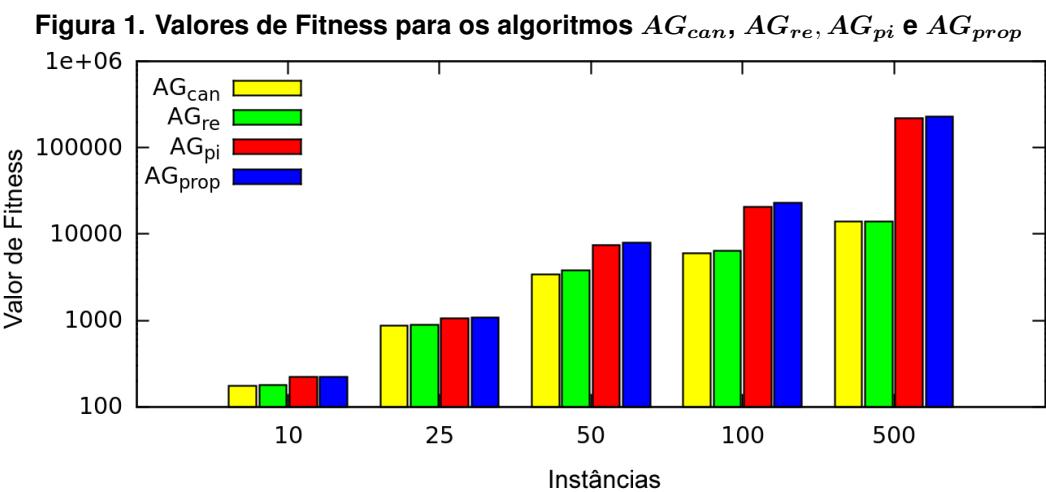
A adaptação de algoritmo genético proposto atingiu os melhores valores de fitness para todas as instâncias. Esses valores chegam a ser, em média 4.5% e 392% melhores

Tabela 3. Resultados dos algoritmos AG_{pi} , AG_{re} e AG_{prop}

Instância	Métrica	Algoritmo		
		AG_{pi}	AG_{re}	AG_{prop}
I_10	Fitness	219.10 ± 0.34	177.87 ± 7.24	219.10 ± 0.34
	Tempo	6.33 ± 4.81	7.33 ± 4.42	5.33 ± 5.61
I_25	Fitness	1054.08 ± 22.07	881.45 ± 94.68	1069.87 ± 13.26
	Tempo	35 ± 5	28 ± 5.41	39 ± 3
I_50	Fitness	7515.05 ± 126.93	3812.68 ± 852.52	7910.63 ± 72.98
	Tempo	195.33 ± 26.29	62.33 ± 15.63	187.66 ± 15.20
I_100	Fitness	20795.8 ± 331.24	6338.76 ± 854.59	22804.3 ± 231.42
	Tempo	1192.33 ± 21.08	154 ± 18.36	1450 ± 156.95
I_500	Fitness	218250 ± 1187.97	14037.3 ± 1732.07	231817 ± 1698.34
	Tempo	32504.6 ± 1297.83	921.66 ± 67.13	333530 ± 38145.1

que os valores de AG_{pi} e AG_{re} , respectivamente. Para instâncias médias e grandes, o ganho em qualidade da solução ao utilizar o algoritmo genético proposto é ainda mais pronunciado. Considerando somente as instâncias a partir de 50 requisitos, o valor de fitness quando comparado com o AG_{pi} é 7% maior, em média. Com relação ao desvio padrão, o algoritmo proposto apresenta valores ainda menores que os de AG_{pi} , sendo 14% menor, em média.

Na Figura 1 podem ser vistos os valores de fitness de todos os algoritmos para todas as instâncias. Percebe-se claramente o melhor desempenho do algoritmo proposto neste trabalho. Vale ressaltar também a semelhança na proporção dos resultados em todas as instâncias, sugerindo uma constância nos resultados, independente da quantidade de requisitos.



A partir dos resultados apresentados pode-se concluir que o algoritmo genético proposto neste artigo consegue atingir uma melhor qualidade na solução e um menor desvio padrão ao realizar tanto a criação da população inicial como a função de reparação considerando as interdependências entre os requisitos. Ao considerar boa parte das possíveis interdependências entre requisitos e ainda apresentar bons resultados, considera-se que a abordagem de planejamento da próxima *release* apresentada neste trabalho está pronta para ser experimentada em projetos reais de software.

6. Conclusão e Trabalhos Futuros

A seleção de quais requisitos serão incluídos na próxima *release* é uma das tarefas mais importantes do modelo de desenvolvimento iterativo e incremental de software. As relações, ou interdependências, entre os requisitos exercem um papel importante neste planejamento, muitas vezes fazendo com que não seja possível a seleção dos requisitos baseado somente em prioridades do(s) cliente(s).

Na área de Engenharia de Software Baseada em Busca, são necessárias abordagens que considerem as possíveis interdependências entre requisitos, para tornar o modelo matemático cada vez mais próximo da realidade. Quando as interdependências são levadas em consideração na resolução do NRP, o espaço de busca sofre algumas modificações, sendo necessária uma adaptação mais especializada dos algoritmos. Neste contexto, foi

proposto uma adaptação de algoritmo genético para tratar melhor as interdependências entre requisitos, com foco principal na geração da população inicial e na função de reparação de indivíduos inválidos.

Avaliações empíricas foram realizados com diferentes algoritmos genéticos para medir separadamente o impacto na qualidade da solução tanto da proposta de população inicial quanto da proposta de reparação. Percebeu-se que a qualidade dos indivíduos na população inicial tem uma grande influência na solução final, de forma que os resultados do algoritmo utilizando a proposta de população inicial e reparação aleatória foram 373% maior que do AG canônico. Utilizando uma população inicial aleatória e a função de reparação proposta, a melhora em fitness não foi tão grande, indicando que o reparo não é responsável pela exploração do espaço de busca mas por reparar os indivíduos inválidos com mais qualidade. Isso é comprovado ao se utilizar tanto a população inicial como a reparação proposta. Comparado com o algoritmo que utiliza somente a população inicial, a melhora em fitness é de 4.5%, em média.

Como trabalhos futuros pretende-se aplicar o algoritmo proposto a instâncias com um número maior de interdependências bem como também aplicá-lo a instâncias e situações reais de planejamento da próxima *release*. No contexto da proposta em si, a mesma ainda pode ser melhorada ao se desenvolver operadores de crossover e mutação específicos para o tratamento das interdependências entre requisitos.

Referências

- Bagnall, A., Rayward-Smith, V., and Whittle, I. (2001). The next release problem. *Information and Software Technology*, 43(14):883–890.
- Baker, P., Harman, M., Steinhofel, K., and Skaliotis, A. (2006). Search based approaches to component selection and prioritization for the next release problem. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 176–185. IEEE.
- Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., and Natt och Dag, J. (2001). An industrial survey of requirements interdependencies in software product release planning. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 84–91. IEEE.
- Dahlstedt, Å. G. and Persson, A. (2005). Requirements interdependencies: state of the art and future challenges. In *Engineering and managing software requirements*, pages 95–116. Springer.
- Del Sagrado, J., Aguila, I., and Orellana, F. (2011). Requirements interaction in the next release problem. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 241–242. ACM.
- John, H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. USA: University of Michigan.
- Nascimento, T. and Souza, J. (2012). Uma abordagem aco para o problema do próximo release com interdependência de requisitos.

Uma Abordagem MultiObjetivo para o Problema de Desenvolvimento de Cronogramas de Projetos de Software

Sophia Nóbrega¹, Sérgio R. de Souza¹, Marcone J. F. Souza²

¹Centro Federal de Educação Tecnológica de Minas Gerais (CEFET/MG)
Av. Amazonas, 7675 – 30.510-000 – Belo Horizonte – MG – Brasil

²Departamento de Computação
Universidade Federal de Ouro Preto (UFOP)
Campus Universitário – 35.400-000 – Ouro Preto – MG – Brasil

sophia@dcc.ufmg.br, sergio@dppg.cefetmg.br, marcone@iceb.ufop.br

Abstract. This paper deals with the Problem of Development Schedules (PDS) for Software Projects. In this study several important aspects are considered, such as availability of resources and skills, project tasks and their interdependencies, deadlines, costs, tasks size estimate, teams productivity and experience. An algorithm based on the multiobjective metaheuristic MOVNS are proposed for solving this problem. The results obtained are competitive and better than those presented by expert project managers.

Resumo. Esse artigo trata o Problema de Desenvolvimento de Cronograma (PDC) para projetos de Software. Nesse estudo importantes aspectos são considerados, como disponibilidade dos recursos e suas habilidades, as tarefas do projeto e suas interdependências, deadlines, custos, estimativa de tamanho das tarefas, produtividade das equipes e sua experiência. Para resolver o problema, é proposto o uso de um algoritmo multiobjetivo baseado na metaheurística MOVNS. Os resultados obtidos são competitivos e melhores que aqueles apresentados por experientes gerentes de projetos.

1. Introdução

Empresas preocupadas em se manterem competitivas buscam sempre reduzir o custo e a duração de seus projetos, mas esses dois objetivos são sempre conflitantes. Existe a necessidade de controlar pessoas e o processo de desenvolvimento, conseguindo alocar eficientemente os recursos disponíveis para executar as tarefas demandadas pelo projeto, sempre satisfazendo uma variedade de restrições.

A abordagem de problemas complexos da Engenharia de Software, como é o caso do problema abordado nesse artigo, utilizando técnicas de otimização, é uma emergente área de pesquisa denominada SBSE (*Search Based Software Engineering*) [Barros and Dias-Neto 2011]. O principal objetivo do SBSE é oferecer mecanismos de apoio ao engenheiro de software para resolver problemas inerentes da Engenharia de Software. Baseado nesses conceitos, a abordagem proposta tem, como objetivo, apoiar e guiar o gerente na atividade de desenvolvimento de cronogramas de projetos de software.

Dentro desse contexto, o presente trabalho apresenta importantes contribuições. É apresentada a primeira abordagem baseada em otimização multiobjetivo usando a

metaheurística MOVNS (*MultiObjective Variable Neighborhood Search*) para resolver o PDC. A segunda contribuição é a proposição de um conjunto de 9 estruturas de vizinhança, com o objetivo de explorar todo o espaço de busca para o problema. Todos os movimentos são descritos na subseção 4.2.

No desenvolvimento desse trabalho, será apresentada uma caracterização detalhada de todos os aspectos e variáveis presentes no dia a dia de um gerente de projeto de software, como habilidades e experiências individuais, custos, prazos, produtividade, horas extras, interdependência das tarefas, duração de tarefas, etc. Para avaliar e validar esta pesquisa, serão realizados estudos empíricos.

2. Trabalhos Relacionados

Em [Antoniol et al. 2005] e [Penta et al. 2011] os autores resolveram, respectivamente, o problema de alocação de pessoas e o problema de cronograma de projetos com alocação de pessoas. Nestes trabalhos, utilizam formulações mono-objetivo e implementam os algoritmos Genéticos, *Simulated Annealing* e *Hill Climbing*. Para validar as implementações, realizaram estudos empíricos com dados de projetos reais, obtendo soluções valiosas para auxiliarem os gerentes de projetos no processo de tomada de decisão. No estudo apresentado por [Penta et al. 2011] também foi modelada uma abordagem multiobjetivo, mas não foi avaliada.

Em [Alba and Chicano 2007], o Problema de Desenvolvimento de Cronograma de projetos é tratado, utilizando-se um Algoritmo Genético clássico, que foi validado utilizando dados fictícios, obtidos a partir de um gerador automático de projetos. Os autores tratam dois objetivos: minimizar o tempo e minimizar o custo do projeto, que são combinados em uma única função objetivo, usando pesos diferentes. Em [Minku et al. 2012], os autores propõem um Algoritmo Evolucionário (AE) multiobjetivo e compararam seu algoritmo com o proposto em [Alba and Chicano 2007].

Em [Gueorguiev et al. 2009], os autores apresentaram a primeira formulação multiobjetivo para esse tipo de problema, no qual robustez e tempo de conclusão do projeto são tratados como dois objetivos concorrentes para resolver o problema de planejamento de projeto de software. O algoritmo SPEA II foi implementado e testado em quatro projetos reais. Os resultados indicam um bom desempenho da abordagem proposta. Em [Colares 2010], o autor utilizou um algoritmo genético multiobjetivo para resolver o problema de alocação de equipes e desenvolvimento de cronogramas. A função de aptidão proposta busca minimizar o tempo total do projeto, o custo total, o atraso nas tarefas e as horas extras.

Segundo o conhecimento dos autores do presente artigo, não foram encontradas referências na literatura ao uso dos algoritmos GRASP e MOVNS para resolver o PDC e, neste sentido, a proposição do uso destas abordagens para este problema como uma contribuição científica de interesse do presente artigo. Observa-se, também, que, na grande maioria dos estudos encontrados na literatura, o PDC é tratado por meio de algoritmos baseados em busca populacional, seja por uma abordagem mono-objetivo ou por uma abordagem multiobjetivo. No presente trabalho, optou-se, por desenvolver algoritmos baseados em busca local para a solução do PDC.

3. Formulação do Problema

A formulação matemática apresentada foi desenvolvida baseada na proposta feita por Colares [Colares 2010]. Algumas características do problema modeladas por Colares foram excluídas, pois não foi possível coletar dados de projetos reais que permitissem o seu uso. O algoritmo proposto recebe, como parâmetros de entrada, tarefas e recursos humanos. As tarefas possuem, como atributos, esforço estimado, nível de importância e datas de inicio e fim. Os recursos humanos são divididos em contratado e empregado. O primeiro possui, como atributos, valor hora e dedicação diária. O empregado possui os atributos salário, dedicação diária, valor hora extra e tempo máximo de hora extra. Ambos os tipos possuem um atributo que representa seu calendário de disponibilidade.

Cada recurso humano possui uma lista de habilidades individuais, com seu respectivo nível de proficiência, e uma lista de tarefas, com seu respectivo nível de experiência, assim como cada tarefa possui uma lista de habilidades necessárias para sua execução. Cada recurso é alocado a uma determinada tarefa em valores percentuais, que variam de 0 (zero) ao máximo permitido para o recurso.

Para definir a interdependência entre tarefas, ou sequenciamento, a abordagem proposta utiliza os quatro conceitos de relacionamento utilizados pela maioria das ferramentas de gerência de projeto: Início-Início (II), Início-Final (IF), Final-Início (FI) e Final-Final(FF). Além disso, o algoritmo reajusta o início de tarefas que não pertencem ao caminho crítico do projeto, evitando a quebra de restrição de recursos.

A produtividade de um recurso $prod_{r,t}$ pode ser obtida pela fórmula:

$$prod_{r,t} = x_{r,t} r^{dedicacao} \left(\prod_{s \in (S^r \cap S^t)} r^{proef}(s) \right) r^{exp}(t) \quad (1)$$

em que $x_{r,t}$ representa a proporção de esforço do recurso r para executar a tarefa t ; $r^{dedicacao}$ é a dedicação diária do recurso r em horas; S^r é o conjunto de habilidades que o recurso r possui; S^t é o conjunto de habilidades requeridos pela tarefa t ; $r^{proef}(s)$ é o fator de ajuste devido à proficiência do recurso r na habilidade s ; e $r^{exp}(t)$ é o fator de ajuste devido à experiência do recurso r na tarefa t . O tempo de duração de uma tarefa em dias $t^{duracao}$ pode ser obtido pela fórmula:

$$t^{duracao} = \frac{tesforco}{\sum_{r \in R} prod_{r,t}}, \quad \forall t \in T \quad (2)$$

em que $tesforco$ é o esforço da tarefa em Pontos de Função (PF).

Estimado o tempo de duração de cada tarefa ($t^{duracao}$), é calculada a duração total do cronograma do projeto, ou *makespan*. A minimização do *makespan* é o o primeiro objetivo proposto. A duração total do projeto, ou *makespan*, é representada pela função:

$$S = makespan \quad (3)$$

O segundo objetivo proposto é minimizar o custo total do projeto, representado pela função:

$$C = \sum_{i=1}^R \sum_{j=1}^T (c_{ij} t_j^{duracao}) \quad (4)$$

O custo total é a soma do pagamento dos recursos por sua dedicação no projeto. Esse custo c_{ij} é calculado multiplicando o salário pago por hora para o empregado pelo seu tempo dedicado ao projeto mais as horas extras. O tempo dedicado ao projeto é, então, calculado pela soma da dedicação do recurso multiplicado pela duração de cada tarefa.

4. Modelo Heurístico

4.1. Representação da Solução

Cada solução s do problema é representada por uma matriz bidimensional, denominada Matriz de Alocação X , e um vetor, denominado Vetor Cronograma Y , que armazena o instante de início de cada tarefa.

Uma dimensão da Matriz de Alocação representa os recursos humanos disponíveis $\{r_1, r_2, \dots, r_{|R|}\}$, enquanto a outra dimensão representa as tarefas que devem ser executadas $\{t_1, t_2, \dots, t_{|T|}\}$. Na matriz X , cada variável $x_{r,t}$ recebe um valor inteiro entre 0 (zero) e dedicação diária de cada recurso, acrescida do tempo máximo de hora extra, se for o caso. Esse valor inteiro é interpretado, dividindo-o pela dedicação diária, de forma a se obter porcentagens de 0 a 100%, que representam o esforço dedicado pelo recurso r na execução da tarefa t . Quando o percentual for superior a 100%, indica a realização de hora extra.

O vetor Cronograma possui dimensão T , sendo T o total de tarefas. Os índices do vetor representam as tarefas, e cada posição do vetor é preenchida por um real, que indica o tempo de inicio de execução da tarefa.

4.2. Estruturas de Vizinhança

Para explorar o espaço de soluções foram criados 9 movimentos. Todos os movimentos descritos a seguir são sempre realizados, respeitando-se as restrições de compatibilidade entre recursos e tarefas:

- **Movimento Realocar Recurso entre Tarefas Distintas - $M^{RD}(s)$:** este movimento consiste em selecionar duas células x_{ri} e x_{rk} da matriz X e repassar a dedicação de x_{ri} para x_{rk} . Assim, um recurso r deixa de trabalhar na tarefa i e passa a trabalhar tarefa k .
- **Movimento Realocar Recurso de uma Tarefa - $M^{RT}(s)$:** este movimento consiste em selecionar duas células x_{it} e x_{kt} da matriz X e repassar a dedicação de x_{it} para x_{kt} . Assim, a dedicação de um recurso i é realocada para um recurso k que esteja trabalhando na tarefa t .
- **Movimento Desalocar Recurso de uma Tarefa - $M^{DT}(s)$:** consiste em selecionar uma célula x_{rt} da matriz X e zerar seu conteúdo, isto é, retirar a alocação de um recurso r que estava trabalhando na tarefa t .
- **Movimento Desalocar Recurso no Projeto - $M^{DP}(s)$:** consiste em desalocar toda a dedicação de um recurso r no projeto. O movimento retira todas as alocações do recurso r , que deixa de trabalhar no projeto. O recurso volta a trabalhar no projeto assim que uma nova tarefa for associada a ele.
- **Movimento Dedicação de Recursos - $M^{DR}(s)$:** este movimento consiste em aumentar ou diminuir a dedicação de um determinado recurso r na execução de uma tarefa t . Neste movimento, uma célula x_{rt} da matriz X tem seu valor acrescido ou decrescido em uma unidade.

- **Movimento Troca de Recursos entre Tarefa - $M^{TB}(s)$:** duas células x_{ri} e x_{rk} da matriz X são selecionadas e seus valores são permutados, isto é, os recursos que trabalham nas tarefas i e k são trocados.
- **Movimento Troca de Recursos de uma Tarefa - $M^{TO}(s)$:** duas células x_{it} e x_{kt} da matriz X são selecionadas e seus valores são permutados, isto é, os recursos i e k que trabalham na tarefa t são trocados.
- **Movimento Insere Tarefa - $M^{IT}(s)$:** este movimento consiste em inserir uma tarefa que está em uma posição i em outra posição j do vetor Y . Esse movimento é realizado somente entre tarefas sem relações de precedência.
- **Movimento Troca Tarefa - $M^{TT}(s)$:** consiste em trocar duas células distintas y_i e y_k do vetor Y , ou seja, trocar o tempo de inicio de execução das tarefas i e k . Os movimentos de trocas serão feitos sempre respeitando a ordem de precedência para executar as tarefas.

5. Algoritmo Proposto

Nesse artigo, é proposto um algoritmo multiobjetivo, nomeado GRASP-MOVNS, que consiste na combinação dos procedimentos Heurísticos *Greedy Randomized Adaptive Search Procedure* - GRASP [Souza et al. 2010] e *Multiobjective Variable Neighborhood Search* - MOVNS [Geiger 2008]. O algoritmo GRASP-MOVNS foi implementado utilizando a mesma estratégia proposta por [Coelho et al. 2012].

O algoritmo 1 apresenta o pseudocódigo do algoritmo GRASP-MOVNS. Na linha 2, é gerado o conjunto inicial de soluções não dominadas através do procedimento parcialmente guloso *GRASP* [Feo and Resende 1995]. São geradas duas soluções iniciais s_1 e s_2 , que são construídas utilizando, como regra de prioridade, as tarefas de maior duração e as tarefas que pertencem ao caminho crítico, respectivamente. A versão do *GRASP* implementada utiliza, como método de busca local, a heurística *Variable Neighborhood Descent* - VND [Hansen and Mladenovic 1997], que envolve a substituição da solução atual pelo resultado da busca local, quando há uma melhora. Porém, a estrutura de vizinhança é trocada de forma determinística, cada vez que se encontra um mínimo local. A solução resultante é um mínimo local em relação a todas as nove estruturas de vizinhanças: M^{IT} , M^{TT} , M^{DR} , M^{RD} , M^{RT} , M^{DP} , M^{DT} , M^{TB} , M^{TO} .

Nas linhas 6 e 7 do Algoritmo 1 é feita a seleção de um indivíduo do conjunto de soluções não-dominadas D , marcando-o como “visitado”. Quando todos os indivíduos estiverem marcados como visitados, a linha 32 retira estes marcadores. As variáveis *level* e *shaking*, mostradas nas linhas 3 e 4 do Algoritmo 1, regulam a perturbação utilizada no algoritmo. Esta versão do algoritmo MOVNS, proposta por [Coelho et al. 2012], possui um mecanismo que regula o nível de perturbação do algoritmo, ou seja, a variável *shaking* é incrementada quando o algoritmo passa um determinado tempo sem obter boas soluções. Da linha 9 à 12 do Algoritmo 1 ocorre o laço de perturbação do algoritmo. A heurística AdicionarSolucao, mostrada no Algoritmo 2, é acionada na linha 16, e adiciona, ao conjunto solução D , as soluções geradas pelo GRASP-MOVNS. No mecanismo utilizado, quanto maior o valor da variável *shaking*, maior será a intensidade da perturbação na solução. Para cada unidade dessa variável, aplica-se um movimento aleatório, dentre as nove vizinhanças: M^{IT} , M^{TT} , M^{DR} , M^{RD} , M^{RT} , M^{DP} , M^{DT} , M^{TB} , M^{TO} . A variável *level* regula quando a variável *shaking* será incrementada. As linhas 24 e 25 retornam os

Algoritmo 1: GRASP-MOVNS

Entrada: Vizinhança $N_K(s)$; $graspMax$; $levelMax$
Saída: Aproximação de um conjunto eficiente D

```
1  início
2       $D \leftarrow GRASP(graspMax)$ 
3       $level \leftarrow 1$ 
4       $shaking \leftarrow 1$ 
5      enquanto (Critério de parada não satisfeito) faca
6          | Seleciona uma solução não visitada  $s \in D$ 
7          | Marque  $s$  como visitada
8          |  $s \leftarrow s'$ 
9          | para  $i \rightarrow shaking$  faca
10         |   | Seleciona aleatoriamente uma vizinhança  $N_k(.)$ 
11         |   |  $s' \leftarrow Perturbação(s', k)$ 
12         |   fim
13         |    $k_{ult} \leftarrow k$ 
14         |    $incrementa \leftarrow verdadeiro$ 
15         |   para  $s'' \in N_{k_{ult}}(s')$  faca
16         |       | adicionarSolucao( $D, s'', f(s'')$ , added)
17         |       | se  $adicionado = verdadeiro$  então
18         |           |   |  $incrementa \leftarrow falso$ 
19         |       | fim
20         |   fim
21         |   se  $incrementa = verdadeiro$  então
22             |       |    $level \leftarrow level + 1$ 
23         |   senão
24             |       |    $level \leftarrow 1$ 
25             |       |    $shaking \leftarrow 1$ 
26         |   fim
27         |   se  $level \geq levelMax$  então
28             |       |    $level \leftarrow 1$ 
29             |       |    $shaking \leftarrow shaking + 1$ 
30         |   fim
31         |   se todo  $s \in D$  estão marcadas como visitadas então
32             |       |   Marque todos  $s \in D$  como não visitado
33         |   fim
34     fim
35     Retorne  $D$ 
36 fim
```

valores das variáveis $level$ e $shaking$ para uma unidade, quando pelo menos uma solução é adicionada ao conjunto eficiente D .

Algoritmo 2: adicionarSolucao

Entrada: conjunto $D, s', z(s')$
Saída: $conjuntoD, adicionado$

```
1  início
2       $adicionado \leftarrow verdadeiro$ 
3      para  $s \in D$  faca
4          | se  $z(s) \leq z(s')$  então
5              |   |  $adicionado \leftarrow falso$ 
6              |   | break
7          | fim
8          | se  $z(s') < z(s)$  então
9              |   |  $D = D \cup s$ 
10             | fim
11         fim
12         se  $adicionado = verdadeiro$  então
13             |   |  $D = D \cup s'$ 
14         fim
15         Retorne  $D, adicionado$ 
16 fim
```

5.1. Experimentos e Análise

Os algoritmos foram implementados em linguagem java, e os experimentos realizados em um notebook Dell Inspirion 14 Core i3-3110M, 3 MB Cache, 2.4 GHz, 4GB de RAM, sob sistema operacional windows 7. Para realizar os testes, foi utilizado um conjunto de 3 instâncias, disponível em www.decom.ufop.br/prof/marcone/projects/SBSE.html. Os dados das instâncias são de projetos reais de duas empresas de desenvolvimento de sistemas de Belo Horizonte, que estão no mercado de há mais de 8 anos.

Foram criados 3 cenário de teste. O cenário A de teste possui 10 casos de uso, totalizando 72 tarefas, 21 recursos e 7 habilidades envolvidas. O cenário B possui 16 casos de uso totalizando 100 tarefas, 16 recursos e 4 habilidades. O cenário C possui 20 casos de uso com 120 tarefas, 11 recursos e 4 habilidades. Todos os recursos possuem uma carga horária de 8 horas diárias e com salários de acordo com a função desempenhada e experiência. O algoritmo foi configurado durante os testes para não permitir que sejam feitas horas extras. Optou-se por, inicialmente, não permitir que sejam realizadas horas extras, pois esta é uma prática comum na indústria de desenvolvimento de software. Em geral, durante o desenvolvimento inicial de um cronograma de projeto, não são consideradas horas extras, que são permitidas apenas em casos pontuais no dia a dia dos projetos. Cada recurso possui suas próprias habilidades e cada tarefa possui uma lista de habilidades requeridas para sua execução. O limite máximo e mínimo de recursos alocados foi registrado com base nas necessidades de cada tarefa executada. Sempre é alocado pelo menos um recurso que possua cada uma das habilidades requeridas para execução da tarefa. Como os dados utilizados não especificavam proficiência e experiência dos recursos, foi considerado sempre o valor “normal”, em vista desses fatores não influenciarem no cálculo de produtividade das equipes.

Os dados dos projetos de teste foram exportados do MS Project [Microsoft 2011] para um arquivo texto, seguindo uma sintaxe pré-definida. No início da execução do algoritmo GRASP-MOVNS, o arquivo texto é importado de forma automatizada, iniciando os objetos modelados para o PDC. Após inicializar os dados do projeto, são carregados os parâmetros dos algoritmos desenvolvidos. Após a realização de alguns testes iniciais, foram definidos os seguintes valores para os parâmetros: $graspMax = 150$ e $levelMax = 10$. A variável *shaking* teve seu valor máximo fixado em 7, pois, durante a fase de construção, foram utilizadas apenas 7 movimentos dentre os 9 propostos. Os movimentos “Realocar Recurso de uma Tarefa” e “Troca de Recursos de uma Tarefa” foram desconsiderados na fase de construção, pois, durante os testes, observou-se que geram uma perturbação pequena nas soluções. Os valores das variáveis *shaking* e *levelMax* foram definidos altos, para permitir uma maior intensidade da perturbação na solução. O valor de *graspMax* indica o número de iterações executadas na fase de construção GRASP. Como critério de parada do algoritmo GRASP-MOVNS, foi utilizando um tempo de execução igual a 180 segundos.

Para avaliar o algoritmo proposto nesse artigo, foi criado um grupo formado por 3 experientes gerentes de projeto de software. Todos os indivíduos participantes receberam os cenários de testes e, utilizando somente seu conhecimento e experiência na atividade de gerência de projetos, construíram o cronograma do projeto. Para realizar a tarefa, os indivíduos dedicaram 4 horas para construir o cronograma e utilizaram o software MS Project 2010. As soluções obtidas foram coletadas e confrontadas com resultados do al-

goritmo, para validar a competitividade e a qualidade das soluções geradas pelo algoritmo GRASP-MOVNS. A próxima subseção apresenta e analisa os resultados obtidos.

5.2. Análise dos Resultados

As Figuras 1, 2 e 3 apresentam os gráficos confrontando os resultados dos gerentes com os obtidos em 30 execuções de cada cenário de teste. Pode-se afirmar que a abordagem proposta nesse artigo produziu soluções comparáveis as dos gerentes, inclusive melhores e mais diversificadas no aspecto geral. Em geral, o algoritmo GRASP-MOVNS produziu melhores soluções na minimização dos dois objetivos propostos, tempo e custo.

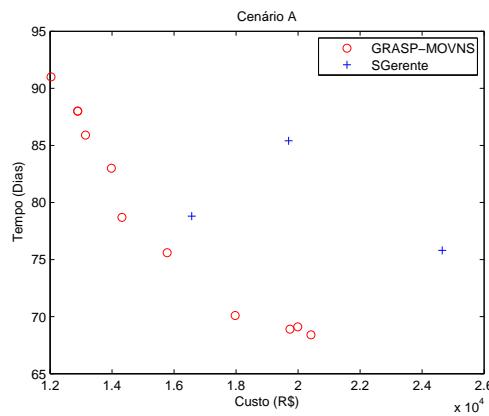


Figura 1. Comparação entre o algoritmo GRASP-MOVNS e as soluções do Gerentes.

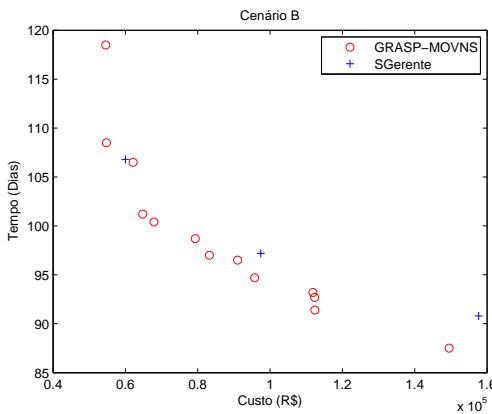


Figura 2. Comparação entre o algoritmo GRASP-MOVNS e as soluções do Gerentes.

Em relação à aplicabilidade da abordagem em projetos de desenvolvimento de software, principal objetivo de análise destes cenários de teste, tem-se como atingido o resultado esperado. Com sucesso, projetos reais de desenvolvimento de software foram adaptados à abordagem, obtendo-se soluções para o planejamento de diversas tarefas do projeto em relação à alocação de equipes e ao desenvolvimento de cronograma.

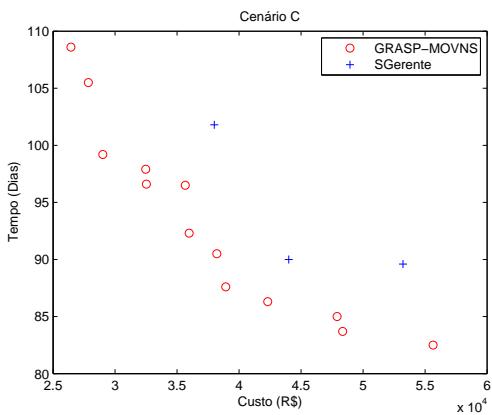


Figura 3. Comparação entre o algoritmo GRASP-MOVNS e as soluções do Gerentes.

6. Conclusões

Este trabalho propôs a utilização de uma abordagem multiobjetivo com o objetivo de encontrar a melhor alocação e o melhor sequenciamento das atividades do projeto, a fim de elaborar bons cronogramas que minimizem o tempo e o custo do projeto. O algoritmo GRASP-MOVNS, proposto nesse artigo para resolver o PDC, obteve resultados competitivos, apresentando resultados melhores que os encontrados por experientes gerentes de projetos.

O Desenvolvimento do Cronograma do Projeto de Software é um problema de difícil resolução, devido à grande quantidade de restrições envolvidas. As principais restrições são implementadas, como alocações em que mais de um recurso pode ser alocado por tarefa e um recurso pode trabalhar em mais de uma tarefa no mesmo dia, disponibilidade de recursos, carga horária e salário individual, quatro tipos de ligação de dependência entre tarefas, e tipos de tarefas. No entanto, existem muitas outras características que podem ficar para trabalhos futuros, como *overhead* de comunicação, coletar dados de projetos que permitam o uso dos coeficientes de experiência e proficiência dos recursos, a permissão de realização de horas extras e a avaliação do impacto dessa configuração para as soluções geradas pela abordagem.

Com o objetivo de melhorar a qualidade das soluções, podem ser feitos aprimoramentos no algoritmo, como a utilização de outras técnicas de otimização e a criação de novas heurísticas. Também pode ser melhor investigada a eficiência e eficácia do algoritmo proposto em relação a outras técnicas de otimização, em especial a técnicas que utilizem algoritmos genéticos. Finalmente, realizar novos estudos de caso, aplicando a abordagem em outros projetos reais com o objetivo de analisar o comportamento da abordagem em diferentes cenários.

Agradecimentos

Os autores agradecem a FAPEMIG, CNPq e CEFET-MG por apoiar o desenvolvimento dessa pesquisa.

Referências

- Alba, E. and Chicano, F. (2007). Software project management with GAs. *Information Sciences*, 177(11):2380–2401.
- Antoniol, G., Penta, M. D., and Harman, M. (2005). Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 240–249.
- Barros, M. O. and Dias-Neto, A. C. (2011). Desenvolvendo uma abordagem sistemática para avaliação dos estudos e experimentais em Search Based Software Engineering. *II Workshop de Engenharia de Software Baseada em Buscas - WESB*, 12:49–56.
- Coelho, V. N., Souza, M. J. F., Coelho, I. M., Guimarães, F. G., and Lust, T. (2012). Algoritmos multiobjetivos para o problema de planejamento operacional de lavra. In *Anais do XV Simpósio de Pesquisa Operacional e Logística da Marinha (SPOLM 2012)*.
- Colares, F. (2010). Alocação de equipes e desenvolvimento de cronogramas em projetos de software utilizando otimização. Dissertação de mestrado, UFMG.
- Feo, T. A. and Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.
- Geiger, M. J. (2008). Randomized variable neighborhood search for multi objective optimization. *Proceedings of the 4th EU/ME Workshop: Design and Evaluation of Advanced Hybrid Meta-Heuristics*, pages 34–42.
- Gueorguiev, S., Harman, M., and Antoniol, G. (2009). Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (CECOO)*, pages 1673–1680.
- Hansen, P. and Mladenovic, N. (1997). Variable neighborhood search. *Computers and Operations Research*, 24:1097–1100.
- Microsoft (2011). MS Project 2010. [Online; accessed 10-December-2011].
- Minku, L. L., Sudholt, D., and Yao, X. (2012). Evolutionary algorithms for the project scheduling problem: runtime analysis and improved design. In *GECCO'12*, pages 1221–1228.
- Penta, M. D., Harman, M., and Antoniol, G. (2011). The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. *Software - Practice and Experience*, 41:495–519.
- Souza, M. J. F., Coelho, I. M., Ribas, S., Santos, H. G., and Merschmann, L. H. C. (2010). A hybrid heuristic algorithm for the open-pit-mining operational planning problem. *European Journal of Operational Research*, 207(2):1041–1051.

A Framework for Selection of Software Technologies using Search-based Strategies

Aurelio da Silva Grande, Rosiane de Freitas Rodrigues, Arilo Claudio Dias-Neto

Institute of Computing, Federal University of Amazonas, CEP: 69077-000, Manaus-AM, Brazil.

aurelio.grande@gmail.com; {rosiane, arilo}@icomp.ufam.edu.br

Abstract. This paper presents a framework to instantiate software technologies selection approaches by using search techniques. To support this development, the software technologies selection problem (STSP) is modeled as a Combinatorial Optimization problem (minimum dominating set) aiming to attend different real scenarios in Software Engineering. The proposed framework works as a top-level layer over generic optimization frameworks that implement a high number of metaheuristics proposed in the technical literature, such as JMetal and OPT4J. It aims supporting software engineers that are not able to use optimization frameworks during a software project due to short deadlines and limited resources or skills.

Keywords. Search-Based Software Engineering; Software Technology Selection; Metaheuristics.

1. Introduction

The selection of software technologies to be applied in software projects consists in an important task that may directly affect the quality of the final software product. This decision-making may be related to different software development activities, such as: project management, requirements elicitation, architectural design, quality assurance, among others, as summarized in [Dias-Neto and Travassos 2009]. This scenario represents a hard decision to be performed by a software engineer, mainly when there are other restrictions, such as time, cost, and resources, in a software project.

In another perspective, in the last ten years it's growing the adoption of search-based techniques as mechanism to support decision-making in software engineering scenarios in which the number of options (possible solutions) available is too large and hard to be manually analyzed by software engineers considering the software project restrictions. This field was entitled as Search-Based Software Engineering (SBSE). Integrating both scenarios, the problem of selecting software technologies for software projects meets the two requirements presented by Harman [Harman and Jones 2001] to be included in the set of SBSE problems, as we will present during this paper.

The modeling and execution of experiments using search-based techniques can be done using some frameworks provided in the technical literature that allow a separate development of optimization algorithms and optimization tasks. They provide different optimization problems, algorithms, and results representation ways that support the conduction of these experiments. However, the adoption of these frameworks for a real scenario of software engineering requires a high effort that usually cannot be spent in a software project due the short deadlines and limited resources, mainly when these strategies are regarding decision making tasks performed by project managers, such as the selection of software technologies, because the skills required by these professionals

do not include knowledge on search-based algorithms and metaheuristics.

To close this gap, this paper proposes a framework tailored to the software technologies selection problem (STSP), modeled as a combinatorial optimization problem [Dias-Neto e Rodrigues, 2011], that instantiates recommendation systems by the application of SBSE techniques. The proposed framework works as a top-level layer that provides input data for other optimization frameworks available in the technical literature (cited previously). It acts as interface to support software engineers in the use of search-based strategies to provide the selection of technologies for software projects.

2. Related Works – Optimization Frameworks

There are several optimization frameworks available in the technical literature, such as JMetal [Durillo and Nebra 2011], Opt4J [Lukasiewycz et al. 2011], EvA2 [Kronfeld et al. 2010], amongst others. These frameworks provide features to use metaheuristics for different problems implemented in the tool (we also can implement new problems).

Analyzing their features regarding the feasibility of integrating them to our proposal, we find that all these optimization frameworks provide support for implementing various types of metaheuristics and some of them are already implemented, such as the JMetal's evolutionary algorithms. These frameworks are extensible, because they have Java classes that represent problems in which new problems, like STSP, can be implemented by coding. In general they provide graphic display of results, and each one has other resources more specific for representation of results.

Due to the short deadlines and limited resources in a software project, probably the software engineer responsible by the selection of technologies would not have time to implement new problems in optimization framework. Therefore, all cited frameworks have an important role in the framework proposed in this work. They would be integrated in our framework to perform experiments using different metaheuristics for the STSP, as will be described in the next section that presents the proposed framework.

3. Framework to Support the STSP

This section will detail the framework developed for selecting software technologies (framework STSP). It will be shown its extensible architecture and how it integrates with other frameworks, the main settings required to use the tool, steps necessary to create a new optimization scenario, design and execution of experiments, and the features for results representation. We consider this proposal as a framework because we can use it to instantiate software technologies selection approaches for different scenarios.

3.1. Modeling of Software Technologies Selection Problem

According Dias-Neto e Rodrigues (2011), the STSP can be modeled as a specific case of the classical set covering problem (SCP), using a model in graphs, known in the literature as the minimum dominating set problem (MDSP). The STSP was modeled in bipartite graph that represents the dimensions: source (technologies), destination (projects) and objectives (functions). Thus, a challenge for a framework to support the STSP would be to provide the common selection characteristics (dimensions) required in any software engineering activity and still make possible the necessary specializations for each activity.

3.2. Framework's Architecture

The STSP framework is Java web tool that was designed to work as a top-level layer acting together with other optimization frameworks, serving as an interface for software engineers. It uses various metaheuristics available in these optimization frameworks

available in the technical literature. Currently, the STSP framework is integrated with JMetal framework and three algorithms implemented by it: NSGAII, SPEA2 and MOCell. Since they are evolutionary approaches, a chromosome is represented by an integers' array with size equal to the number of variables to be combined. The value of each allele on chromosome represents the index of a technique that identifies it in the repository of technologies.

The framework comprises methods responsible for calculating the values for the optimization goals, that is, the fitness functions (Figure 1). It evaluates the functions fitness using the *Jaccard Coefficient*¹ formula [Jaccard, 1901]. Thus, it counts how many attributes required by the software project are being attended by the selected technologies.

$$Fit.\ Function = \frac{\sum \text{Attrs Required by the Project} \cap \text{Attrs Supplied by the Technologies}}{\sum \text{Attrs Required by the Project}}$$

Figure 1. Mathematical Formula of STSP Fitness Function.

3.3. Configuration of New Scenarios and Experiments Design

To configure new software engineering scenarios for performing experiments (selection of software technologies), the framework user must follow this process:

(i) Creating a scenario and identify the attributes that will be used to characterize both technologies and projects; (ii) Establishing a body of knowledge of software technologies for the created scenario, containing their respective technologies found in the technical literature, as well as the characterization of their attributes. This step represents the instantiation of the software engineering scenario; (iii) Characterizing the software project (or a portfolio of projects) that you want to apply the selected technologies by the attributes of the scenario.

Once the scenario is configured, the STSP framework is ready to carry out the experiments based on search strategies. An experiment is defined by the definition of the 3 dimensions: Source (if the generated solutions will comprise a single technique or combination of techniques), Destination (in which software projects the selected software technologies will be applied to) and Objectives (combination of the attributes). Then, software engineers can also chose which external frameworks and metaheuristics integrated to the STSP framework they want to execute for the configured scenario. Furthermore, the parameters required to use these algorithms (these parameters are defined by the original framework that implements the algorithms) must be filled out, such as: population size, number of generations, percentage of crossover and mutation, amongst others.

3.4. Results Representation

After the experiments execution, the results are imported from the external framework and they are stored in a database and displayed on the STSP framework screen. In this screen, users can view a chart with the solutions generated for each algorithm. The chart used to represent the results varies according to the number of optimization objectives defined for the scenario: (i) If an experiment has only one objective (mono-objective), just an ordered list with the best generated solutions is displayed; (ii) If it has two objectives, the list of best compromise (trade-off) solutions and a line chart in the XY

¹ *Jaccard Coefficient* (1901) compares the number of similar elements and the total of elements in a set (it is called *similarity coefficient*).

plane (two dimensions) are displayed; **(iii)** If the experiment has more than 2 objectives, the list of best compromise (trade-off) solutions is presented. Moreover, the results will be graphically displayed in radar chart. The system can be accessed (with only read privileges) on www.icomp.ufam.edu.br/experts/stsp using the login *stsp* and password *stsp*.

4. Conclusions

This paper presented a framework has been developed to support the selection of software technologies. It works as a bridge between software engineers, who usually has limited knowledge on search techniques, and the available optimization frameworks, that implement in their core some algorithms and combinatorial optimization problems. The STSP framework works together with external optimization frameworks and it is already integrated with JMetal framework. Thus, we have already a modeled and implemented solution to support the STSP, as well as a mechanism to evaluate the quality of the suggested solutions using fitness functions that can be defined using the framework. Graphical representation of the best (trade-off) solutions generated by an algorithm using charts is provided according to the number of objectives used in the scenario modeling. Extensibility is one important characteristic of the STSP framework. New optimization frameworks and algorithms can be integrated to it. At this moment, it is integrated to JMetal framework and its evolutionary algorithms. As ongoing works, we intend to execute new experiments evaluating the meta-model adequacy for other scenarios. Finally, new experiments are being planned to compare the effectiveness of the infrastructures instantiated from the proposed framework to other solutions.

Acknowledgments

The authors acknowledge the support granted by FAPEAM, CNPq to the INCT-SEC (National Institute of Science and Technology – Critical Embedded Systems – Brazil), and INCT-Web Science, processes 573963/2008-8 and 08/57870-9.

References

- Dias-Neto, A. C. ; Rodrigues, R. F. (2011). Uma Proposta de Framework de Apoio à Seleção de Tecnologias de Software aplicando Estratégias de Busca. In: Workshop de Engenharia de Software Baseada em Busca (WESB), 2011, São Paulo. CBSoft.
- Dias-Neto, A.C.; Travassos, G.H. (2009). “Model-based Testing Approaches Selection for Software Projects”, In: Information and Software Technology, v.51, p.1487-1504.
- Durillo, J.J.; Nebro, A.J.; (2011); “[jMetal: a Java Framework for Multi-Objective Optimization](#)”. Advances in Engineering Software, v. 42, pp. 760-771.
- Harman, M., Jones, B.F. (2001), Search-based software engineering, Information and Software Technology, 2001, pp. 833-839.
- Jaccard, Paul (1901), Étude comparative de la distribution florale dans une portion des Alpes et des Jura, Bulletin del la Société Vaudoise des Sciences Naturelles 37, pp. 547-579.
- Kronfeld, M.; Planatscher, H.; Zell, A.; (2010); “The {Eva2} Optimization Framework. Learning and Intelligent Optimization Conference”, Special Session on Software for Optimization (LION-SWOP). Lecture Notes in Computer Science, LNCS. Pp: 247-250. Venice, Italy.
- Lukasiewycz M.; Glaß, M.; Reimann, F.; Teich J. (2011); “Opt4J - A Modular Framework for Metaheuristic Optimization”. Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011), Dublin, Ireland, p: 1723-1730.

Mapeamento da Comunidade Brasileira de SBSE

Wesley K. G. Assunção¹, Márcio de O. Barros², Thelma E. Colanzi^{1,3}, Arilo C. Dias-Neto⁴, Matheus H. E. Paixão⁵, Jerffeson T. de Souza⁵, Silvia R. Vergilio¹

¹Departamento de Informática (DInf), Universidade Federal do Paraná (UFPR),
CP: 19081, CEP: 81531-980, Curitiba-PR, Brasil

²Departamento de Informática Aplicada, Universidade Federal do Estado do Rio de Janeiro, CEP: 22240-090, Rio de Janeiro-RJ, Brasil

³Departamento de Informática (DIN), Universidade Estadual de Maringá (UEM),
CEP: 87020-900, Maringá-PR, Brasil

⁴Instituto de Computação, Universidade Federal do Amazonas,
CEP: 69077-000, Manaus-AM, Brasil

⁵Grupo de Otimização em Engenharia de Software (GOES.UECE),
Universidade Estadual do Ceará, Fortaleza-CE, Brazil

{wesleyk, thelmae, silvia}@inf.ufpr.br, marcio.barros@uniriotec.br,
arilo@icomp.ufam.edu.br, mhepaixao@gmail.com, jerffeson.souza@uece.br

Abstract. Research communities evolve over time, changing their interests for specific problems or research areas. Mapping the evolution of a research community, including the most frequently addressed problems, the strategies selected to propose solution for them, and the collaboration among distinct groups may provide lessons on actions that can positively influence the growth of research in a given field. To this end, this paper presents an analysis of the Brazilian SBSE research community. We present our major research groups working in this field, the software engineering problems most addressed by them, and the search techniques most frequently used to solve these problems. We could conclude that the Brazilian community is still expanding, both geographically and in terms of publications.

Resumo. Comunidades de pesquisa evoluem ao longo do tempo, mudando seus interesses para problemas ou áreas de pesquisa específicos. Mapear a evolução de uma comunidade de pesquisa, incluindo os problemas mais frequentemente abordados, as estratégias selecionadas para propor soluções para eles e a colaboração entre grupos distintos, pode prover lições sobre ações que podem influenciar positivamente o crescimento das pesquisas em uma dada área. Com este objetivo, este artigo apresenta uma análise da comunidade brasileira em SBSE. São apresentados os principais grupos que realizam pesquisas neste tema, os problemas de engenharia de software mais abordados por eles e as técnicas de busca mais frequentemente usadas para resolver estes problemas. As conclusões indicam que a comunidade brasileira ainda está em expansão geograficamente e em termos de publicações.

1. Introdução

Na literatura técnica, podem ser encontrados estudos que reportam um crescimento do número de trabalhos e diversidade de áreas da Engenharia de Software sendo pesquisadas no campo da Engenharia de Software Baseada em Busca (em inglês, *Search Based Software Engineering – SBSE*) (Harman et al., 2009). Analisando o repositório SEBASE¹, pode ser observado, como acontece com muitos outros campos de pesquisa, que os primeiros trabalhos em SBSE foram publicados em conferências Europeias e Norte-Americanas: 166 dos 184 artigos de conferência publicados até 2005 foram reportados em eventos europeus e norte-americanos. A pesquisa foi expandida para a Ásia e Oceania na segunda metade da última década e 62 dos 454 artigos publicados até 2009 foram reportados em conferências realizadas nestes locais. Mais recentemente, uma forte participação da América do Sul pode ser percebida, essencialmente concentrada no Brasil; apenas um artigo de conferência foi publicado em um evento na Argentina e neste trabalho não foi identificado outro país da América do Sul no qual artigos de SBSE foram publicados.

Uma rápida análise no SEBASE mostra que 6% das suas publicações (de um total de 1093 artigos até a data analisada) são creditados a autores brasileiros. Este número expressivo se deve à organização, nos últimos três anos, de um evento nacional na área: o Workshop Brasileiro de Engenharia de Software Baseada em Busca (WESB). Desde então (2010), um aumento no número de artigos de SBSE publicados por autores brasileiros pode ser observado (Freitas e Souza, 2011; Colanzi et al., 2012) e a área de SBSE está crescendo rapidamente no Brasil.

Além disso, o Brasil tem conseguido recentemente atenção internacional devido aos grandes eventos que aqui serão realizados, tais como A Copa do Mundo da FIFA em 2014 e os Jogos Olímpicos de 2016, e isso tem atraído investimentos de diferentes fontes. Isto aumenta o interesse em pesquisas realizadas no Brasil. Este crescente interesse e o aumento do número de trabalhos brasileiros na área de SBSE servem como motivação para este artigo, que apresenta resultados de um mapeamento da comunidade brasileira de SBSE. Este mapeamento visa a responder à seguinte questão de pesquisa: Quais são os grupos de pesquisa trabalhando com SBSE no Brasil? Esta questão permite a identificação dos grupos existentes: instituições e regiões do país; áreas de Engenharia de Software abordadas; técnicas de busca usadas; e número de pesquisadores.

As principais contribuições da pesquisa e análise conduzida e relatada neste artigo são a apresentação dos pesquisadores e grupos brasileiros trabalhando em SBSE e a descrição de uma visão geral dos trabalhos produzidos por eles, contribuindo para a consolidação da área no Brasil.

Este artigo está organizado da seguinte forma. A Seção 2 descreve a metodologia adotada neste estudo, incluindo como os dados foram coletados e as categorias usadas na análise. A Seção 3 analisa os dados coletados para responder à questão de pesquisa definida. A Seção 4 apresenta os trabalhos relacionados. Finalmente, a Seção 5 descreve as considerações finais.

¹Repositório SEBASE SBSE, http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/.

2. Metodologia

O principal objetivo de um estudo de mapeamento é prover uma visão geral de uma área de pesquisa, identificando a quantidade e tipo de pesquisa e resultados disponibilizados por elas (Petersen et al., 2008). No caso deste trabalho, o objetivo é fornecer uma visão geral da área de SBSE no Brasil. Para atingir este objetivo, sete pesquisadores estiveram envolvidos na realização dos passos de mapeamento, de acordo com a estratégia delineada por Petersen et al. (2008), para responder à questão de pesquisa apresentada na Seção 1. A seguir, são descritos os passos que foram conduzidos neste estudo.

2.1. Condução das Buscas e Análise dos Trabalhos

A busca por publicações relevantes foi realizada usando o repositório SEBASE de SBSE. Esta escolha foi feita por duas razões: a primeira é que as bibliotecas digitais, como IEEEXplorer e ACM Digital Library, podem não incluir todos os trabalhos de uma área específica. A segunda razão é que o repositório escolhido é considerado uma base comprehensiva na área de SBSE e tem sido usado como uma referência por um trabalho similar (Freitas e Souza, 2011). Ele contém publicações de diferentes fontes e é frequentemente atualizado.

Como um primeiro passo, foi solicitado junto ao pesquisador responsável por manter e atualizar o repositório SEBASE uma lista de artigos publicados por brasileiros. Então, a lista foi verificada, considerando um único critério de inclusão: ao menos um dos autores é um pesquisador brasileiro. Nós definimos um pesquisador brasileiro como alguém que trabalha ou estuda em um instituto de pesquisa brasileiro, tipicamente uma universidade ou um laboratório de pesquisa para uma empresa pública ou privada. Esta definição elimina pesquisas realizadas por brasileiros que atuam em empresas e universidades fora do país, exceto se estes pesquisadores, em algum momento, voltaram ao seu país natal e participaram em pesquisas locais chegando ao ponto de se tornar autor de pelo menos um artigo. Esta busca foi concluída em Fevereiro de 2013 e 73 trabalhos foram identificados.

2.2. Esquema de Classificação, Extração e Mapeamento de Dados

De acordo com a questão de pesquisa deste estudo, análises foram conduzidas considerando o esquema de classificação com diferentes categorias, descritas a seguir:

- **Grupos de Pesquisa:** foram identificadas as principais instituições presentes nos artigos selecionados e a regularidade das publicações. A partir disso, os principais grupos de pesquisa foram identificados;
- **Áreas da engenharia de software:** este aspecto reporta as principais áreas de interesse dos grupos identificados. As áreas foram identificadas usando o Sistema de Classificação de Computação da ACM². Áreas contendo um ou dois artigos foram agrupadas em uma categoria chamada *Outros assuntos*;
- **Técnicas de busca:** esta dimensão mostra quais são as preferências e expertise de cada grupo, e quais são as técnicas de busca mais usadas em cada área. Fora consideradas categorias tais como clássico e meta-heurística. Também foi

² www.computer.org/portal/web/publications/acmsoftware

decidido classificar as meta-heurísticas de acordo com o tipo de algoritmo evolutivo usado e o número de objetivos. Nesta análise os *surveys* foram excluídos, assim como outros artigos que não mencionam claramente o algoritmo aplicado.

De acordo com este esquema de classificação, as seguintes informações foram extraídas de cada artigo e copiadas para uma planilha para análise futura: autores; afiliações; ano de publicação e veículo (nome da conferência ou período); cidade/estado; área da engenharia de software; técnica baseada em busca usada. Após isso, os dados foram analisados em diferentes perspectivas para responder à questão de pesquisa. Essas perspectivas são apresentadas na próxima seção.

3. Comunidade Brasileira de SBSE

Esta seção contém uma análise das informações obtidas para responder à questão de pesquisa deste estudo. Primeiramente foram analisadas as principais áreas e técnicas de busca abordadas pela comunidade brasileira. Em seguida, foram identificados os principais grupos de pesquisa, bem como suas expertises e interesses.

3.1. Áreas da Engenharia de Software

Os artigos brasileiros sobre SBSE foram agrupados por área da Engenharia de Software de acordo com o esquema de classificação escrito na Seção 2.2. As áreas selecionadas são: Teste de Software, Gerenciamento de Projetos, Engenharia de Requisitos, Projeto de Software e Outros Assuntos. Cinco *surveys* sobre SBSE foram publicados por pesquisadores brasileiros, mas esses *surveys* não foram incluídos nesta análise sobre áreas da engenharia de software.

Teste de software é a área que recebe mais atenção dos pesquisadores: 45% de todos os trabalhos são dedicados a ela. Isto também acontece no cenário internacional (Harman et al., 2009). A maioria dos trabalhos aborda geração de dados de teste. Outros trabalhos focam em priorização e seleção de casos de teste, teste de integração, seleção de estratégias de teste, alocação de casos de teste e seleção e avaliação de casos de teste.

Teste de software é seguido por Gerenciamento de Projeto (18%) e Engenharia de Requisitos (16%). Gerenciamento de Projeto é uma área na qual abordagens baseadas em busca foram aplicadas para otimizar tanto o processo de desenvolvimento de software como o produto desenvolvido. Trabalhos abordam diferentes tarefas relacionadas ao planejamento do projeto, incluindo definição de cronograma e alocação de tarefas e recursos. Outros trabalhos estão relacionados à seleção de portfolio de projetos e tecnologias de software.

Entre os trabalhos em Engenharia de Requisitos, O Problema do Próximo Release (*Next Release Problem*) tem sido abordado por pesquisadores brasileiros. O objetivo é encontrar um conjunto ideal de requisitos para uma versão de software considerando diferentes objetivos, tais como necessidades do cliente, restrições de recurso e características dos requisitos. Mais recentemente, priorização dos requisitos também tem sido um tema abordado (em 2011 e 2012).

Desde 2010 trabalhos envolvendo outras áreas da engenharia de software aparecem no cenário brasileiro de SBSE. No entanto, pode ser observado que o interesse

em Teste de Software, Gerenciamento de Projetos e Engenharia de Requisitos não diminuiu. Portanto, esta mudança apenas indica que novas áreas de Engenharia de Software que aplicam SBSE estão emergindo. Em 2012 os primeiros artigos sobre Projeto de Software foram publicados (9%), incluindo refatoração arquitetural e clusterização de módulos de software. Outros assuntos, tais como refatoração de código, predição de confiabilidade de software, busca por componentes, clusterização de componentes, gerenciamento de banco de dados e avaliação de SBSE, têm sido abordados recentemente. Eles representam 12% de todos os artigos selecionados.

3.2. Técnicas de busca

Nesta seção, os artigos de SBSE são analisados para determinar os algoritmos baseados em busca mais utilizados por pesquisadores brasileiros. As técnicas evolutivas mais utilizadas são: Algoritmos Genéticos (*Genetic Algorithm* – GA), Algoritmos Evolutivos Multi-Objetivos (*Multi-Objective Evolutionary Algorithms* – MOEAs) e Programação Genética (*Genetic Programming* – GP) representam 23%, 36% e 6%, respectivamente. NSGA-II é o MOEA mais usado, mas outros algoritmos frequentemente usados são SPEA2 e MOCell. Além de técnicas evolutivas, 21% dos estudos aplicam outras meta-heurísticas, tais como Arrefecimento Simulado (*Simulated Annealing*), Subida da Montanha (*Hill Climbing*), GRASP, Busca Aleatória, etc. Além disso, quatro artigos (6%) usaram Otimização por Colônia de Formigas (*Ant Colony Optimization* – ACO) e um artigo empregou *Pareto ACO* (PACO) (1%). Técnicas clássicas são aplicadas por 8% dos artigos: nesta categoria algoritmo Guloso e *Branch-and-Bound* são os mais aplicados.

A Figura 1 apresenta o número de artigos em SBSE que usam algum tipo de algoritmo de otimização por área da engenharia de software. MOEAs foram aplicados em artigos de quase todas as áreas. Considerando todos os artigos, todos os tipos de algoritmos de otimização foram usados para resolver problemas relacionados a Teste de Software. Neste caso, MOEAs foram os mais usados. Várias outras meta-heurísticas foram também aplicadas para resolver problemas de teste de software. Para Gerenciamento de Projeto, o algoritmo mais aplicado foi GA. Para Engenharia de Requisitos, algoritmos multi-objetivos são normalmente aplicados; ACO, GA, Arrefecimento Simulado e GRASP foram também aplicados. Finalmente, para Projeto de Software, MOEAs foram novamente os mais usados.

3.3. Grupos de Pesquisa

Para mapear os grupos de pesquisa trabalhando com SBSE no Brasil, primeiramente foram identificadas as instituições de cada autor que apareceu entre os artigos selecionados. Para cada instituição, foi contado o número de artigos publicados, assim como a frequência e regularidade das publicações. O resultado está apresentado na Figura 2. Esta figura apresenta as instituições, número e percentual de artigos publicados, assim como o ano da primeira publicação. Artigos contendo mais que um autor com afiliação em diferentes grupos foram contados para todos os grupos. O tamanho dos círculos representa o número de autores.

Pode ser observado que o maior número de autores está na UECE (em Fortaleza/Ceará, com 16 autores) e em UFPR (em Curitiba/Paraná, com 10 autores), as mesmas universidades que possuem os maiores números de publicações. Além disso, UNICAMP possui 8 autores, UNIRIO 6, seguidos por UFPB, UFPI e UFRJ com 4.

UFAM possui 2 pesquisadores. O primeiro artigo foi publicado pelo grupo da UNICAMP em 2000. Quatro instituições (das 13) publicaram o primeiro artigo em 2010, quando a primeira edição do WESB foi promovida; elas foram seguidas por duas instituições em 2011 e duas em 2012. Seis instituições publicaram apenas um ou dois artigos. Nesta análise de caracterização dos grupos de pesquisa tais instituições não foram incluídas, visto que foi considerado que estas não possuem ainda um grupo consolidado com claro interesse em SBSE. Portanto, elas não aparecem na Figura 3, que descreve informações sobre as áreas de engenharia de software e técnicas abordadas pelos grupos de pesquisa.

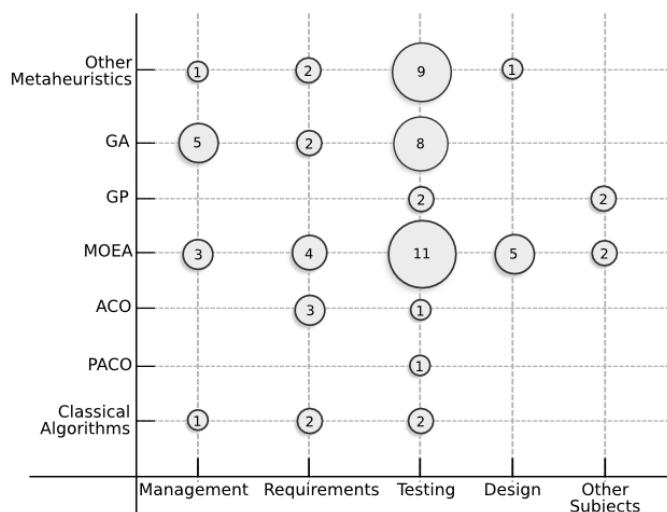


Figura 1. Gráfico de bolha indicando o uso de técnicas de busca para tratar problemas de engenharia de software conforme reportado em artigos brasileiros de SBSE

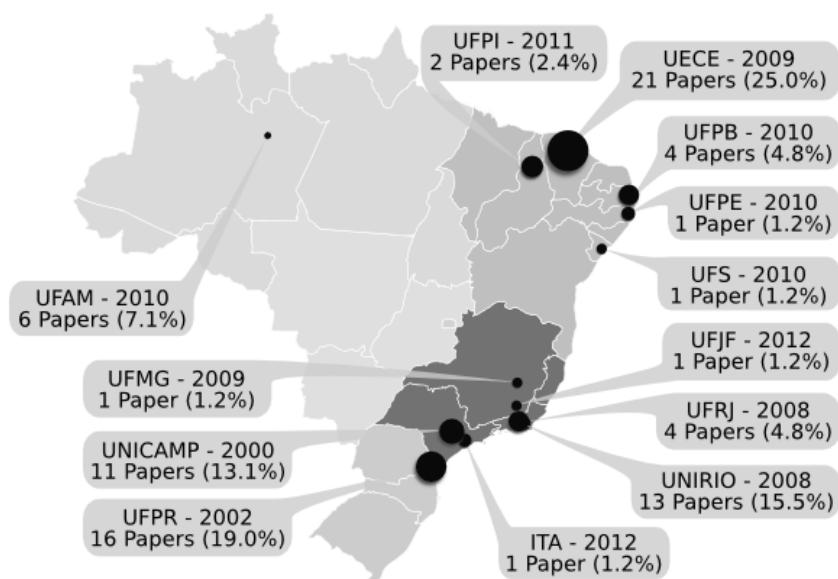


Figura 2. Grupos de Pesquisa no Brasil – Número e percentual de publicações por região, juntamente com o número de pesquisadores em SBSE (tamanho da bolha)

Levando-se em consideração a área da Engenharia de Software, o grupo da UECE possui publicações em quase todas elas, mas foi observado um grande interesse por Engenharia de Requisitos, seguido por Teste de Software. Teste é também a área mais abordada por grupos das seguintes instituições: UFPR, UNICAMP e UFAM. O grupo da UFPR também possui interesse em outras áreas, tais como Projeto de Software e Outros Assuntos (Qualidade de Software e Refatoração). Os principais interesses do grupo da UNIRIO são Gerenciamento de Projeto e Projeto de Software. Gerenciamento de Projeto é também abordado pelos grupos da UFPB e UFRJ. Todos os grupos possuem publicações em pelo menos duas áreas da engenharia de software.

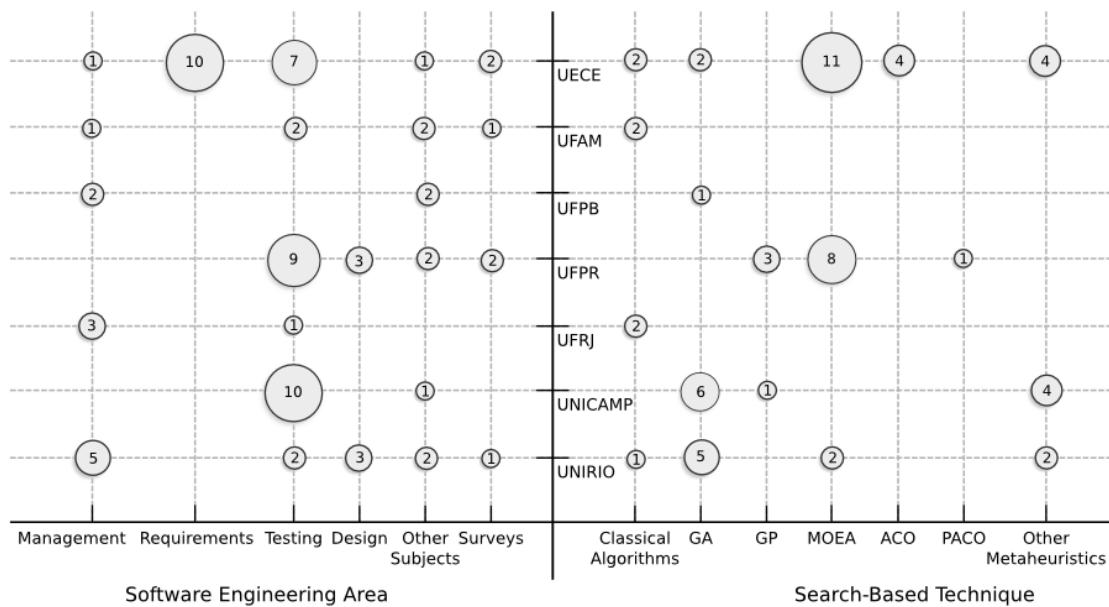


Figura 3. Áreas e Técnicas abordadas pelos grupos

A respeito das técnicas de busca, pode ser observado que MOEAs são as técnicas mais adotadas pelos grupos da UECE e UFPR. UECE possui ainda um número expressivo de publicações usando ACO e Outras Meta-heurísticas. O algoritmo multiobjetivo PACO foi usado apenas pelo grupo da UFPR. Este grupo possui ainda um número expressivo de trabalhos que usam Programação Genética (GP). O grupo de pesquisa da UNIRIO tem concentrado seus esforços em GA e busca local (*Hill Climbing*). UNICAMP e UFPB têm utilizado principalmente GA. Finalmente, UFAM e UFRJ publicaram artigos apenas com Algoritmos Clássicos.

4. Trabalhos Relacionados

Nos últimos anos, um grande número de *surveys* em SBSE foi publicado como análise bibliométrica, estudo de mapeamento ou revisão sistemática da literatura. Por exemplo, em (Freitas e Souza, 2011), os autores descrevem a primeira análise bibliométrica sobre publicações em SBSE. Este estudo cobriu 677 publicações da comunidade de SBSE de 2001 até 2010. Os autores focaram em quatro categorias: Publicação, Fontes, Autorias e Colaboração. Adicionalmente, algumas estimativas de métricas de publicações para os próximos anos são apresentadas. O estudo ainda analisou a aplicabilidade das leis bibliométricas em SBSE.

Em (Colanzi et al., 2012), os autores apresentam resultados de um mapeamento que eles realizaram para prover uma visão geral da área de SBSE no Brasil. O principal objetivo foi mapear a comunidade brasileira de SBSE por meio da identificação dos pesquisadores mais ativos, do foco dos trabalhos publicados, veículos e frequência de publicações. Os autores usarem o Simpósio Brasileiro de Engenharia de Software como uma base para identificar trabalhos relacionados à otimização em Engenharia de Software. Tendo um escopo mais amplo, as análises apresentadas no presente artigo são baseadas em um repositório mantido pela comunidade internacional, na qual trabalhos sobre predição e clusterização estão também incluídos (além dos trabalhos em otimização). Por isso, resultados diferentes são obtidos quando comparados aos resultados apresentados em (Colanzi et al., 2012). Assim, a principal contribuição do presente artigo, que ainda não foi apresentada anteriormente, é a identificação dos grupos de pesquisa e colaborações existentes entre eles. Novos autores e grupos foram também identificados, o que é muito importante para expandir e consolidar a comunidade brasileira de SBSE.

Revisões Sistemáticas da Literatura (RSL) também têm sido realizadas no contexto de SBSE. Uma RSL é “uma forma de avaliar e interpretar todas as pesquisas relevantes disponíveis para uma questão de pesquisa particular, área ou fenômeno de interesse” (Dyba et al., 2005). Por exemplo, em (Afzal et al., 2009) os autores apresentam uma RSL que examinou trabalhos existentes em teste de software não-funcional baseado em busca (TSNFBB). Eles analisaram tipos de testes não-funcionais abordados usando técnicas de busca, diferentes funções de avaliação (*fitness functions*) usadas em tipos diferentes de TSNFBB, e desafios na aplicação dessas técnicas. A RSL foi baseada em um conjunto de 35 artigos publicados a partir de 1996 até 2007. Os resultados indicam que técnicas de busca têm sido aplicadas para testes não-funcionais em diferentes aspectos, incluindo tempo de execução, qualidade do serviço, segurança e usabilidade. Um grande número de técnicas de busca foi encontrado sendo aplicados a estes problemas, incluindo arrefecimento simulado, busca tabu, GAs, ACO, evolução gramatical, GP (e suas variações) e métodos de inteligência de enxame. A revisão reportou diferentes funções de avaliação usadas para guiar a busca para cada aspecto identificado.

Ali et al. (2010) descrevem o resultado de uma RSL que visou caracterizar como estudos experimentais têm sido projetados para investigar custo-eficiência de teste de software baseado em busca (*search-based software testing – SBST*) e quais evidências experimentais estão disponíveis na literatura a respeito do custo-eficiência e escalabilidade de SBST. Os autores ainda proveem um framework que direciona o processo de coleta de dados e pode ser o ponto de partida para diretrizes sobre como técnicas de SBST podem ser avaliadas experimentalmente. De acordo com os autores, a intenção seria auxiliar futuros pesquisadores na condução de estudos experimentais em SBST provendo uma visão sem viés do corpo de evidência experimental e guiando-os na realização de estudos experimentais bem projetados e executados.

Finalmente, Harman (2007) descreve um *survey* que analisou um conjunto de trabalhos na aplicação de técnicas de otimização em engenharia de software. O artigo brevemente revisa técnicas de otimização usadas amplamente e os ingredientes chaves requeridos para suas aplicações com sucesso em engenharia de software, provendo uma visão geral dos resultados existentes em oito domínios da engenharia de software. O

artigo ainda descreve os benefícios que possam advir do crescente corpo de trabalho nesta área e prove um conjunto de problemas em aberto, desafios e trabalhos futuros.

5. Conclusões

A comunidade brasileira de SBSE tem sido reconhecida nos últimos anos como uma das mais ativas do mundo. Tanto em termos quantitativos como em qualitativos, esta comunidade de pesquisa tem apresentado um crescimento recente e significante que a qualifica como um caso interessante a ser apresentado e refletido. Neste sentido, este artigo discutiu esta comunidade, apresentando os pesquisadores e grupos brasileiros em SBSE e provendo uma visão geral dos trabalhos que estão sendo produzidos por eles.

Em especial, este trabalho serve para destacar aspectos interessantes da comunidade, incluindo a importância do Workshop Brasileiro de Engenharia de Software Baseada em Buscas (WESB), que teve sua primeira edição em 2010 e ajudou significativamente no crescimento e consolidação da comunidade brasileira de SBSE. Além disso, o mapeamento mostrou uma dispersão significativa nos grupos ativos em SBSE em termos geográficos, o que fornece uma perspectiva de uma disseminação ainda maior no futuro.

Além disso, este pode servir ainda como motivação para que outros grupos de pesquisa nacionais ou institucionais apresentem suas contribuições para a área de SBSE, o que pode ajudar a comunidade de pesquisa de SBSE a se entender e usar este conhecimento para direcionar sua própria evolução. Como a comunidade brasileira de SBSE se prepara para organizar o principal evento internacional nesta área, o *Symposium on Search-Based Software Engineering* (SSBSE), em 2014, este trabalho pode servir tanto como apresentação como um convite para este importante evento, que irá contribuir para fortalecer e consolidar a pesquisa em SBSE no Brasil.

Sobre o futuro da pesquisa e prática de SBSE no Brasil, foram identificadas algumas tendências, analisando os artigos publicados recentemente. Essas tendências incluem otimização de projeto em alto-nível, seleção de estratégias de refatoração, avaliação de estudos experimentais em SBSE, aplicação de abordagens de SBSE em casos reais na indústria e uso de novas abordagens de otimização heurísticas (tais como ACO) para lidar com problemas clássicos (tais como o problema do próximo *release*) em casos mais complexos. Eles podem direcionar os próximos passos em pesquisa, o que pode incluir mais interação com a indústria local, mais colaboração com parceiros internacionais, um entendimento profundo de certos problemas clássicos (tais como clusterização de módulos de software e o problema do próximo *release*), e problemas de customização e suas soluções para as especificidades da indústria brasileira.

Agradecimentos

Os autores gostariam de agradecer ao CNPq pelo apoio financeiro recebido.

Referências Bibliográficas

- Afzal, W.; Torkar, R. and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.* 51, 6 (June 2009), 957-976. DOI=10.1016/j.infsof.2008.12.005.
- Ali, S.; Briand, L. C.; Hemmati, H. and Panesar-Walawege, R. K. (2010). “A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation”, *IEEE Transactions on Software Engineering*, vol. 36 (6), November/December 2010, pp. 742-762.
- Colanzi, T. E.; Vergilio, S. R.; Assunção, W. K. G.; Pozo, A. (2012). Search Based Software Engineering: Review and analysis of the field in Brazil, In: *Journal of Systems and Software*, pp. 1-15.
- Dybå, T.; Kitchenham, B.; Jorgensen, M. (2005). Evidence-based software engineering for practitioners, *IEEE Software*, Vol. 22 (1), pp. 158–165.
- Freitas, F. G.; Souza. J. T. (2011). Ten years of search based software engineering: a bibliometric analysis. In: *Proceedings of the Third international conference on Search based software engineering (SSBSE'11)*, Myra B. Cohen and Mel Ó Cinnéide (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 18-32.
- Harman, M., Mansouri, S. A. and Zhang, Y. (2009). Search based software engineering: A comprehensive analysis and review of trends techniques and applications, *Tech. Rep. TR-09-03*, King's College London (April 2009).
- Harman, M. (2007). The Current State and Future of Search Based Software Engineering. In: *Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, pp. 342-357. DOI=10.1109/FOSE.2007.29.
- Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. 12th International Conference on Evaluation and Assessment in Software Engineering (EASE), pp. 1-10.

Um Algoritmo Genético Coevolucionário com Classificação Genética Controlada aplicado ao Teste de Mutação

André Assis Lôbo de Oliveira¹, Celso Gonçalves Camilo-Junior¹, Auri Marcelo Rizzo Vincenzi¹

¹Instituto de Informática – Universidade Federal de Goiás (UFG)

Caixa Postal 131 – 74.001-970 – Goiânia – GO – Brasil

andre.assis.lobo@gmail.com, {celso,auri}@inf.ufg.br

Abstract. This paper is situated in the field of genetic algorithms that aim coevolutionary selection of good subsets of test cases and mutants in the context of Mutation Testing. It was selected and evaluated two approaches from this field of study. Such evaluation supported the development of a new Coevolutionary Genetic Algorithm with Genetic Controled Classification (CGA_GCC). The experiments compares the results of proposed algorithm with three other methods applied to two real benchmarks. The results shows a significant improvement of CGA_GCC compared to other approaches, due to the increase in mutation score without significative increase in runtime.

Resumo. Este artigo situa-se no campo dos algoritmos genéticos coevolucionários que objetivam a seleção de bons subconjuntos de casos de teste e mutantes, no contexto do Teste de Mutação. Desse campo de estudo, selecionou-se e avaliou-se duas abordagens existentes. Tal avaliação, subsidiou o desenvolvimento de um novo Algoritmo Coevolucionário com Classificação Genética Controlada (AGC_CGC). A experimentação compara os resultados do algoritmo proposto com outros três métodos aplicados em dois benchmarks reais. Os resultados revelam uma melhora significativa do AGC_CGC sobre as outras abordagens quando se considera o aumento do escore de mutação sem aumentar acentuadamente o tempo de execução.

1. Introdução

Atividades de Validação e Verificação (V&V) consomem cerca de 50% à 60% do custo total no ciclo de vida de um software [Papadakis, Malevris e Kallia 2010], sendo o teste de software uma das mais utilizadas. Neste contexto, surge a *Search-Based Software Testing (SBST)* [McMinn 2011], uma abordagem de pesquisa que utiliza as metaheurísticas como técnicas de otimização para resolver alguns dos problemas do Teste de Software com o objetivo de minimizar tais custos. Dentre as metaheurísticas da SBST, destacam-se os Algoritmos Genéticos (AGs), uma das técnicas mais utilizadas e eficientes em problemas de otimização combinatória [De Jong 2006] com alta complexidade de busca.

Entre as várias técnicas e critérios do Teste de Software, este trabalho aborda o Teste de Mutação (TM), um critério de teste conhecido por sua grande eficácia em detectar defeitos. No entanto, o alto custo computacional, proveniente da grande quantidade de programas mutantes gerados, torna o TM pouco utilizado na prática. Além disso, dentre os mutantes gerados, existem os equivalentes, indesejáveis porque não contribuem para evolução de conjunto de teste exigindo esforço humano para sua

classificação como equivalentes. Tal esforço se dá porque a verificação automática da equivalência entre programas é um problema indecidível [Jia and Harman 2011].

Neste contexto, a presente pesquisa propõe o Algoritmo Genético Coevolucionário com Classificação Genética¹ Controlada (AGC_CGC) para seleção automatizada de bons subconjuntos de casos de teste e mutantes para o TM.

As principais contribuições deste artigo consiste em empregar: a) a evolução da população de mutantes seguindo a evolução proposta por Adamopoulos, Harman e Hierons (2004); b) a evolução da população de casos de teste seguindo a evolução proposta por Oliveira, Camilo-Junior e Vincenzi (2013), porém controlando a Classificação Genética e; c) adaptar a coevolução das duas populações (população de casos de teste e programas mutantes) unindo as abordagens de Adamopoulos, Harman e Hierons (2004) e Oliveira, Camilo-Junior e Vincenzi (2013). O avanço que considerado mais significativo pela coevolução empregada pelo AGC_CGC, foi a possibilidade de selecionar um subconjunto de casos de teste com alto escore de mutação sem aumentar drasticamente o tempo de execução do algoritmo.

Este artigo está estruturado da seguinte forma: a Seção 2 apresenta um revisão sobre o TM, bem como traz uma análise sobre pesquisas que aplicam a coevolução no TM. A Seção 3 apresenta a abordagem proposta. Na Seção 4 a experimentação é descrita. Por fim, a Seção 5 realiza as conclusões e os trabalhos futuros.

2. Revisão

2.1. Análise de Mutantes

A Análise de Mutantes (*Mutation Testing* ou *Program Mutation*), nome dado ao Teste de Mutação (TM) no nível de unidade, surgiu na década de 1970 na *Yale University* e *Georgia Institute of Technology*. Um dos primeiros artigos publicados sobre o Teste de Mutação foi o de DeMillo, Lipton, e Sayward (1978) apresentando a ideia da técnica que está fundamentada na hipótese do programador competente e no efeito de acoplamento. A hipótese do programador competente assume que programadores experientes escrevem programas muitos próximos de estarem corretos. Assumindo a validade dessa hipótese, pode-se dizer que os defeitos são introduzidos no programa por meio de pequenos desvios sintáticos, que embora não causem erros sintáticos, alteram a semântica do programa. Por sua vez, o efeito de acoplamento assume que defeitos complexos estão relacionados a defeitos simples. Deste modo, a detecção de um defeito simples pode levar a descoberta de defeitos complexos.

A partir de um programa original P um conjunto P' de programas modificados (mutantes) por *operadores de mutação* inserem pequenos desvios sintáticos no programa (source code or byte code) com o objetivo de avaliar o quanto um conjunto de teste T é adequado [DeMillo, Lipton, e Sayward 1978]. O caso de teste executa sobre o programa original e o mutante, caso as saídas sejam diferentes o mutante é dito estar *morto*. Um problema enfrentado no Teste de Mutação é a geração de uma grande quantidade de programas mutantes traduzindo-se em um alto custo computacional para sua realização. Outro problema, consiste na geração de mutantes equivalentes que são sintaticamente diferentes do programa original, porém ambos são semanticamente

¹ A Classificação Genética faz parte da abordagem de Oliveira, Camilo-Junior e Vincenzi (2013). A proposta do AGC_CGC consiste em favorecer a Classificação Genética, a medida em que se aumenta as gerações, para amenizar custos em termos de tempo de execução.

iguais, não sendo morto por qualquer caso de teste. Dessa forma, verificar a equivalência entre dois programas exige esforço humano por ser um problema indecidível [Jia and Harman 2011].

A métrica que define a adequabilidade de um conjunto de teste chama-se *escore de mutação*. De acordo com DeMillo, Lipton, e Sayward (1978), o *escore de mutação* é um número real que varia entre 0 e 1, calculado conforme a Equação 1. Quanto maior o *escore de mutação* de um caso de teste, maior a sua capacidade em matar programas mutantes, ou seja, demonstrar que o programa em teste não contém o defeito representado pelo mutante.

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)} \quad (1)$$

Onde:

- $ms(P, T)$: escore de mutação do conjunto de testes T ;
- P : programa original;
- T : total de casos de teste;
- $DM(P, T)$: total de programas mutantes mortos por T ;
- $M(P)$: conjunto de mutantes do programa P .
- $EM(P)$: total de mutantes equivalentes.

2.2. Trabalhos Correlatos

O conceito de Algoritmo Genético (AG) surgiu com artigos de Holland [De Jong 2006]. Inspirado na teoria de Darwin, AG é um algoritmo que simula o processo evolucionário de uma população de indivíduos, baseado em parâmetros e operadores genéticos em busca de uma solução ótima ou próxima da ótima. No contexto da Engenharia de Software, a utilização dos AGs recebe um destaque importante para resolução de problemas de Teste de Software.

Já a coevolução, é uma evolução complementar de espécies associadas. Podemos visualizar a coevolução na natureza, por exemplo, entre plantas e insetos em duas partes: (1) para sobreviver a planta necessita evoluir um mecanismo para se defender dos insetos, e (2) os insetos precisam das plantas como fonte de alimento [Engelbrecht 2007].

Nesse processo coevolucionário, as plantas e os insetos, em cada geração, melhoraram suas regras ofensivas e defensivas. Tal interação entre as espécies, faz com que as próximas gerações recebam informações genéticas das gerações anteriores. Este ambiente coevolucionário pode ser incorporados às metaheurísticas evolucionárias para otimizar problemas reais.

Análise das Abordagens que aplicam a Coevolução no Teste de Mutação (TM)

A coevolução foi proposta pela primeira vez no TM por Adamopoulos, Harman e Hierons (2004). A abordagem visa a seleção de: a) um subconjunto de casos de teste com alto escore de mutação e b) um subconjunto de mutantes, que não sejam equivalentes, com alta capacidade de evitar serem mortos. Na abordagem, coevolui-se duas populações: uma população de casos de teste e uma população de mutantes.

Pode-se pontuar dois aspectos positivos da abordagem de Adamopoulos, Harman e Hierons (2004) acerca da função de aptidão para a população de mutantes: 1) a alta capacidade de selecionar mutantes difíceis de serem mortos, em outras palavras, “hard-

to-kill" [Offutt e J. Huffman Hayes 1996]; 2) evita mutantes equivalentes. A função de aptidão de Adamopoulos, Harman e Hierons (2004) para a população de mutantes é descrita pela Equação 2.

$$Mf = \begin{cases} \frac{\sum_{i=1}^L S_i}{L} & \text{if } \forall i. S_i \neq 1. \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Adamopoulos, Harman e Hierons (2004) utilizam o conceito de escore de mutação, só que no sentido inverso, para adequar o conceito de escore à aptidão de um mutante. Em outras palavras, "escore de mutação" quando se trata de um mutante, é sua capacidade em evitar ser morto pelos casos de teste. A Equação 2 descreve Mf que se constitui na aptidão de um indivíduo da população de mutantes (subconjunto de mutantes). S_i é o escore mutação do mutante i . L é o tamanho do indivíduo da população de mutantes.

Portanto, um indivíduo é constituído de L mutantes. Logo, Mf é a razão do somatório dos escores dos mutantes do indivíduo pelo seu tamanho, ou seja, a média dos escores. Acerca da penalização, a Equação 2 verifica se há algum mutante i com $S_i = 1$. Se houver, significa que o mutante não morreu com nenhum dos casos de teste que o executou. Para o indivíduo que contém este tipo de mutante no seu código genético, atribui-se zero (0) à aptidão Mf desse indivíduo, uma vez que esse mutante pode ser um possível equivalente.

A função de aptidão Mf da Equação 2 é a média dos "escores de mutação" dos mutantes que constituem o indivíduo. Tal característica pode guiar a evolução do AG para selecionar os mutantes do tipo "*hard-to-kill*", o que é interessante, pois esses tipos de mutantes podem conduzir a casos de testes fortes [Offutt e J. Huffman Hayes 1996]. Por isso, o AGC_CGC utiliza a aptidão descrita na Equação 3 para a população de mutantes.

Entretanto, pode-se pontuar um aspecto negativo na abordagem de Adamopoulos, Harman e Hierons (2004): a função de aptidão para os indivíduos da população de casos de teste pode não conduzir a um subconjunto de casos de teste com alto escore de mutação. A Equação 3 descreve a função de aptidão para um indivíduo da população de casos de teste.

$$Tf = \frac{\sum_{i=1}^L MS_i}{L} \quad (3)$$

Tf consiste na aptidão de um indivíduo da população de casos de teste. MS_i é o escore de mutação do caso de teste i e L é o tamanho do subconjunto de casos de teste (indivíduo). Logo, Tf é igual a razão do somatório dos escores de mutação dos casos de teste indivíduo pelo seu tamanho, ou seja, a média dos escores.

Tf é a média dos escores de mutação dos casos de teste que constituem o indivíduo. Na busca por um subconjunto forte, isso é um problema, uma vez que a evolução é conduzida na seleção de casos de teste que contenham alto escore individual, mas não necessariamente que estes casos de teste terão um alto escore em conjunto. No trabalho de Oliveira, Camilo-Junior e Vincenzi (2013), a abordagem de Adamopoulos, Harman e Hierons (2004) foi aplicada e a Equação 3 não guiou à seleção de

subconjuntos com alto escore de mutação, comparada com as outras abordagens.

O trabalho de Oliveira, Camilo-Junior e Vincenzi (2013) tem o mesmo objetivo do trabalho de Adamopoulos, Harman e Hierons (2004) e também coevolui-se duas populações em um AG: uma população de casos de teste e uma população de mutantes, e os indivíduos também são representados por subconjuntos. A diferença está na escolha das melhores soluções baseando-se no conceito chamado Efetividade Genética. Na Efetividade Genética procura-se por genes efetivos, ou seja, aqueles genes que contribuem no aumento da aptidão do indivíduo. Para se empregar o conceito de Efetividade Genética Oliveira, Camilo-Junior e Vincenzi (2013) propõem a realização da Classificação Genética. A Figura 1 ilustra um indivíduo da população de casos de teste com os seus genes classificados.



Figura 1. Indivíduo da população de casos de teste com os genes classificados.

Na Figura 1, o único gene efetivo é o caso de teste 91 que está classificado como *EFG* (*Effective Gene*), ou seja, ele contribui para o escore do subconjunto. Os casos de teste 129 e 856 são do tipo *REG* (*Redundant Gene*), pois não contribuem para o escore do subconjunto, de forma que os mutantes mortos por casos de teste do tipo *REG* só matam mutantes já mortos pelos casos de teste do tipo *EFG*. Por fim, o caso de teste 07 é do tipo *NEG* (*Non Effective Gene*), ou seja, não matam nenhum mutante e não contribuem para o escore do subconjunto de casos de teste.

O aspecto positivo da Classificação Genética é que ela informa para o operador de cruzamento e mutação (propostos também por Oliveira, Camilo-Junior e Vincenzi (2013)) quais são os casos de teste que realmente interessam para formar um suconjunto de casos de teste com alto escore de mutação. No trabalho de Oliveira, Camilo-Junior e Vincenzi (2013) constatou-se que a Efetividade Genética foi eficaz no alcance de subconjuntos de casos teste com alto escore de mutação. Por isso, o AGC_CGC emprega a Classificação Genética de forma controlada, para a população de casos de teste, visando a obtenção de subconjuntos de casos de teste com alto escore de mutação.

A Classificação Genética é empregada na abordagem de Oliveira, Camilo-Junior e Vincenzi (2013) em dois momentos. No primeiro momento, é aplicada antes da realização do cálculo da função de aptidão. Em um segundo momento, é aplicada no operador de cruzamento *ES* (*Effective Son*). Este operador utiliza dois indivíduos (um pai e uma mãe) e realiza uma Classificação Genética nos genes desses indivíduos, possibilitando a formação de um filho com maior quantidade de genes efetivos, ou seja, com maior aptidão.

Por outro lado, o aspecto negativo é que a realização da Classificação Genética, embora reduza o custo relacionado à quantidade de testes realizados² quando comparado ao teste exaustivo, resulta em um tempo de execução maior do que o método proposto por Adamopoulos, Harman and Hierons (2004).

O AGC_CGC proposto no presente artigo reúne as características positivas das duas abordagens citadas acima. A Seção 3 fornece uma explicação mais detalhada sobre

² A quantidade de testes realizados refere-se a quantidade de vezes que se realiza os passos 2 e 3 para aplicação do teste de mutação. O passo 2 consiste na execução do programa em teste. O passo 3 consiste na execução do programa mutante, verificando se o erro foi identificado ou não.

como o AGC_CGC integra as evoluções propostas por Adamopoulos, Harman e Hierons (2004) e Oliveira, Camilo-Junior e Vincenzi (2013) em um só algoritmo.

3. O Algoritmo Genético Coevolucionário com Classificação Genética Controlada (AGC_CGC)

O AGC_CGC emprega a coevolução considerando: uma população de casos de teste e uma população de programas mutantes. A Figura 2 ilustra o ambiente coevolucionário empregado pelo AGC_CGC que pode ser descrito pelos seguintes passos:

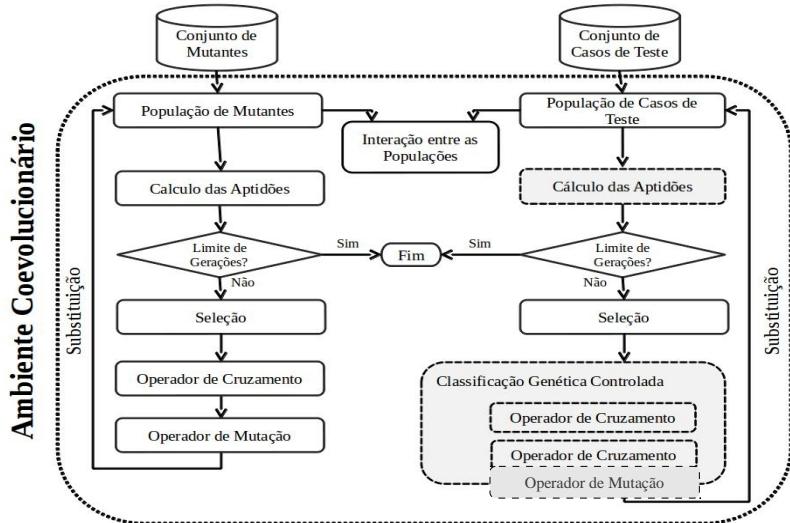


Figura 2. Ambiente Coevolucionário empregado pelo AGC_CGC.

1. **Inicialização das populações:** as populações de casos de teste e mutantes são iniciadas aleatoriamente, cada população tem um tamanho fixo para formar os indivíduos, conforme estabelecido nos parâmetros;
2. **Calculo das aptidões:**
 - a) **População de mutantes:** os indivíduos da população de casos de testes executam o teste sobre os indivíduos da população de programas mutantes. A partir dessa interação, calcula-se o aptidão dos indivíduos da população de mutantes, conforme a Equação 2;
 - b) **População de casos de teste:** após os indivíduos da população de casos de testes terem executados os testes sobre os indivíduos da população de programas mutantes, ocorre a escolha da função de aptidão com base na quantidade de gerações:

Seja G_c o número atual de gerações. Seja G_t a quantidade total de gerações (definida nos parâmetros do AG). O cálculo da aptidão dos indivíduos da população de casos de teste, obedece a seguinte regra:

I) caso $G_c \leq G_t/2$, calcula-se a aptidão conforme Equação 3. A intenção consiste favorecer a aptidão dos indivíduos com casos de teste fortes individualmente.

II) caso $G_c > G_t/2$, calcula-se a aptidão conforme Equação 1, mas sem subtrair a quantidade equivalentes da quantidade de mutantes mortos, pois o AG não sabe o valor de equivalentes. A intenção consiste em favorecer a

aptidão dos indivíduos fortes em conjunto.

3. **Condição de parada do AG:** a quantidade de gerações é a condição de parada do AG. Este número é fornecido como um parâmetro de entrada do AG. Quando o AG alcança a quantidade de gerações definida, o AG pára a execução. O melhor indivíduo da população de casos de teste é fornecido como saída (subconjunto com melhor escore de mutação) e o melhor indivíduo da população de mutantes (subconjunto de mutantes mais difícil de ser morto).
4. **Operador de seleção:** seleciona os pais para o cruzamento utilizando o operador de seleção *Torneio*.
5. **Operador de Cruzamento:**
 - a) População de mutantes: o operador de cruzamento utilizado para os indivíduos dessa população é a *máscara uniforme*³.
 - b) População de casos de teste: aplica-se a Classificação Genética Controlada.
6. **Operador de Mutação:**
 - a) População de mutantes: o operador de mutação⁴ utilizado para os indivíduos dessa população é a *mutação uniforme*.
 - b) População de casos de teste: caso se tenha utilizado o operador de cruzamento *ES (Effective Son)*, utiliza-se o operador de mutação *MG (Muta Gene)*⁵. Caso contrário, aplica-se a mutação uniforme.
7. **Substituição:** os filhos substitui a população corrente, mantendo o melhor indivíduo da população de casos de teste e os melhores indivíduos das populações.

3.1. A Classificação Genética Controlada

A Classificação Genética Controlada é aplicada somente na população de casos de teste, especificamente, no operador de cruzamento.

O controle para aplicação da Classificação Genética baseia-se na Taxa de Classificação Genética (*TCG*), considerando o Percentual Máximo de Classificação (*PMC*) que, por sua vez, é estabelecido como parâmetro do AGC_CGC. A *TCG* é calculada conforme Equação 4.

$$TCG = (Gc * PMC) / Gt \quad (4)$$

Em cada geração corrente (*Gc*), baseando-se na quantidade de gerações (*Gt*) e no *PMC*, a *TCG* é calculada. Considerando *Gt* = 500 e *PMC* = 0.30, na primeira geração (*Gc* = 1) *TCG* = 0.0006. Com geração corrente de número 200 (*Gc* = 200) *TCG* = 0.12 e, por exemplo, na última geração, ou seja, na geração de número 500 (*Gc* = 500) *TCG* = 0.30. Vale observar que a medida que a geração corrente aumenta, o valor de *TCG* também aumenta, sendo sempre *TCG* = *PMC* na última geração.

O valor *TCG* define qual o operador de cruzamento será utilizado. O controle da

³ Uma máscara binária é gerada de maneira aleatória. Dois filhos são construídos pela troca de genes do pai e da mãe.

⁴ Para cada gene do indivíduo, gera-se um número aleatório. Caso esse número seja menor ou igual a *taxa de mutação* (estabelecida nos parâmetros) o gene recebe mutação.

⁵ O operador de mutação *MG (Muta Gene)* é proposto na aborda de Oliveira, Camilo-Junior e Vincenzi (2013). Este operador atribui mutação de 100% aos genes que não sejam do tipo *EFG*.

Classificação Genética funciona da seguinte forma:

1. Gera-se, aleatoriamente, o número real AL com valor entre 0 e 1;
2. Calcula-se o valor atual para TCG ;
3. Caso $TCG \leq AL$, aplica-se o operador de cruzamento ES . Caso contrário, aplica-se o operador de máscara uniforme.

A característica da TCG aumentar conforme se aumenta o número de gerações minimiza o número de aplicações do operador de cruzamento ES . Além disso, objetiva aumentar a probabilidade de aplicação desse operador nas últimas gerações a fim de que a Classificação Genética seja empregada em informações evoluídas.

4. Experimentação

4.1. Algoritmos e Benchmarks

A experimentação compreende a comparação do algoritmo proposto (AGC_CGC) com outros três algoritmos:

1. A1: algoritmo proposto em Adamopoulos, Harman e Hierons (2004);
2. AGC: algoritmo proposto em Oliveira, Camilo-Junior e Vincenzi (2013);
3. AL: consiste no algoritmo de abordagem aleatória. O algoritmo pode ser descrito em 4 passos: 1) gera-se um subconjunto de casos de teste e um subconjunto de mutantes aleatoriamente na quantidade de gerações utilizadas nos outros algoritmos; 2) calcula-se as aptidões dos subconjuntos conforme Equação 1; 3) seleciona-se o melhor subconjunto de mutantes e o melhor subconjunto de casos de teste de todas as gerações.

São utilizados dois benchmarks reais na experimentação (Tabela 1). Para geração dos programas mutantes foi utilizada a ferramenta Proteum [Delamaro 1996] que aplica o TM em dois programas escritos na linguagem C:

1. *Programa cal*: utilitário linux em que o usuário fornece como entrada o ano e/ou o mês, por linha de comando, e o programa retorna: a) o calendário do ano, quando se fornece apenas um parâmetro e; b) o calendário de um mês referente a um ano específico, quando se fornece dois parâmetros.
2. *Programa comm*: utilitário linux em que o usuário fornece como parâmetro dois arquivos de entrada, por linha de comando, e o programa compara linha por linha entre os dois arquivos.

Tabela 1. Informações dos benchmarks utilizados

Nome	Total Mutantes	Total Equivalentes	Total Casos de Teste
<i>cal</i>	4622	344	2000
<i>comm</i>	1869	222	801

4.2 Parâmetros dos algoritmos

Para todos os algoritmos foram fixados os seguintes parâmetros: a) operador de seleção: torneio, com dois competidores; b) taxa de cruzamento: 100%; c) taxa de mutação: 5%; d) elitismo: 1 indivíduo; e) tamanho do indivíduo da população de casos de teste: 14 e; f) tamanho do indivíduo da população de programas mutantes: 14.

4.3 Discussão dos Resultados

Os resultados da execução dos algoritmos para cada benchmark estão expressos nas Tabelas 2 e 3. Todos os resultados das Tabelas 2 e 3 consistem numa média de 10 execuções dos parâmetros. A fim de ajudar na análise dos subconjuntos de casos de teste e programas mutantes selecionados pelos AGs, todo o conjunto de mutantes e casos de testes, dos benchmarks utilizados, foram avaliados. Os 20% dos mutantes mais difíceis de serem mortos, foram classificados como mutantes “*Hards*”, seguindo a ideia de Offutt e Hayes (1996). Analogamente, os 20% dos casos de teste com maior capacidade em matar mutantes foram classificados como “*Fortes*”. Com essa informação, é possível avaliar, para cada e em cada benchmark, se as seleções dos subconjuntos fazem parte desses grupos.

Tabela 2. Benchmark cal

Algoritmo	Escore Máximo		Média		%Hards	%Fortes	Quant. Equivs	Tempo de Execução
	IndTC	IndMT	IndTC	IndMT	IndMT	IndTC		
AGC_CGC	0.9862	1.0	0.9577	0.9995	0.8749	0.4285	0	35.9 s
AGC	0.9943	1.0	0.9907	1.0	0.5	0.4390	0	501.3 s
A1	0.6905	1.0	0.6827	1.0	0.8214	1.0	0	18.1 s
AL	0.8475	1.0	0.8410	1.0	0.12	0.48	1.2	2 s

Tabela 3. Benchmark comm

Algoritmo	Escore Máximo		Média		%Hards	%Fortes	Quant. Equivs	Tempo de Execução
	IndTC	IndMT	IndTC	IndMT	IndMT	IndTC		
AGC_CGC	0.9914	1.0	0.9837	1.0	1.0	0.4357	0	23 s
AGC	0.9890	1.0	0.9803	1.0	0.5277	0.4642	0	191 s
A1	0.8476	1.0	0.8392	1.0	1.0	1.0	0	9 s
AL	0.8904	1.0	0.8804	1.0	0.19	0.50	1.3	1 s

Considerando o programa *comm*, o AGC_CGC teve um melhor desempenho na seleção de um subconjunto de casos de teste (IndTC). Já considerando o programa *cal*, o AGC teve o melhor desempenho, nesse quesito. O pior desempenho, foi o do algoritmo A1 que perdeu inclusive para o algoritmo com abordagem aleatória.

Vale observar a coluna %Fortes em que o algoritmo A1 teve o pior desempenho na seleção dos subconjuntos de casos de com maior escore, porém com maior percentual na seleção de casos de testes fortes. Esse resultado traz uma evidência de que casos de teste fortes individualmente não significa que são fortes em conjunto.

Todos os algoritmos que empregaram a penalização, evitaram os mutantes equivalentes. A única exceção foi o algoritmo de abordagem aleatória AL que selecionou uma média de 1.2 e 1.3 mutantes equivalentes para os benchmarks *cal* e *comm*, respectivamente.

O algoritmo AL foi o que teve um menor tempo de execução, uma vez que ele é uma abordagem aleatória e não tem o mecanismo de um AG, como os demais algoritmos. O segundo algoritmo mais eficiente nesse quesito, foi o algoritmo A1. O terceiro mais eficiente foi o algoritmo AGC_CGC e o quarto foi o AGC.

5. Conclusões e Trabalhos Futuros

Neste artigo foi proposto um novo algoritmo o AGC_CGC para seleção de um bom subconjunto de casos de teste e um bom subconjunto de programas mutantes no contexto do TM.

Duas abordagens existentes foram estudadas e o AGC_CGC foi concebido levando em consideração os pontos fortes e fracos dos algoritmos que propuseram a coevolução no TM. A experimentação revelou um bom desempenho do AGC_CGC que reduziu o tempo de execução, quando comparado ao algoritmo AGC, e melhorou o escore de mutação quando comparado ao algoritmo A1. Tal redução, deve-se ao controle realizado na Classificação Genética empregada pelo AGC_CGC.

Diante disso, acredita-se que o AGC_CGC constitui-se numa abordagem promissora para seleção de casos de teste e programas mutantes para o TM.

Como trabalhos futuros, pretende-se aplicar o AGC_CGC em número maior de benchmarks com uma variação maior dos parâmetros utilizados para extrair maiores conclusões da abordagem proposta.

Referências

- A. Oliveira, C. Camilo-Junior and Vincenzi (2013) “A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing”. In: IEEE Congress on Evolutionary Computation (CEC).
- A. Jefferson Offutt and J. Huffman Hayes (1996) “A semantic model of program faults”. In: International Symposium on Software Testing and Analysis (ISSTA '96) p. 195-200.
- A. P. Engelbrecht (2007) “Computational intelligence: an introduction”. In: John Wiley & Sons.
- K. Adamopoulos, M. Harman, and R. M. Hierons (2004) “How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution”. In: Genetic and Evolutionary Computation Conference (GECCO), p. 1338-1349.
- K. A. De Jong (2006) “Evolutionary Computation: a unified approach”, MIT Press.
- M. Papadakis, N. Malevris, and M. Kallia (2010) “Towards automating the Generation of Mutation Tests”. Workshop on Automation of Software Test (AST '10) p. 111-118
- M. E. Delamaro (1996) “Proteum - A Tool for the Assessment of Test Adequacy for C ”. In: Proceedings of the Conference on Performability in Computing Systems (PCS), p. 79-95.
- P. McMinn (2011) “Search-Based Software Testing: past, present and future. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW) p. 153-163
- R. A. DeMillo, R. J. Lipton, and F. G. Sayward (1978) “Hints on test data selection: help for the practicing programmer”. In: IEEE Computer Society Press Los Alamitos, p. 34-41 .
- Y. Jia, and M. Harman (2011) “An analysis and survey of the development of mutation testing”. In: IEEE Transactions on Software Engineering, p. 649-678.

Seleção de produto baseada em algoritmos multiobjetivos para o teste de mutação de variabilidades

Édipo Luis Féderle¹, Giovani Guizzo¹, Thelma Elita Colanzi^{1,2},
Silvia Regina Vergilio¹, Eduardo J. Spinosa¹

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

²DIN - Universidade Estadual de Maringá (UEM) – Maringá, PR – Brasil

{elfederle, gguizzo, thelmae, silvia, spinosa}@inf.ufpr.br

Resumo. *O teste baseado em mutação de variabilidades é utilizado para selecionar produtos para o teste de uma Linha de Produto de Software (LPS). Visto que na maioria das vezes testar todos os produtos da LPS não é factível, um subconjunto de produtos é selecionado considerando possíveis defeitos que podem estar presentes em um diagrama de características. Além disso, o escore de mutação pode ser utilizado para avaliar a qualidade de um conjunto de produtos. Obter um subconjunto com alto escore de mutação e um número pequeno de casos de teste (produtos) é então um problema de otimização multiobjetivo, para o qual diferentes boas soluções são possíveis. Considerando este fato, este trabalho apresenta resultados da aplicação de dois Algoritmos Evolutivos Multiobjetivos (MOEAs), NSGA-II e SPEA2, para a seleção de casos de teste baseada em mutação. Os resultados mostram uma redução de aproximadamente 99% na quantidade de casos de teste para um dos problemas.*

Abstract. *The variability mutation testing is used to select products in the test of Software Product Lines (SPLs). Since in most cases the testing of all SPL products is infeasible, a subset of products is selected considering possible faults that can be present in feature diagrams. In addition to this, the mutation score is used to the quality evaluation of a set of products. To obtain a subset with high mutation score and a reduced number of test cases (products) is then a multi-objective optimization problem, for that different good solutions are possible. Considering this fact, this work presents results from the application of two Multiobjective Evolutionary Algorithms (MOEAs), NSGA-II and SPEA2, for mutation-based selection of test cases. The results show a reduction around 99% of the amount of test cases to one of the problems.*

1. Introdução

Uma Linha de Produto de Software (LPS) pode ser definida como um conjunto de produtos que compartilham características comuns. Essas características satisfazem necessidades específicas de um segmento de mercado em particular e são desenvolvidas a partir de um conjunto comum de artefatos [Bergey et al. 2010, van der Linden et al. 2007].

A engenharia de LPS está auxiliando empresas a criarem software minimizando seu custo e maximizando sua qualidade [van der Linden et al. 2007]. Esta adoção crescente de LPS na indústria tem criado uma demanda por técnicas de teste específicas para

LPS. Uma atividade importante é verificar se uma LPS descreve os produtos de software como desejado, e se as variabilidades e partes comuns da LPS estão especificadas corretamente. Um produto é uma derivação customizada das características da LPS, geralmente representadas no diagrama de características e pode ser entendido como um caso de teste. Idealmente, todos os produtos possíveis gerados a partir de uma LPS deveriam ser testados, entretanto, dada a complexidade das aplicações, o número de produtos possíveis é exponencial e o teste exaustivo é impraticável [Cohen et al. 2006]. Por isto, apenas um subconjunto representativo desses produtos deve ser selecionado.

Na literatura podem ser encontrados alguns critérios para auxiliar a seleção de produtos. O trabalho de Ferreira et al. (2013) propõe uma abordagem de teste baseada em mutação de variabilidades do modelo de características. Transformações sintáticas são introduzidas para gerar diagramas mutantes. Cada transformação descreve um possível defeito que pode estar presente em um diagrama. Um mutante é considerado morto se ele valida um produto (caso de teste) de maneira diferente da maneira avaliada pelo diagrama original em teste. Caso contrário ele é considerado vivo.

Os mutantes podem ser utilizados para a seleção de casos de teste, ou seja, de produtos das LPS a serem testadas. A ideia é selecionar, dentre todos os possíveis produtos de uma LPS, um conjunto que satisfaça algumas restrições como por exemplo, matar a maior quantidade de mutantes, associada ao escore de mutação, e usar a menor quantidade possível de produtos.

Portanto a seleção de produtos para o teste de mutação de variabilidades em uma LPS é um problema de otimização multiobjetivo, o qual é o foco do presente trabalho, que descreve uma solução implementada com dois Algoritmos Evolutivos Multiobjetivos (MOEAs) bastante utilizados em SBSE (*Search-Based Software Engineering*): NSGA-II (*Non Sorting Genetic Algorithm II*) [Deb et al. 2002] e SPEA2 (*Improved Strength Pareto Evolutionary Algorithm*) [Zitzler et al. 2001]. Esses algoritmos são utilizados para encontrar, dentre os possíveis produtos (casos de teste) da LPS, um determinado conjunto com máximo escore de mutação e um número mínimo de produtos. O trabalho apresenta resultados do uso destes dois algoritmos em dois casos reais que mostram que ambos resolvem eficientemente o problema, apresentando soluções úteis para o testador.

Este trabalho está organizado da seguinte maneira: as Seções 2 e 3 fazem uma revisão, respectivamente, do teste de mutação de variabilidades em LPS e sobre otimização multiobjetivo. A Seção 4 descreve a representação adotada para o problema, operadores utilizados e funções objetivos. A Seção 5 apresenta como foi conduzido o experimento. A Seção 6 apresenta os resultados das avaliações, comparando os algoritmos e avaliando as soluções. A Seção 7 conclui o artigo e apresenta trabalhos futuros.

2. Teste de Mutação de Variabilidades

No contexto de LPS, um modelo de características descreve as características funcionais e de qualidade de um domínio. Com essa modelagem, as variabilidades, pontos de variação e variantes da LPS podem ser graficamente representados em forma de árvore, no diagrama de características, provendo assim uma clareza na definição de variabilidades e evitando más interpretações por parte dos interessados [van der Linden et al. 2007].

A Figura 1(a) apresenta um exemplo de um diagrama de características para a LPS “Car Audio System” (CAS) [Weissleder et al. 2008]. A LPS, para sistemas de áudio

de carros, tem funcionalidades obrigatórias (estão presentes em todos os produtos) como controle de volume, controle de frequência de rádio e uso de *playback*, e funcionalidades opcionais como suporte para múltiplos formatos de áudio e sistema de navegação.

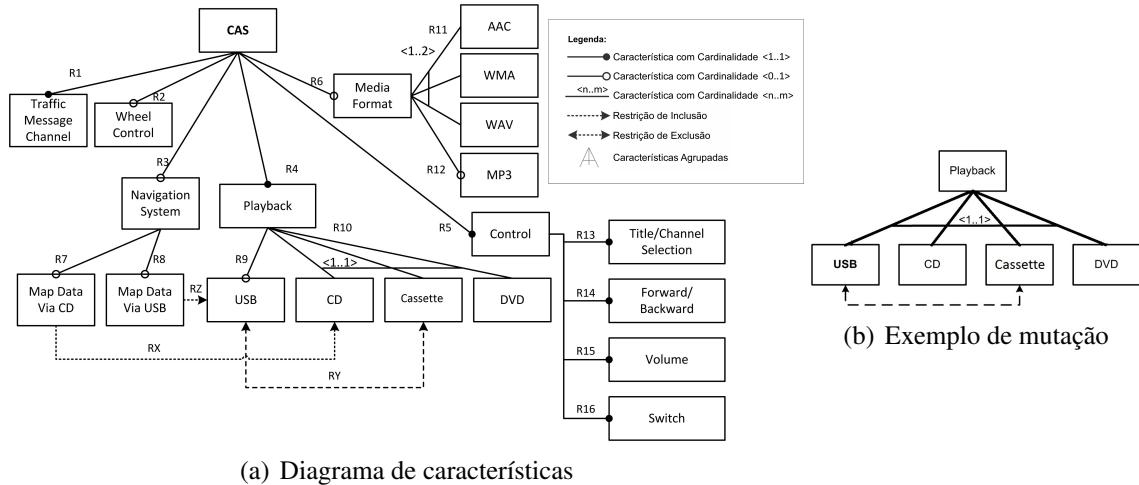


Figura 1. Exemplo da LPS CAS [Ferreira et al. 2013]

No exemplo da Figura 1(a), a característica raiz é o elemento CAS. Este elemento é composto por um conjunto de restrições (inclusão e exclusão, RX e RY respectivamente) e por associações, que podem ser: i) associações binárias, que por sua vez podem ser obrigatórias (R1), opcionais (R2) ou baseadas em cardinalidade; ou ii) conjunto de associações (R10 e R11 por exemplo). Uma característica pode ser de três diferentes tipos e contém no mínimo uma associação. Um conjunto de associações é composto por pelo menos duas características agrupadas ("CD", "Cassette" e "DVD" em R10 para a característica "Playback"). Uma associação binária é composta por apenas uma característica, que por sua vez possui uma cardinalidade. Por exemplo a característica "Wheel Control" possui cardinalidade [0..1], enquanto que "Playback" possui cardinalidade [1..1].

Para realizar o teste de mutação, Ferreira et al. (2013) introduziram um conjunto de operadores de mutação que realizam mudanças sintáticas no diagrama original, como por exemplo, mudança nos tipos de restrições e associações presentes no diagrama, mudanças nas cardinalidades, etc. Um exemplo de mutante pode ser visto na Figura 1(b). Neste exemplo, apenas a parte modificada da figura é apresentada. Pode-se observar que a característica "USB" foi adicionada ao grupo de características agrupadas existente.

O diagrama de características é utilizado para instanciar produtos, e estes são utilizados como casos de teste para matar os mutantes gerados. Um diagrama mutante é dito morto se ele valida um produto inválido para o diagrama original, ou o contrário ele não valida um produto válido para o diagrama original. Por exemplo, um produto que inclui as características "USB" e "CD", é capaz de matar o mutante exemplo. Ele é válido para o diagrama original, mas é inválido de acordo com o mutante que não permite ambas características em um mesmo produto. O objetivo é matar todos os mutantes. O escore de mutação, dado pela razão entre o número de mutantes mortos pelo número de mutantes gerados, avalia a qualidade de um conjunto de teste (de produtos).

3. Otimização Multiobjetivo

Muitos problemas da vida real estão associados a fatores (objetivos) conflitantes. Para estes problemas, diversas boas soluções existem, chamadas não-dominadas. Se todos os objetivos de um problema forem de minimização, uma solução x domina outra solução y ($x \prec y$), se e somente se: $\forall z \in Z : z(x) \leq z(y)$ e $\exists z \in Z : z(x) < z(y)$. Uma solução Pareto ótima não pode ser melhorada em um objetivo sem que haja prejuízo em pelo menos um outro objetivo. Os valores das funções objetivo das soluções do conjunto Pareto ótimo são chamados de PF_{true} (Fronteira de Pareto real) [Coello et al. 2007]. Os algoritmos geralmente obtêm apenas uma aproximação desta fronteira, chamada PF_{approx} (Fronteira de Pareto aproximada). Algoritmos evolutivos se encaixam perfeitamente neste contexto, pois problemas multiobjetivos são baseados em uma população de soluções.

Apesar de existirem inúmeros algoritmos evolutivos multiobjetivos na literatura, algoritmos como o NSGA-II e o SPEA2 são os mais conhecidos, utilizados e eficientes, além de possuírem diferentes estratégias de evolução e diversificação [Coello et al. 2007]. Geralmente os algoritmos diferem: i) no procedimento de atribuição de *fitness*; ii) na abordagem de diversificação; e iii) no elitismo.

O algoritmo NSGA-II [Deb et al. 2002] ordena em cada geração os indivíduos das populações pai e filha baseando-se na relação de não-dominância. Diversas fronteiras são criadas, e depois da ordenação, as piores soluções (que dominam menos soluções) são descartadas. O NSGA-II utiliza essas fronteiras em sua estratégia de elitismo. Além disso, de modo a garantir a diversidade das soluções, o NSGA-II utiliza o procedimento *crowding distance* para ordenar os indivíduos de acordo com a sua distância em relação aos vizinhos da fronteira. Ambas ordenações (de fronteiras e de *crowding distance*) são usadas pelo operador de seleção.

Além de ter sua população principal, o SPEA2 [Zitzler et al. 2001] possui uma população auxiliar de soluções não-dominadas encontradas no processo evolutivo. Essa população auxiliar fica armazenada em um arquivo externo, sendo que para cada solução deste arquivo e da população principal é calculado um valor de *strength* (usado no cálculo do *fitness* do indivíduo). O valor de *strength* de uma solução corresponde a quantidade de indivíduos (do arquivo e da população principal) dominados por esta solução. Este valor é utilizado durante o processo de seleção. No processo evolutivo, a população auxiliar é utilizada para complementar a nova geração com soluções não-dominadas. Se o tamanho máximo da população auxiliar for excedido, um algoritmo de agrupamento é utilizado com o intuito de remover as piores soluções do arquivo.

4. Representação do Problema

Os dados de entrada do algoritmo consistem em uma matriz binária de m linhas e n colunas (Tabela 1). Os produtos são representados nas colunas e os mutantes nas linhas. Na tabela podem ser vistos quais produtos são capazes de matar quais mutantes. Quando um mutante i é morto por um produto j , o valor da célula i, j é preenchido com *true*, caso contrário é preenchido com *false*. Por exemplo, os conjuntos de produtos $\{1, 3\}; \{1, 4\}; \{2, 3, 5\}; \{2, 4, 5\};$ e $\{1, 2, 3, 4, 5\}$ são capazes de matar todos os mutantes.

Um conjunto de casos de teste X_p pode ser visto como o conjunto de variáveis de decisão de uma solução p (cromossomo) da população P de tamanho N , e cada produto

Tabela 1. Exemplo de entrada de dados

Mutantes		Produtos				
		1	2	3	4	5
1	false	false	true	true	false	
2	true	false	false	false	true	
3	true	true	false	false	false	
4	false	false	true	true	false	

do conjunto de casos teste X_p representa um gene x da solução p . De fato, um cromosoma p é uma matriz binária parcial ou igual a matriz original, sendo geralmente formado por uma quantidade reduzida de colunas (produtos).

Neste contexto existem duas funções objetivo (*fitness*): i) minimizar a quantidade de produtos no conjunto de casos de teste (função objetivo = $|X|$); e ii) minimizar a quantidade de mutantes que foram deixados vivos pelo conjunto de casos de teste (função objetivo = $|X| - |mutantesMortosPor(X)|$). Como esses dois objetivos são conflitantes, ao diminuir a quantidade de produtos de uma solução é muito provável que a quantidade de mutantes vivos aumente.

Foram criados operadores de mutação e recombinação adaptados ao problema em questão. O operador de seleção utilizado para a implementação dos algoritmos foi o disponível no framework jMetal [Durillo and Nebro 2011].

O operador de mutação criado e utilizado neste trabalho é chamado *Product Set Mutation Operator* e é constituído de três processos básicos: i) mutação por inclusão de produto; ii) mutação por troca de produto; e iii) mutação por remoção de produto.

Um ou nenhum destes três processos é utilizado em cada execução do operador. Primeiro o operador dá a preferência de execução para o processo de inclusão de produto. Internamente, o processo utiliza a probabilidade de mutação $\rho_{mutation}$ para decidir se ele vai ser executado ou não. Se este processo não for utilizado, então o operador tenta utilizar o processo de troca de produto. Mais uma vez a probabilidade de mutação é consultada para definir se o processo será utilizado ou não. Caso não seja, o operador tenta utilizar o processo de remoção de produto. Se um dos processos for executado na mutação, então os demais são descartados e o algoritmo continua de onde parou.

O processo de mutação por inclusão de produto adiciona um produto aleatório $x' \in D : x' \notin X_p$ ao final do conjunto de variáveis de decisão X_p da solução p , onde X_p é um conjunto de produtos (casos de teste) no espaço da solução p . A única restrição para a execução deste processo é $|X_p| < n$. O processo de mutação por troca de produto remove um produto $x_i \in X_p$ do conjunto de variáveis de decisão X_p , onde i é um valor aleatório $0 < i \leq |X_p|$, e depois adiciona um produto $x' \in D : x' \notin X_p \wedge x' \neq x_i$ ao final do conjunto de variáveis de decisão X_p da solução p , onde X_p é um conjunto de produtos (caso de teste) no espaço da solução p . A única restrição para a execução deste processo é $|X_p| < n$. O processo de mutação por remoção de produto remove um produto $x_i \in X_p$ do conjunto de variáveis de decisão X_p da solução p , onde i é um valor aleatório $0 < i \leq |X_p|$ e X_p é um conjunto de produtos (caso de teste) no espaço da solução p . A única restrição para a execução deste processo é $|X_p| > 1$.

O operador de recombinação (*crossover*) foi chamado de *Product Set Crossover Operator*. Esse operador recebe duas soluções p_1 e p_2 e, caso passe na probabilidade de recombinação $\rho_{crossover}$, gera duas soluções filhas q_1 e q_2 com vetores de variáveis de decisão de tamanho $\frac{|X_{p1}|}{2} + \frac{|X_{p2}|}{2}$. Os dois novos filhos gerados possuem $\frac{|X_{p1}|}{2}$ genes provenientes do primeiro pai e $\frac{|X_{p2}|}{2}$ provenientes genes do segundo pai. Os genes são todos distintos e escolhidos aleatoriamente nos vetores X de cada pai. Caso um dos pais possua apenas um gene, então esse único gene é passado a ambos os filhos e a seleção dos genes provenientes do outro pai dá-se de forma convencional, conforme mencionado anteriormente. Porém, se ambos os pais possuírem apenas um gene cada e esses genes forem distintos, então os novos filhos terão os dois genes copiados de seus pais. Se os dois pais possuírem apenas um gene cada e estes genes forem iguais para ambos, então os novos filhos serão formados unicamente por este gene.

5. Experimento

Foram utilizados neste experimento duas matrizes de entrada a partir do trabalho de Ferreira et al. [Ferreira et al. 2013]. A primeira matriz (M1) tem 263 mutantes e 43 produtos, e a segunda matriz (M2) 254 mutantes e 916 produtos.

Para medir a qualidade de um resultado, foi utilizado o *hypervolume* [Zitzler and Thiele 1999] como indicador de qualidade. Resumidamente o *hypervolume* mede a área do espaço de busca que é coberta (dominada) por uma fronteira (vetor de soluções). Portanto, quanto maior o *hypervolume* de uma fronteira, melhor a sua qualidade.

Para a seleção de parâmetros dos algoritmos, foram avaliadas 10 configurações diferentes levando em conta o tamanho da população, probabilidade de recombinação e mutação, e no caso do SPEA2, tamanho da população auxiliar. Foram feitas trinta execuções para cada algoritmo (NSGA-II e SPEA2). Após testes empíricos [Arcuri and Fraser 2011], as configurações da Tabela 2 foram constatadas como sendo as mais eficientes e utilizadas na análise dos resultados.

Tabela 2. Configurações adotadas

Parâmetros	NSGA-II	SPEA2
Tamanho da População (N)	100	100
Gerações (g)	20000	20000
Probabilidade de recombinação ($\rho_{crossover}$)	30%	90%
Probabilidade de Mutação ($\rho_{mutation}$)	80%	50%
Tamanho da População Auxiliar (\bar{N})	-	100

Cada execução teve como resultado um conjunto de soluções não-dominadas em uma PF_{approx} . Para cada resultado gerado foi calculado o *hypervolume* com um ponto de referência comum e ao final das trinta execuções de cada algoritmo foram calculados a média e o desvio padrão dos *hypervolumes*. Como para o problema em questão não se sabe a PF_{true} , o ponto de referência foi determinado pelo pior valor das funções objetivos mais um. Por exemplo, para M1 o ponto de referência é (264, 44) (mutantes vivos, quantidade de produtos). Para cada algoritmo foi obtido o melhor *hypervolume* sendo exatamente o que determinou o melhor resultado. Além disso foi calculado o tempo de execução de cada algoritmo.

6. Resultados

A Tabela 3 apresenta alguns dos resultados obtidos que apoiam a comparação entre os dois MOEAs utilizados. Ao final de todas as execuções dos algoritmos, foram obtidos os melhores valores (coluna 4), valores médios (coluna 3) e desvio padrão do *hypervolume* (coluna 5) de cada algoritmo (coluna 2) em cada problema. Além disso, o tempo de execução dos algoritmos é apresentado na sexta coluna.

Tabela 3. Resultados dos cálculos de *hypervolume* e tempo

LPS	Algoritmo	Média	Melhor	Desvio Padrão	Tempo (s)
M1	NSGA-II	0.89582	0.89583	2.6981×10^{-5}	2811
	SPEA2	0.89581	0.89583	3.5974×10^{-5}	6052
M2	NSGA-II	0.99716	0.99718	1.7210×10^{-5}	1695
	SPEA2	0.99715	0.99718	1.9482×10^{-5}	3990

Houve uma diferença significativa no tempo de execução dos algoritmos: nos dois casos o SPEA2 levou mais do que o dobro do tempo do NSGA-II para ser executado. Por outro lado, para o *hypervolume* não foi encontrada uma diferença significativa entre os algoritmos. Para confirmar essa afirmação foi utilizado o teste de Wilcoxon Mann-Whitney [Wilcoxon 1945] com o intuito de apresentar diferenças estatísticas considerando 95% de confiança (*p-value* menor que o nível de significância $\alpha = 0.05$). O teste estatístico não revelou diferenças entre os resultados do NSGA-II e SPEA2 em ambos problemas. Mesmo não havendo diferenças estatísticas, comparando os valores da Tabela 3, é possível verificar que o resultado do NSGA-II foi ligeiramente superior ao resultado do SPEA2 nas médias do *hypervolume* e os melhores resultados foram idênticos. As melhores PF_{approx} encontradas pelos algoritmos são ilustradas nos gráficos das Figuras 2 e 3 para as LPSs das matrizes M1 e M2, respectivamente. Nos dois casos, é possível perceber que justamente por não haver diferença significativa entre os melhores *hypervolumes*, as soluções de cada fronteira estão praticamente sobrepostas.

De modo a exemplificar os resultados da minimização apresentada, as soluções da melhor fronteira encontrada pelo algoritmo NSGA-II para M2, seus respectivos valores objetivo, e a cobertura do teste (*escore*) são apresentados na Tabela 4.

A população final gerada foi de tamanho 100 (parâmetro N), entretanto foram descartadas as soluções repetidas, o que resultou em 30 diferentes soluções não-dominadas. As soluções com a maior cobertura de teste (em negrito) obtiveram um escore de 100% utilizando apenas 10 produtos dentre os 916 disponíveis. Portanto, utilizar para o teste os 916 produtos ou apenas os 10 encontrados pelo algoritmo, acarreta em um mesmo escore de mutação, porém a segunda opção é menos custosa. O testador tem ainda a liberdade de escolher conjuntos com menos produtos que este, porém eles não cobrirão 100% dos defeitos descritos pelo teste de mutação. Para a LPS da matriz M1, ambos os algoritmos encontraram conjuntos de casos de teste com 27 produtos e cobertura de 100%.

A principal vantagem de utilizar esta abordagem é o baixo esforço do testador em gerenciar e utilizar os casos de teste. Basta que ele defina o mínimo de cobertura exigido (idealmente 100%), execute o algoritmo desejado e escolha o conjunto já reduzido que mais se adapta as suas necessidades. Outro fator importante é a redução do uso de recursos computacionais. Como apresentado no exemplo, utilizar apenas 10 produtos

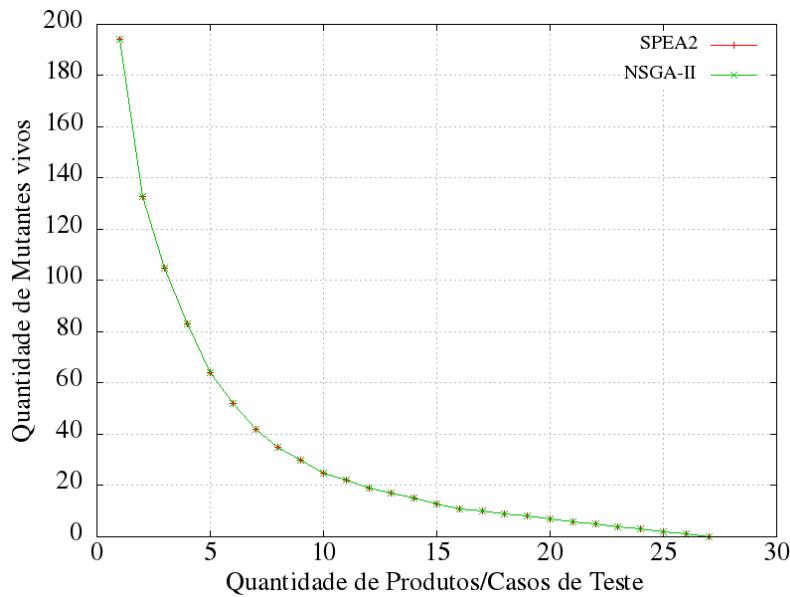


Figura 2. Melhores PF_{approx} para M1

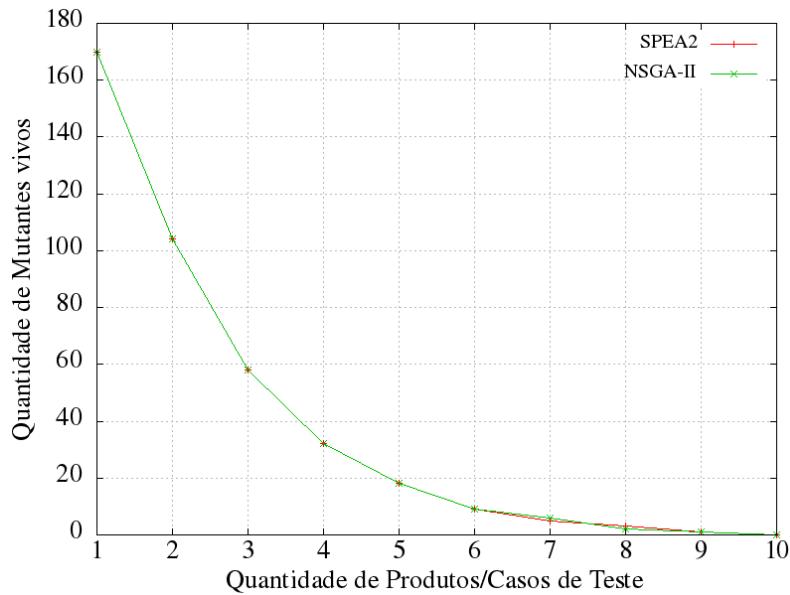


Figura 3. Melhores PF_{approx} para M2

dentre os 916 disponíveis (equivalente a 1,09% da quantidade total de produtos) e obter 100% de cobertura de teste, diminui consideravelmente a quantidade de testes a serem feitos, reduzindo o tempo necessário para a atividade de teste.

Com relação à aplicabilidade da técnica proposta no presente trabalho, por se tratar de LPS, o número de produtos derivados pode ser grande, e diante de evoluções da LPS que impliquem em mudanças no modelo de características, é mais viável reexecutar o algoritmo para selecionar e testar o menor número de produtos possível, do que testar todos os produtos para garantir que o modelo de características satisfaz os requisitos da LPS. Como o trabalho de Ferreira et al. (2013) fornece apoio automatizado, o esforço

Tabela 4. Melhor PF_{approx} encontrada para M2

Conjunto de Produtos	Produtos	Mutantes Vivos	Cobertura (escore)
{872}	1	170	33,07%
{524 905}	2	104	59,05%
{572 794 903}	3	58	77,16%
{487 604 794 861}	4	32	87,40%
{487 604 734 881 891}	5	18	92,91%
{487 532 604 734 881 891}	6	9	96,45%
{487 532 591 642 734 872 903}	7	6	97,63%
{487 532 604 646 734 881 891}	7	6	97,63%
{487 493 532 604 734 881 891}	7	6	97,63%
{487 532 604 675 734 881 891}	7	6	97,63%
{487 491 532 604 734 881 891}	7	6	97,63%
{487 532 604 676 734 881 891}	7	6	97,63%
{487 489 512 591 638 734 872 903}	8	2	99,21%
{21 487 489 512 591 638 734 872 903}	9	1	99,60%
{23 487 489 512 591 638 734 872 903}	9	1	99,60%
{487 489 498 512 591 638 734 872 903}	9	1	99,60%
{487 489 495 512 591 638 734 872 903}	9	1	99,60%
{487 489 493 512 591 638 734 872 903}	9	1	99,60%
{487 489 497 512 591 638 734 872 903}	9	1	99,60%
{487 489 491 512 591 638 734 872 903}	9	1	99,60%
{17 487 489 512 591 638 734 872 903}	9	1	99,60%
{21 487 489 497 512 591 638 734 872 903}	10	0	100%
{21 487 489 499 512 591 638 734 872 903}	10	0	100%
{21 487 489 491 512 591 638 734 872 903}	10	0	100%
{23 487 489 493 512 591 638 734 872 903}	10	0	100%
{17 21 487 489 512 591 638 734 872 903}	10	0	100%
{17 22 487 489 512 591 638 734 872 903}	10	0	100%
{21 487 489 494 512 591 638 734 872 903}	10	0	100%
{21 487 489 497 512 591 734 872 903 907}	10	0	100%
{23 487 489 493 512 591 734 872 877 903}	10	0	100%

necessário para obter uma nova relação de quais produtos matam quais mutantes e depois reexecutar a técnica proposta acaba não sendo tão grande.

7. Conclusão

Este trabalho apresentou a aplicação de algoritmos evolutivos multiobjetivos na resolução do problema de seleção de casos de teste em duas situações reais de LPS. Foram utilizados dois algoritmos (NSGA-II e SPEA2) com o intuito de diminuir a quantidade de casos de teste e aumentar a cobertura dos testes (escore de mutação), considerando o teste de mutação de variabilidades.

Os resultados experimentais mostram que, mesmo sem diferenças estatísticas entre os algoritmos, o NSGA-II foi ligeiramente mais eficiente que o SPEA2 na qualidade dos resultados e duas vezes mais rápido na velocidade de execução. Entretanto, esse resultado não descarta o uso do SPEA2, uma vez que este algoritmo também apresentou resultados satisfatórios. Isso pode ser constatado com base no resultado final, onde, para ambos algoritmos, foram obtidos escores de 100% com o uso de conjuntos de 27 produtos dentre os 43 disponíveis para a LPS da matriz M1 e com o uso de 10 produtos dentre os 916 disponíveis para a LPS de M2. Portanto, utilizar todos os casos de teste disponíveis ou somente o conjunto reduzido gera um mesmo resultado de escore. Essa minimização acarreta em uma execução mais rápida e menos custosa dos testes de LPS, sem que haja um prejuízo na qualidade dos resultados, de acordo com o teste de mutação.

Como trabalhos futuros, pode-se repetir o experimento realizado utilizando outras LPS, bem como avaliar outros algoritmos multiobjetivos para a resolução do problema, como por exemplo Otimização por Nuvem de Partículas (PSO) e Otimização por Colônia de Formigas (ACO). Além disso, podem ser propostos outros operadores de mutação e cruzamento, de modo a aprimorar a qualidade dos resultados.

Referências

- Arcuri, A. and Fraser, G. (2011). On parameter tuning in search based software engineering. In *Proceedings of the Third International Symposium on Search Based Software Engineering*, SSBSE'11, pages 33–47, Szeged. Springer-Verlag.
- Bergey, J. K., Jones, L. G., Sholom, C., Donohoe, P., and Northrop, L. (2010). Software Product Lines : Report of the 2010 U.S. Army Software Product Line Workshop. Technical Report June, Carnegie Mellon University, Pittsburgh, PA.
- Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. V. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition*. Springer Science+Business Media, LLC, 2nd edition.
- Cohen, M. B., Dwyer, M. B., and Shi, J. (2006). Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 53–63, Lincoln, Nebraska, USA. Department of Computer Science and Engineering, University of Nebraska, ACM.
- 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.
- Durillo, J. J. and Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771.
- Ferreira, J. M., Vergilio, S. R., and Quinaia, M. (2013). A Mutation Based Approach to Feature Testing of Software Product Lines. In *The 25th International Conference on Software Engineering and Knowledge Engineering*, SEKE'13, Boston, MA.
- van der Linden, F., Schimid, K., and Rommes, E. (2007). *Software Product Lines in Action*. Springer-Verlag Berlin Heidelberg, Berlin, Germany.
- Weissleder, S., Sokenou, D., and Schlinglo, B.-H. (2008). Reusing State Machines for Automatic Test Generation in Product Lines. In *1st Workshop on Model-based Testing in Practice (MoTiP 2008)*, Berlin.
- Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83.
- Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical report, Department of Electrical Engineering, Swiss Federal Institute of Technology, Zurich, Suíça.
- Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.

Um Estudo Comparativo de Heurísticas Aplicadas ao Problema de Clusterização de Módulos de Software

Alexandre F. Pinto, Adriana C. de F. Alvim, Márcio de O. Barros

Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

alexandre.pinto@uniriotec.br, marcio.barros@uniriotec.br,
adriana@uniriotec.br

Abstract. This paper describes a comparative evaluation of heuristics applied to the problem of Software Module Clustering. The heuristics cover three different approaches: constructive methods, local search methods, and meta-heuristics. Comparative studies were performed with a greedy algorithm, a Hill Climbing search with multiple restarts, and two meta-heuristics: genetic algorithms and Iterated Local Search. In our experiments, the Iterated Local Search meta-heuristic yielded better results than the local search and the genetic algorithm for the selected instances used.

Resumo. Este artigo descreve um estudo experimental comparativo de heurísticas aplicadas ao problema de Clusterização de Módulos de Software. As heurísticas abrangem três diferentes abordagens: métodos construtivos, métodos de busca local e meta-heurísticas. Foram realizados estudos comparativos com um algoritmo guloso, um algoritmo de busca local usando a técnica Hill Climbing com múltiplos reinícios e duas meta-heurísticas, algoritmos genéticos e Iterated Local Search. Em nossos experimentos, a meta-heurística Iterated Local Search obteve melhores resultados que o algoritmo de busca local e o algoritmo genético para as instâncias utilizadas.

1. Introdução

Clusterização de Módulos de Software (CMS) é o problema de distribuir os módulos componentes de um projeto de software entre estruturas maiores (denominadas *clusters*). Entre as vantagens oferecidas por uma boa distribuição de módulos em *clusters* podemos citar: organizá-los funcionalmente [Larman, 2002], oferecer melhor navegação entre os componentes do software [Gibbs e Tsichritzis, 1990] e aumentar a compreensão do software [McConnell, 2004].

Entre os trabalhos que envolvem o uso de técnicas baseadas em buscas para encontrar soluções para o CMS, podemos mencionar os trabalhos realizados pelo grupo da Drexel University [Doval, Mancoridis e Mitchell, 1999][Harman, Hierons e Proctor, 2002]. Diversos estudos experimentais conduzidos por este grupo mostraram que os resultados obtidos pelos algoritmos de busca local, utilizando a técnica de *Hill Climbing* com múltiplos reinícios, se mostraram superiores em relação aos resultados obtidos pelos algoritmos genéticos mono-objetivo. Estes resultados motivaram o presente estudo, que tem como objetivo realizar uma análise comparativa de três abordagens aplicadas ao CMS: construtiva, busca local e meta-heurística.

2. Clusterização de Módulos de Software

O problema de clusterização de módulos de software (CMS) consiste em organizar em grupos os componentes unitários do software, denominados módulos. Componentes melhor agrupados e integrados funcionalmente implicam em um software mais fácil de manter, testar e evoluir. Esta reorganização é realizada visando otimizar características que indiquem a boa qualidade do software, tais como a coesão e o acoplamento [Yourdon e Constantine 1979]. Os agrupamentos de módulos são denominados clusters, podendo ser representados como pacotes ou *namespaces*.

Acoplamento é uma medida de dependência entre os componentes de um software, enquanto que coesão é uma medida de quanto o componente está dedicado a resolver um único problema. Usualmente, o acoplamento de um software é medido através do número de dependências entre módulos pertencentes a diferentes clusters, enquanto que a coesão é medida através das dependências entre módulos pertencentes a um mesmo cluster. O presente trabalho tem como objetivo distribuir os módulos do software em um determinado número de clusters de forma a minimizar o acoplamento e maximizar a coesão, obedecendo a restrição de que cada módulo deve pertencer a um único cluster. Para medir a qualidade da clusterização, será utilizada a função de avaliação *Modularization Quality* (MQ) [Doval, Mancoridis e Mitchel, 1999]. Esta função considera um balanço entre a coesão e o acoplamento dos clusters, recompensando clusters com módulos que tenham muitas dependências com outros módulos do mesmo cluster e penalizando aqueles com muitas dependências com módulos de outros clusters. As heurísticas propostas neste trabalho têm como objetivo maximizar o valor de MQ.

3. Estudo comparativo

3.1. Instâncias utilizadas

Os experimentos foram realizados em um computador com processador Intel Core i7-2600 3.40 GHz, com memória 4GB e dedicação exclusiva. Doze instâncias (Tabela 1) foram coletadas a partir de aplicações reais, utilizando a ferramenta PF-CDA (<http://www.dependency-analyzer.org/>), algumas destas tendo sido utilizadas em outras publicações que abordaram o CMS [Barros, 2012]. A primeira coluna exibe um identificador para a instância, a segunda exibe o nome do software, a terceira o número de módulos e a última coluna exibe o número de dependências entre os módulos.

Tabela 1. Instâncias utilizadas.

Identificador	Descrição	Módulos	Dependências
JNANOXML	Parser de XML para Java	25	64
APACHE	Utilitário para compressão de arquivo	36	86
JSCATTERPLOT	Biblioteca para gráficos de dispersão (parte do JTreeview) v1.1.6	74	232
JUNIT	JUnit:biblioteca para testes unitários	100	276
TINYTIM	Tiny TIM: Topic Maps Engine	134	564
GAE_PLUGIN	Google Plugin para Eclipse	140	375
JDENDOGRAM	Biblioteca para gráficos dendogramas (parte do JTreeview) v.1.1.6	177	583
PDF_RENDERER	Renderizador Java para arquivos PDF v0.2.1	199	629
JUNG_VISUALIZATION	Jung Graph – visualização de classes v2.0.1	221	919
PFCDA_SWING	Classes de interface de software para análise de código fonte v1.1.1	252	885
JML	Biblioteca Java para MSN Messenger v1.0	270	1745
NOTE PAD_FULL	Editor para tablets v0.2.1	299	1349

3.2.Algoritmos utilizados

O algoritmo construtivo utilizado foi o *Fast Greedy Modularity Optimization* [Clauset, Newman e Moore, 2004]. A busca local implementada utiliza a estratégia de *Hill Climbing* com múltiplos reinícios aleatórios [Mancoridis et al., 1998].

O algoritmo *Iterated Local Search* utilizou como método de geração da solução inicial o algoritmo guloso *Fast Greedy Modularity Optimization* [Clauset, Newman e Moore, 2004]. Como método de busca local foi utilizada a técnica *Hill Climbing*. Para o método de perturbação foram utilizadas duas estratégias: a movimentação de módulos entre clusters e a troca de módulos entre clusters distintos.

Para configurar o algoritmo genético utilizado no estudo comparativo foi realizado um estudo inicial que avaliou diversas configurações, considerando o tamanho da população, o operador de recombinação e o uso de elitismo. A configuração que obteve os melhores resultados neste estudo foi selecionada para comparação com as demais técnicas. Esta configuração utiliza operador de recombinação 2-point crossover, operador de seleção por torneio e elitismo de 30%. O operador de mutação utilizado foi mutação uniforme, com percentual de 0.4% vezes $\log n$, onde n é o número de módulos.

3.3. Estudo comparativo entre as abordagens heurísticas

A Tabela 2 apresenta o resultado do algoritmo guloso, do *Hill Climbing*, do *Iterated Local Search* e do algoritmo genético. São apresentados o tempo de execução, em segundos, e os valores de MQ obtidos em cada heurística. A melhor média de cada instância é apresentada em negrito. Com exceção do método guloso, os demais métodos utilizaram como critério de parada um número máximo de avaliações de soluções candidatas igual a 200 vezes n^2 , sendo n o número de módulos. Os algoritmos *Hill Climbing*, *Iterated Local Search* e algoritmo genético foram executados 30 vezes para cada instância. Não houve necessidade de cálculo de média para o algoritmo guloso, uma vez que este retorna sempre a mesma solução para uma dada instância.

O *Iterated Local Search* obteve melhores resultados em 11 das 12 instâncias. Os resultados para o problema CMS indicam que o reinício partindo de uma solução modificada (perturbação) a partir de um ótimo local foi mais vantajoso que o simples reinício partindo de uma solução aleatória, conforme utilizado na busca local.

Tabela 2. Comparativo entre as abordagens heurísticas.

INSTÂNCIA	Guloso		Hill Climbing		Iterated Local Search		Algoritmo Genético	
	Tempo (s)	MQ	Tempo (s)	Média (MQ)	Tempo (s)	Média (MQ)	Tempo (s)	Média (MQ)
JNANOXML	0,02	3,77	0,01	$3,81 \pm 0,02$	0,01	$3,82 \pm 0,00$	0,1	$3,79 \pm 0,03$
APACHE	≈ 0	5,75	0,02	$5,74 \pm 0,02$	0,02	$5,77 \pm 0,00$	0,3	$5,75 \pm 0,02$
JSCATTERPLOT	0,01	10,68	0,08	$10,64 \pm 0,03$	0,10	$10,74 \pm 0,01$	2,8	$10,65 \pm 0,07$
JUNIT	0,02	11,08	0,13	$11,06 \pm 0,04$	0,19	$11,09 \pm 0,00$	6,7	$10,92 \pm 0,06$
TINYTIM	0,06	12,35	0,29	$12,26 \pm 0,04$	0,41	$12,49 \pm 0,02$	20,7	$11,59 \pm 0,15$
GAE_PLUGIN	0,05	17,27	0,25	$17,32 \pm 0,02$	0,38	$17,29 \pm 0,02$	20,9	$16,25 \pm 0,19$
JDENDOGRAM	0,13	25,95	0,45	$26,02 \pm 0,02$	0,64	$26,07 \pm 0,01$	49,0	$22,05 \pm 0,26$
PDF_RENDERER	0,17	21,31	0,55	$21,34 \pm 0,18$	0,88	$21,78 \pm 0,16$	72,6	$18,29 \pm 0,30$
JUNG_VISUALIZATION	0,24	20,20	0,75	$20,44 \pm 0,17$	1,09	$20,86 \pm 0,15$	107,6	$16,42 \pm 0,33$
PFCDA_SWING	0,35	28,79	0,93	$28,72 \pm 0,07$	1,33	$28,95 \pm 0,03$	161,3	$19,95 \pm 0,36$
JML	0,54	17,18	1,49	$17,20 \pm 0,10$	2,21	$17,38 \pm 0,04$	231,0	$12,11 \pm 0,28$
NOTEPAD_FULL	0,64	29,26	1,47	$29,07 \pm 0,07$	2,17	$29,43 \pm 0,05$	296,9	$18,8 \pm 0,42$

4. Conclusões

O presente trabalho apresentou um estudo comparativo entre diferentes abordagens para a solução heurística do CMS, com a utilização da função de avaliação mono-objetivo MQ (*Modularization Quality*).

Em termos de qualidade de solução, *Iterated Local Search* se mostrou superior quando comparado com os demais métodos. Para um conjunto de 12 instâncias, o ILS obteve 11 soluções de melhor qualidade. A abordagem construtiva, isoladamente, também se mostrou eficiente quando comparada com a busca local e os algoritmos genéticos. O método guloso encontrou 6 (entre 12) soluções melhores do que a busca local e 10 soluções melhores que o algoritmo genético.

Como trabalhos futuros, para comparação com os resultados obtidos pelo método baseado em ILS, recomenda-se a implementação de uma abordagem baseada na meta-heurística GRASP (*Greedy Random Adaptative Search Procedure*). GRASP combina um método construtivo com busca local. Para finalizar, sugere-se o uso do método guloso *Fast Greedy Modularity Optimization* [Clauset, Newman e Moore, 2004] para gerar soluções iniciais para o CMS em abordagens baseadas em busca local e meta-heurísticas.

Agradecimentos

Os autores agradecem à CAPES e ao CNPq pelo auxílio financeiro recebido por este projeto.

Referências

- Barros, M. (2012), “An analysis of the effects of composite objectives in multiobjective software module clustering”. Proceedings of the 14th international conference on Genetic and evolutionary computation conference - GECCO '12, p. 1205.
- Clauset, A.; Newman, M. E. J., Moore, C. (2004), “Finding community structure in very large networks”, Physical Review E, pp. 1- 6.
- Doval, D., Mancoridis, S. and Mitchell, B. S. (1999). “Automatic clustering of software systems using a genetic algorithm”. STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice, p. 73–81.
- Gibbs, S., Tsichritzis, D. , Casais, E., Nierstrasz, O., Pintado, X. (1990). “Class Management for Software Communities”, Communications of the ACM, v. 33, n. 9, pp.90-103
- Harman, M., Hierons, R. M., Proctor, M. (2002), “A New Representation And Crossover Operator for Search-based Optimization of Software Modularization”, in Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA, pp. 1351-1358
- Larman, C. (2002), “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and the Unified Process”. Prentice Hall, Upper Saddle River, NJ.
- Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., Gansner, E. R. (1998) “Using automatic clustering to produce high-level system organizations of source code”. In International Workshop on Program Comprehension (IWPC'98), pages 45–53, Los Alamitos, California.
- McConnell, S. (2004), “Code Complete”, Second Edition, Microsoft Press.
- Yourdon, E. and Constantine, L. L. (1979), “Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design”. Prentice-Hall.

Modelo Computacional para Apoiar a Configuração de Produtos em Linha de Produtos de Software

Juliana Alves Pereira, Eduardo Figueiredo, Thiago Noronha

Laboratório de Engenharia de Software (LabSoft), Departamento de Ciência da Computação (DCC), Universidade Federal do Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

{juliana.pereira, figueiredo, tfn}@dcc.ufmg.br

Abstract. *Software Product Line (SPL) is a set of software systems QUE sharing a set of characteristics to satisfy specific needs of a particular domain. The configuration of a product among thousands of possible combinations of features has proved to be impractical even for small SPL. To support companies in semiautomatic product extraction and maximize customer satisfaction, this paper proposes a model based on search algorithms. The model has two implementations: (i) exhaustive enumeration with preprocessing and backtracking and (ii) a greedy algorithm. Due the NP-complete nature of the problem, our experiments revealed that the exhaustive enumeration is impractical for larger instances. Moreover, the greedy heuristic implementation solves the problem in polynomial time with 92% of accuracy.*

Resumo. *Linha de produtos de software (LPS) é um conjunto de sistemas de software que compartilham um conjunto de características, e que satisfazem as necessidades específicas de um determinado segmento de mercado. A configuração de produtos, dentre as milhares de combinações de características possíveis, tem se mostrado inviável mesmo para pequenas LPS. A fim de apoiar as empresas na configuração semi-automática de produtos que maximizem a satisfação de clientes, este trabalho propõe um modelo baseado em algoritmos de busca e otimização. O modelo foi implementado utilizando duas soluções: (i) algoritmos de enumeração exaustiva com pré-processamento e backtracking e (ii) heurística gulosa. Devido à complexidade NP-completo, os experimentos realizados mostraram que a solução por enumeração exaustiva é inviável para grandes instâncias do problema. Por outro lado, a implementação da heurística gulosa resolve o problema em tempo polinomial com uma taxa de acerto de 92%.*

1. Introdução

A crescente necessidade de desenvolvimento de sistemas de software maiores e mais complexos exige melhor suporte para reutilização de software [10]. Muitas técnicas, como Linha de Produtos de Software (LPS), têm sido propostas para oferecer avançado suporte a reutilização de software. LPS pode ser definida como um conjunto de sistemas de software definidos sobre uma arquitetura comum, que compartilham um mesmo conjunto de características e satisfazem as necessidades específicas de um determinado segmento de mercado [10]. Uma característica é um incremento na funcionalidade de um sistema de software [1]. LPS explora o fato de sistemas em um mesmo domínio serem semelhantes e terem potencial para reutilização.

Grandes empresas de software, tais como Boeing, Hewlett Packard, Nokia e Siemens, adotam LPS para desenvolver seus produtos [11]. LPS permite às empresas

rápida entrada no mercado por fornecer capacidade de reutilização em larga escala com customização em massa [10]. Pela adoção desta técnica, espera-se obter ganhos na produtividade, produtos mais confiáveis e preço mais acessível [14]. Esses potenciais benefícios prometem um aumento significativo na satisfação do cliente. Entretanto, a customização de produtos para maximizar a satisfação do cliente é uma tarefa cuja complexidade aumenta exponencialmente à medida que aumenta o número de características envolvidas e restrições a serem consideradas na LPS [1][2][3][8].

Para minimizar este problema, este trabalho propõe um modelo para apoiar a configuração de produtos em LPS através da utilização de algoritmos de busca e otimização. O objetivo do modelo é maximizar a satisfação do cliente. Este modelo foi implementado inicialmente utilizando algoritmos de enumeração exaustiva com pré-processamento e backtracking [5]. Devido à natureza NP-Completo [5] deste problema, à medida que a instância do problema cresce, estes algoritmos passam a exigir muitos recursos computacionais (essencialmente, tempo de processamento) para encontrar a solução ótima. Portanto, é proposta a utilização de um algoritmo guloso de busca, baseado em heurísticas, para encontrar soluções suficientemente boas em um menor horizonte de tempo.

Os estudos experimentais realizados mostram a complexidade exponencial da resolução por enumeração exaustiva, mesmo com o uso de pré-processamento e backtracking. Por outro lado, o algoritmo guloso proposto possui complexidade polinomial. Além do ganho em recursos computacionais, o algoritmo guloso apresenta uma taxa de acerto de 92% em relação aos algoritmos de enumeração exaustiva como mostram os resultados dos experimentos.

O restante do artigo está organizado da seguinte forma. A Seção 2 introduz os conceitos fundamentais de LPS e define o problema tratado pelo trabalho. A Seção 3 apresenta a modelagem do problema e os algoritmos implementados. A Seção 4 discute os resultados dos experimentos computacionais realizados. A Seção 6 conclui o trabalho e propõe direções para trabalhos futuros.

2. O Problema de Configuração de Produtos em LPS

O problema de configuração de produtos em LPS corresponde ao problema de customizar um produto para atender as necessidades específicas de um cliente. A Seção 2.1 introduz os conceitos de LPS e a Seção 2.2 descreve o problema tratado pelo trabalho.

2.1. Linha de Produtos de Software

Em uma Linha de Produtos de Software (LPS), características comuns a um domínio formam o núcleo e outras características definem pontos de variação [10]. Uma característica é um incremento na funcionalidade de um sistema de software [1]. Em uma LPS, o domínio do sistema é decomposto em características coerentes, bem definidas, independentes e facilmente combináveis. Assim, diferentes configurações de produtos podem ser possíveis mediante a inclusão ou exclusão de determinadas características [4]. A comercialização de produtos que se diferenciam por variações em suas características está se tornando comum e grandes empresas têm investido em LPS [11]. Por exemplo, os automóveis de um mesmo modelo se diferenciam por itens como airbag. Existem características comuns a todos os produtos e outras características podem variar de um produto para outro.

Modelo de características é o padrão para representar a variabilidade de uma LPS e o espaço de possíveis configurações de produtos [13]. Um modelo de características, geralmente representado utilizando árvores, possibilita a visualização hierárquica das características de uma LPS e suas relações [12][13]. As características do modelo de características são classificadas em mandatórias, opcionais, alternativas não exclusivas (*OR*) e alternativas exclusivas (*XOR*).

As características mandatórias, representadas por um círculo cheio na Figura 1a, devem obrigatoriamente estar presente na composição de um produto. As características opcionais, representadas por um círculo vazio na Figura 1b, podem opcionalmente estar presentes na composição de um produto. No caso do grupo de características alternativas não exclusivas, representadas por arestas interligadas e conectadas por um arco cheio na Figura 1c, uma ou mais sub-características podem ser selecionadas para composição de um produto. No grupo de características alternativas exclusivas, representado por arestas interligadas e conectadas por um arco vazio na Figura 1d, apenas uma sub-característica pode ser selecionada para composição de um produto.

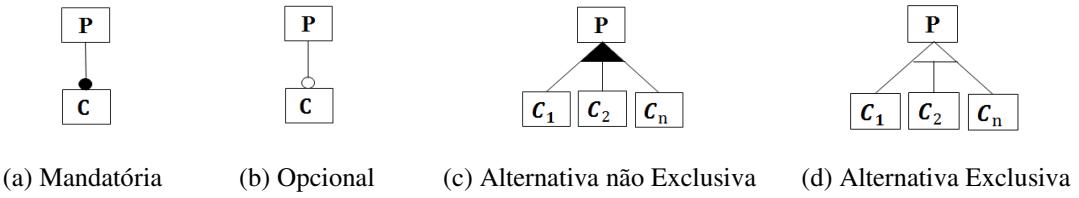


Figura 1 Classificação das características

Além das características e suas relações, um modelo de características pode incluir regras de composição referentes às restrições adicionais para combinações de características [10]. As regras de composição de características são responsáveis pela validação da configuração de produtos. Elas normalmente descrevem uma restrição de inclusão ou de exclusão enunciadas da forma: se a característica C₁ for incluída, então a característica C₂ também deve ser incluída (ou excluída). Por exemplo, no modelo de características de um automóvel (vide Figura 2). Onde, *Som Automotivo* e *Bateria do Tipo 1* são características da LPS. Uma regra de composição: *Som Automotivo* requer (inclusão) *Bateria do Tipo 1* é definida. Neste caso, se a característica *Som Automotivo* for selecionada para compor um produto, obrigatoriamente a característica *Bateria do Tipo 1* deve ser selecionada para compor o mesmo produto.

2.2. Descrição do Problema

A configuração de produtos em LPS corresponde a customizar produtos para clientes baseando-se em restrições (como por exemplo, preferências e orçamento). Isso implica em selecionar características da LPS que satisfaçam restrições impostas pelas empresas fornecedoras do produto e pelos clientes que irão adquirir o produto. Na implementação deste problema, como exemplificado pela Figura 2, as restrições são representadas pelo orçamento disponível para aquisição do produto, preço e grau de importância das características que compõem o produto. Assim, com base no orçamento que o cliente dispõe para aquisição do produto, o objetivo é otimizar a configuração de características e apoiar as empresas na configuração de produtos que maximizem a satisfação de clientes.

As principais constantes definidas pelo modelo do problema são: (i) o limite de orçamento que o cliente possui para aquisição do software ($O \in \text{Reais}$); (ii) o grau de importância da característica i para o cliente ($G_{1\dots n} \in [0, 1]$); (iii) o preço em adicionar a característica i ao produto a ser entregue ao cliente ($P_{1\dots n} \in \text{Reais}$); e (iv)

a expressão proposicional na forma normal conjuntiva derivada a partir da árvore modelada ($E(C)$). Onde, $C(n) \in \text{Naturais}$ representa a quantidade de características presentes no modelo e $C_i \in \{0, 1\}$ representa a não inclusão ou inclusão da característica i no produto configurado. Adicionalmente, todas as restrições definidas pelo modelo devem ser satisfeitas.

$$\text{A função objetivo é maximizar } z(n) = \sum_{i=1}^n G_i C_i .$$

$$\text{Sujeito a: } \sum_{i=1}^n P_i C_i \leq O, \text{ satisfazendo } E(C) .$$

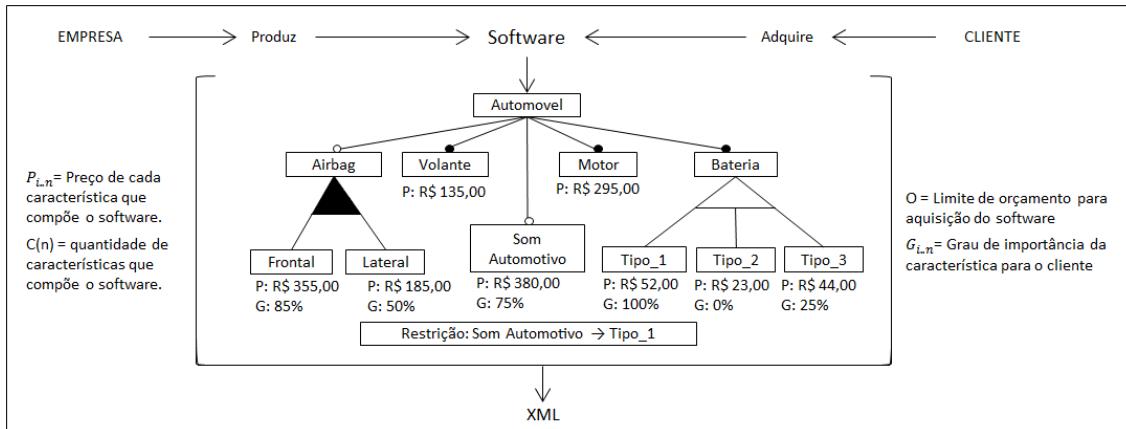


Figura 2 Modelo para apoiar a configuração de produtos em uma LPS

3. Modelo Computacional Proposto

Durante a configuração de produtos, o crescimento exponencial da quantidade de combinações possíveis das características em um modelo varia em relação à quantidade de características que o modelo possui. A Seção 3.1 descreve o modelo de otimização utilizado, e as Seções 3.2 e 3.3 apresentam a implementação de uma solução exponencial (algoritmo de enumeração exaustiva) e polinomial (solução gulosa) implementadas para o problema.

3.1. Modelo de Otimização

O problema de otimização combinatória descrito na Seção 2 pertence à classe de complexidade NP-completo [5]. Uma proposta para a análise automatizada de modelos de características é baseada no mapeamento dos modelos em resolvidores SAT (problema da satisfazibilidade booleana) [3]. As regras para a tradução de modelos de características às restrições são listadas na Tabela 1. Estas regras possibilitam a modelagem do problema através de problemas conhecidos de otimização. Modelamos o problema como uma combinação de MAX-SAT Ponderado [9] com o Problema da Mochila [7].

Tabela 1. Representação matemática dos tipos de características para SAT

Relação	SAT
Mandatória	$(\neg P \vee C) \wedge (\neg C \vee P)$
Opcional	$\neg C \vee P$
Alternativa não Exclusiva	$(\neg P \vee C_1 \vee C_2 \vee \dots \vee C_n) \wedge (\neg C_1 \vee P) \wedge (\neg C_2 \vee P) \wedge \dots \wedge (\neg C_n \vee P)$
Alternativa Exclusiva	$(C_1 \vee C_2 \vee \dots \vee C_n \vee \neg P) \wedge (\neg C_1 \vee \neg C_2) \wedge \dots \wedge (\neg C_1 \vee \neg C_3) \wedge (\neg C_1 \vee P) \wedge (\neg C_2 \vee \neg C_3) \wedge \dots \wedge (\neg C_2 \vee \neg C_n) \wedge (\neg C_3 \vee \neg C_n) \wedge \dots \wedge (\neg C_{n-1} \vee \neg C_n) \wedge \dots \wedge (\neg C_{n-1} \vee P) \wedge (\neg C_n \vee P)$

O modelo de otimização proposto neste trabalho usa inicialmente o mapeamento de modelo de características para SAT [3]. O SAT é um problema da lógica proposicional. Seu objetivo é determinar um conjunto de valores (verdadeiro ou falso) para as variáveis proposicionais que faça uma dada expressão na forma normal conjuntiva (FNC) satisfazível - ou mostre que nenhum conjunto de valores a satisfaz [5]. Para a modelagem do problema de otimização SAT é necessário representar o modelo de características como um conjunto das relações traduzidas seguindo as regras da Tabela 1. Além das regras descritas, uma restrição adicional é a inclusão da raiz em todas as cláusulas (Figura 1). Por exemplo, para a resolução do problema como um modelo de otimização SAT, decomponemos o modelo de características apresentado na Figura 2 na expressão abaixo.

$$E(C) = ((\neg \text{Automovel} \vee \text{Volante}) \wedge (\neg \text{Volante} \vee \text{Automovel})) \wedge ((\neg \text{Automovel} \vee \text{Motor}) \wedge (\neg \text{Motor} \vee \text{Automovel})) \wedge ((\neg \text{Automovel} \vee \text{Bateria}) \wedge (\neg \text{Bateria} \vee \text{Automovel})) \wedge ((\neg \text{Som_Automotivo} \vee \text{Automovel}) \wedge (\neg \text{Airbag} \vee \text{Automovel})) \wedge ((\neg \text{Bateria} \vee \text{Tipo_1} \vee \text{Tipo_2} \vee \text{Tipo_3}) \wedge (\neg \text{Tipo_1} \vee \text{Bateria}) \wedge (\neg \text{Tipo_2} \vee \text{Bateria}) \wedge (\neg \text{Tipo_3} \vee \text{Bateria}) \wedge ('\text{Tipo_1} \vee \neg \text{Tipo_2}) \wedge (\neg \text{Tipo_2} \vee \neg \text{Tipo_3})) \wedge ((\neg \text{Tipo_2} \vee \text{Frontal}) \wedge (\neg \text{Frontal} \vee \text{Tipo_2})) \wedge ((\neg \text{Airbag} \vee \text{Frontal} \vee \text{Lateral}) \wedge (\neg \text{Frontal} \vee \text{Airbag}) \wedge (\neg \text{Lateral} \vee \text{Airbag}))$$

Após a decomposição do modelo em uma expressão na FNC, o problema é modelado como um problema de otimização MAX-SAT Ponderado. O problema MAX-SAT Ponderado é uma unidade do problema SAT. Ele atribui um peso a cada cláusula [9]. Denotaremos esses pesos por $G_{1\dots n}$, onde o objetivo é selecionar as características com um maior grau de importância para o cliente. Posteriormente, a analogia com o Problema da Mochila é realizada. No Problema da Mochila temos uma situação em que o objetivo é preencher uma mochila com o maior valor possível, dado objetos de diferentes pesos e valores, não ultrapassando a capacidade máxima da mochila [7]. Fazendo uma analogia ao problema abordado neste trabalho, o objetivo é obter uma configuração de produto composto por características que somam um maior grau de importância (valor) para o cliente e que não ultrapassem o orçamento disponível (capacidade máxima), baseando-se nos preços de desenvolvimento de cada uma das características.

3.2. Algoritmos de Enumeração Exaustiva

Por ser um problema combinatório, é possível encontrar a melhor configuração do produto por enumeração exaustiva. Ou seja, podem-se enumerar todas as soluções possíveis, comparando-as para identificar a melhor solução. A instância deste problema de decisão é uma expressão booleana escrita somente com operadores *AND*, *OR*, *NOT*, *variáveis* e *parênteses*. Para resolução, as variáveis recebem valores binários e as expressões são solucionadas através da representação por uma árvore, na qual as folhas são operandos e os nós internos são operadores. A enumeração de todos os produtos que possam ser extraídos da LPS analisada gera todas as sequências possíveis de dados.

Este trabalho implementou inicialmente o algoritmo de enumeração exaustiva com pré-processamento e backtracking. O algoritmo de pré-processamento tem como objetivo condensar o modelo de características analisado. Assim, características classificadas como mandatórias, características que possuem um grau de importância elevado para o cliente e características que excedem o orçamento disponível são pré-processadas e agrupadas na árvore (juntamente com a soma de seus preços de

desenvolvimento). A Figura 3 mostra um exemplo da execução do algoritmo de pré-processamento, para o modelo de características apresentado na Figura 2.

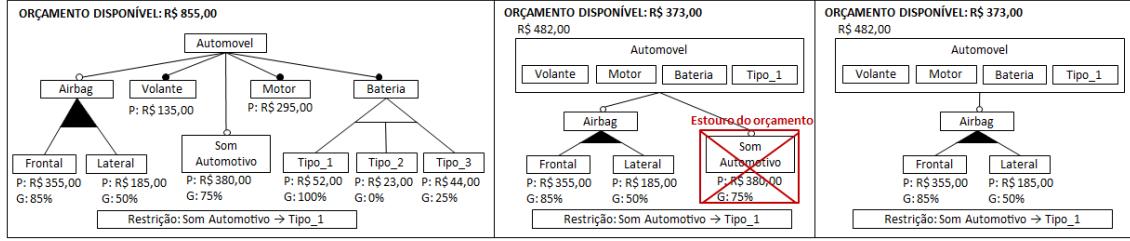


Figura 3 Pré-processamento do modelo de características

Backtracking é um algoritmo refinado da busca por enumeração exaustiva, no qual boa parte dos caminhos que não levam à solução desejada são eliminados sem serem explicitamente examinados [5]. Para a implementação do backtracking, o algoritmo analisa o preço e grau de importância de cada uma das características presentes no modelo. Um dos pontos é detectar através de restrições do problema as características e/ou grupos de características mortas (*dead features*). Ou seja, a detecção de características que por influência das restrições e do resultado parcial da configuração se tornam inutilizáveis, não podendo ser oferecidas para o cliente. A eficiência desta estratégia depende das possibilidades de limitar a busca (poda da árvore), eliminando os ramos que não levam à solução desejada.

Apesar de estes algoritmos apresentarem métodos exatos de resolução, conduzindo a soluções ótimas globais. A eficiência do algoritmo de enumeração exaustiva com pré-processamento e backtracking pode ser comprometida por problemas como: (i) um número irrelevante de características mandatórias presentes no modelo e (ii) restrições do modelo não favorecem a realização de podas. Assim, sua formulação e/ou resolução exata pode levar a uma complexidade intratável por seu longo tempo de execução, dado que a quantidade de soluções possíveis para o problema cresce exponencialmente em função do número de características. Logo, optou-se por colocar em segundo plano a solução ótima e buscar algoritmos que forneçam boas soluções em tempo polinomial. Dessa forma, é necessário à implementação de um algoritmo não exato (heurística) para maximização de ganhos de eficiência e de custo. Como solução para o problema, utilizamos a heurística gulosa descrita a seguir.

3.3. Heurística de Solução Gulosa

Este trabalho propõe um algoritmo que usa heurística gulosa. A heurística gulosa é aplicada a problemas que necessitam obter um subconjunto que satisfaça algumas restrições. Qualquer subconjunto que satisfaça as restrições é chamado de solução viável [5]. O que queremos é uma solução viável que maximize a função objetivo descrita na Seção 2.2. Chamamos a solução viável que satisfaz a função objetivo de solução ótima. Construímos um algoritmo que trabalha em estágios, selecionando uma característica por vez e, em cada estágio, é tomada uma decisão considerando que uma característica particular é uma solução ótima, na esperança de que esta escolha leve até a solução ótima global. O algoritmo implementado consiste das etapas detalhadas na Tabela 2.

A Figura 4 mostra um exemplo da execução do algoritmo guloso, para o modelo de características apresentado na Figura 2, até que o orçamento disponível seja excedido. Apesar da complexidade deste algoritmo ser polinomial, nem sempre ele nos fornece o melhor resultado possível. Isso ocorre, por exemplo, durante a seleção de uma

característica com grau de interesse elevado para o cliente, mas que possui preço alto. A seleção desta característica pode impossibilitar a seleção de outras características. Neste caso, a combinação de duas ou mais características de preço menor poderia resultar em um produto de maior interesse para o cliente.

Tabela 2 Heurística gulosa

Algoritmo. Heurística Gulosa (Entrada: Modelo de Características em XML)

01. Pré-processamento do modelo de características
02. Verificar nível 2 da árvore de características
03. Se existir característica XOR
04. C_i = selecionar característica XOR com maior G_i
05. Se não
06. Se existir um grupo de características OR e nenhuma característica do grupo tenha sido selecionada
07. C_i = selecionar característica OR do grupo com maior G_i
08. Se não
09. C_i = selecionar característica com maior G_i
10. Se C_i é uma característica abstrata e todas as características filhas de C_i possuem $P > O$
11. Eliminar C_i e todas suas filhas
12. Se não
13. Se C_i é uma característica abstrata
14. C_i é configurada para o cliente
15. Se não
16. Se $P_i \leq O$
17. C_i é configurada para o cliente e P_i é decrementado do O
18. Se C_i compõe um grupo de características XOR
19. Eliminar as demais características XOR que pertencem ao mesmo grupo de C_i
20. Características filhas de C_i (se houver) passam a pertencer ao nível 2 da árvore
21. Se não
22. Eliminar C_i e todas suas filhas (se houver)
23. Executar a linha 02 até que todas as características da árvore sejam verificadas

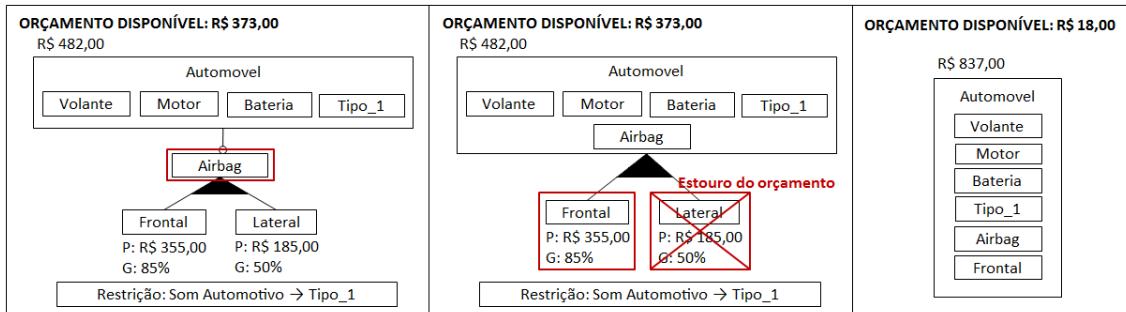


Figura 4 Exemplo da execução do algoritmo guloso

4. Experimentos Computacionais

Os experimentos foram realizados para 12 LPS que se diferenciam com relação ao número de características, classificação das características e restrições adicionais. Realizamos os experimentos agrupando as LPS em 4 grupos com 3 LPS por grupo. Variamos em 5, 10, 15 e 20 o número de características presentes nas LPS analisadas. Esta variação não excede 20 características, uma vez que a solução exata (por enumeração exaustiva) para grandes LPS é intratável devido a complexidade exponencial do problema. Os modelos de características foram obtidos do repositório SPLOT [8]. SPLOT é uma ferramenta Web de código fonte aberto que possui um repositório com centenas de modelos de características criados por usuários, além de modelos de características gerados automaticamente pela ferramenta. Os modelos de características na ferramenta podem ser exportados em formato XML e usados como entrada para os algoritmos implementados.

Verificamos o tempo médio utilizado por cada grupo durante a execução da enumeração exaustiva. A Figura 5a mostra que à medida que a quantidade de características aumenta, obtemos um crescimento exponencial do tempo de execução do algoritmo. Uma das características deste problema é que quanto maior o tamanho da expressão a ser solucionada, maior o tempo de processamento do algoritmo (Figura 5b). Neste caso o tempo independe da quantidade de produtos válidos gerados pela expressão. Isso ocorre porque o algoritmo de enumeração exaustiva testa todas as combinações possíveis, sejam elas válidas ou inválidas.

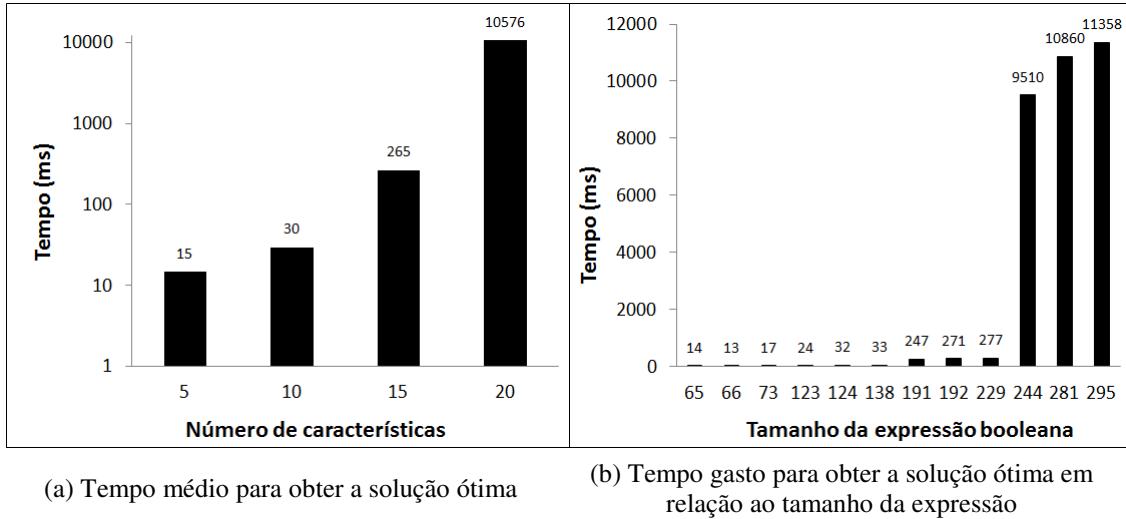


Figura 5 Experimentação por enumeração exaustiva

Realizamos os mesmos testes e avaliações para o algoritmo de enumeração exaustiva com pré-processamento e backtracking. O objetivo é observar o quanto este algoritmo é mais eficiente comparado ao algoritmo anterior, que implementa apenas o algoritmo de enumeração exaustiva sem estas otimizações. A Figura 6b mostra que com a implementação da técnica de pré-processamento, o tamanho da expressão correspondente a cada uma das LPS analisadas serão reduzidas. Por exemplo, o tamanho da expressão para uma mesma LPS passa de 65 caracteres para 9 caracteres (Figura 5b e 6b). Adicionalmente, com a implementação da técnica backtracking o tempo de execução será minimizado e boa parte das soluções que não levam a solução ótima serão eliminadas, ou seja, a diminuição do número de soluções possíveis facilita a resolução (Figura 6a). Por exemplo, o tempo médio de execução somente com o algoritmo de enumeração exaustiva para um modelo de características com 20 características é 10576 ms, enquanto que, para o mesmo algoritmo com a implementação das técnicas de pré-processamento e backtracking o tempo médio de execução é 56 ms. Vale a pena ressaltar, que a implementação destas técnicas garantem a obtenção da solução ótima.

Apesar dos experimentos computacionais com o algoritmo de enumeração exaustiva com pré-processamento e backtracking demonstrarem um ganho significativo, o crescimento ainda é exponencial em função do tamanho do modelo de características. Portanto, foi avaliada a implementação da heurística gulosa para resolução do problema. Este algoritmo possui crescimento polinomial de tempo em função do tamanho da entrada. O tempo de processamento utilizado pelo algoritmo, para um modelo com 20 características, é menor que 10 ms. A Figura 7 mostra a taxa de acertos do algoritmo guloso durante a configuração de 10 produtos em uma LPS com 20 características, a partir de parâmetros (orçamento e grau de importância das características) fornecidos por 10 clientes com pretensões distintas. A taxa de acerto do algoritmo foi calculada

através da execução do algoritmo de enumeração exaustiva, pois ele nos fornece a solução ótima para o problema. De 10 produtos gerados, somente 3 não fornecem como resultado a solução ótima, ou seja, 3 configurações são compostas por características que não fazem parte da solução ótima e/ou deixam de compor características que fazem parte da solução ótima. Assim, para os testes realizados, a taxa de acerto do algoritmo guloso é em média 92%. Entretanto, apesar de nem sempre o algoritmo guloso conduzir a soluções ótimas globais, o algoritmo encontra a solução ótima global na maioria dos casos. Isso ocorre devido ao padrão balanceado adotado pelos modelos de características em uma LPS, o que sugere a aplicabilidade deste algoritmo para resolução de problemas reais de grandes dimensões.

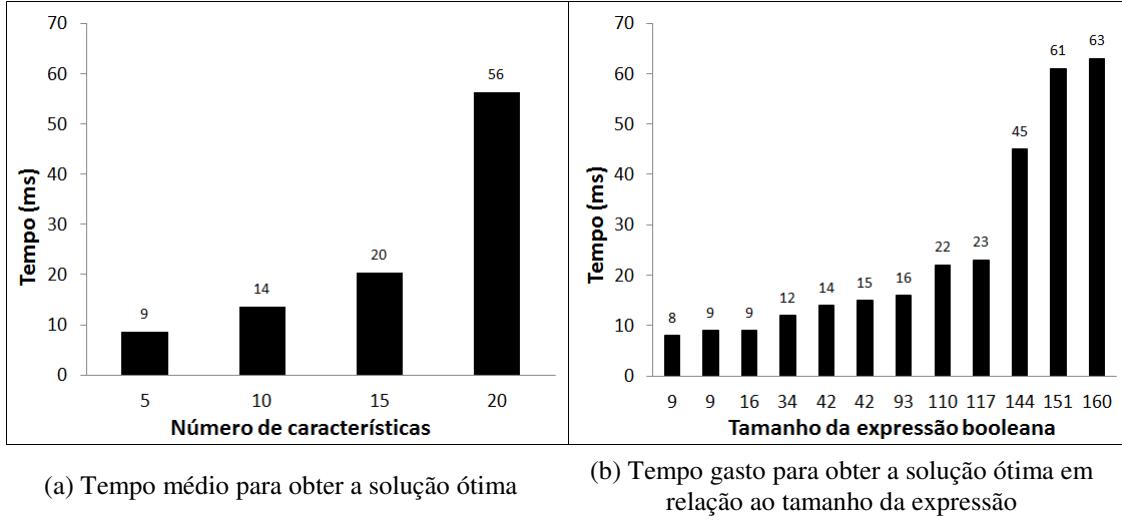


Figura 6 Experimentação por pré-processamento e backtracking

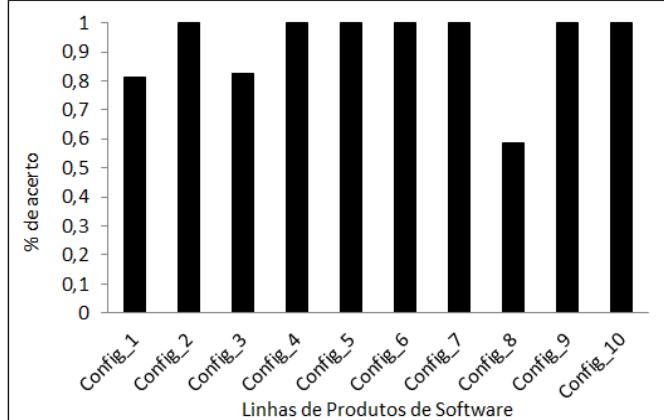


Figura 7 Taxa de acertos da heurística gulosa

5. Conclusão

Algoritmos de busca e otimização têm sido aplicados em diversas áreas da Engenharia de Software [6]. Neste trabalho, aplicamos estas técnicas para modelar o problema de configuração de produtos em uma LPS objetivando maximizar a satisfação do cliente. Dois tipos de algoritmos foram utilizados: enumeração exaustiva e solução gulosa. Os experimentos nos mostram que a enumeração exaustiva pode ser inviável, dado uma quantidade relativamente grande de características presentes em modelos reais. A utilização de algoritmos de pré-processamento e backtracking mostrou que o esforço necessário para encontrar a solução ótima tende a decrescer, devido ao número de caminhos que são eliminados, alcançando ganhos consideráveis em eficiência.

Por outro lado, mesmo utilizando pré-processamento e backtracking, o número de soluções viáveis para o problema continua a crescer exponencialmente em função do número de características. Assim, este trabalho propôs um algoritmo heurístico guloso. Este algoritmo se mostrou uma opção mais indicada para encontrar uma boa solução para o problema em significativamente menor tempo. A heurística proposta tem complexidade polinomial e apresenta uma taxa de 92% de acerto em relação à solução ótima, o que faz este modelo viável em aplicações práticas do problema.

Como trabalho futuro, pretende-se: (i) aplicar a heurística gulosa em LPS maiores, a fim de verificar o tempo necessário para execução, bem como a precisão dos resultados alcançados e (ii) considerar variáveis adicionais ao modelo proposto. Por exemplo, para maximizar os benefícios do ponto de vista da organização desenvolvedora, pretende-se incluir variáveis como a minimização do esforço de desenvolvimento e integração de características.

Agradecimentos

Este trabalho recebeu apoio financeiro da FAPEMIG, processos APQ-02376-11 e APQ-02532-12, e do CNPq processo 485235/2011-0.

Referências

- [1] Batory, D. (2005) “Feature Models, Grammars, and Propositional Formulas”, In Proc. of the International Software Product Line Conference (SPLC), pp. 7–20.
- [2] Batory, D., Sarvela, J., Rauschmayer. (2004) “Scaling Step-wise Refinement”, IEEE Transactions on Software Engineering, 30(6), pp. 355–371.
- [3] Benavides, D. et al. (2006) “A First Step Towards a Framework for the Automated Analysis of Feature Models”, In Managing Variability for SPL: Working With Variability Mechanisms.
- [4] Boucher, Q. et al. (2010) “Introducing TVL, a Text-based Feature Modelling Language”, In: Variability Modelling of Software-Intensive Systems (VaMoS).
- [5] Cormen, T. et al. (2009) “Introduction to algorithms”, Cambridge, EUA: Massachusetts Institute of Technology.
- [6] Harman, M. (2007) “The Current State and Future of Search Based Software Engineering”, In Proceedings of the Future of Soft. Eng. Workshop, Minneapolis.
- [7] Martello, C. and Toth, P. (1990) “Knapsack Problems: Algorithms and Computer Implementations”, John Wiley & Sons.
- [8] Mendonça, M. et al. (2009) “S.P.L.O.T. – Software Product Lines Online Tools”, Int’l Conf. on OO Programing, Systems, Languages, and Applications (OOPSLA).
- [9] Pipatsrisawat K. et al. (2008) “Solving Weighted Max-SAT Problems in a Reduced Search Space: A Performance Analysis”, Journal on Satisfiability Boolean Modeling and Computation.
- [10] Pohl, K.; Bockle, G.; Linden, F. J. van der. (2005) “Software Product Line Engineering: Foundations, Principles and Techniques”, Springer.
- [11] Product Line Hall of Fame. Disponível em: <http://splc.net/fame.html>
- [12] Sochos P. et al. (2004) “Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture”, OO and Internet-based Technology.
- [13] Schobbens, P. Y. et al. (2006) “Feature Diagrams: A Survey and a Formal Semantics, In International Requirements Engineering Conference (RE).
- [14] Sharp, D. C. (1998) “Reducing Avionics Software Cost Through Component Based Product Line Development”. Software Technology Conference.

Otimizando arquiteturas de LPS: uma proposta de operador de mutação para a aplicação automática de padrões de projeto

Giovani Guizzo¹, Thelma E. Colanzi^{1,2}, Silvia Regina Vergilio^{1*}

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

²DIn Universidade Estadual de Maringá (UEM) – Maringá, PR – Brasil

{gguizzo, thelmae, silvia}@inf.ufpr.br

Resumo. Padrões de projeto visam a melhorar o entendimento e o reúso de arquiteturas de software. No projeto baseado em busca eles têm sido aplicados com sucesso por meio de operadores de mutação no processo evolutivo. No contexto de Arquiteturas de Linha de Produtos (ALPs), alguns trabalhos têm aplicado padrões de projeto manualmente, mas não existe abordagem baseada em busca que considere o uso destes padrões. Para permitir este uso, este artigo introduz um operador de mutação para aplicação dos padrões de projeto GoF em uma abordagem de otimização de ALPs baseada em algoritmos multiobjetivos. O procedimento de mutação é descrito utilizando dois padrões: Strategy e Brigde, e um exemplo de aplicação em uma ALP real mostra bons resultados em termos de coesão e acoplamento.

Abstract. Design patterns aim at improving the understanding and reuse of software architectures. In the search based design they have been successfully applied with the use of mutation operators in the evolutionary process. In the context of Product Line Architectures (PLAs) some works have manually applied design patterns, but there is no search-based approach that considers the use of design patterns. To allow such use, this paper introduces a mutation operator procedure for application of the GoF design patterns in a PLA optimization approach based on multi-objective algorithms. The procedure is described by using two patterns Strategy and Brigde, and an example of application in a real PLA shows good results in terms of cohesion and coupling.

1. Introdução

Uma Linha de Produto de Software (LPS) é composta por um núcleo de artefatos que engloba características comuns e variáveis [Linden et al. 2007]. Características são atributos de um sistema de software que afetam diretamente os usuários finais e costumam ser representadas no modelo de características. Um produto de uma LPS é obtido pela combinação das suas características. A Arquitetura de LPS (ALP) contém o projeto que é comum a todos os produtos derivados da LPS [Linden et al. 2007]. O projeto da ALP inclui componentes para realizar todas as características comuns e variáveis da LPS. Então a ALP é o artefato da LPS que permite o reúso em larga escala. Na ALP, as variabilidades são representadas por pontos de variação e variantes. Pontos de variação são associados às variantes que representam alternativas de projeto.

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

Dentre as atividades da engenharia de LPS, o projeto da ALP é uma tarefa difícil que pode se beneficiar com o uso de algoritmos de otimização. Neste sentido, foi proposta uma abordagem baseada em algoritmos multiobjetivos para otimizar o projeto de ALPs de modo a reduzir o esforço do desenvolvimento de LPS [Colanzi 2012]. A abordagem produz um conjunto de soluções com bons resultados para diferentes objetivos, como por exemplo extensibilidade e modularidade de ALPs. O foco dessa abordagem é o projeto de ALP, representado no diagrama de classes da UML (*Unified Modeling Language*), uma vez que esse tipo de modelo é extensamente utilizado para modelar arquiteturas de software em um nível detalhado [Colanzi and Vergilio 2013, Ziadi et al. 2003].

Outros possíveis benefícios provêm da aplicação de padrões de projeto, como por exemplo os do catálogo GoF (*Gang of Four*) [Gamma et al. 1995]. Esses padrões incluem soluções comuns provenientes de diversos projetos e que são amplamente utilizadas entre desenvolvedores. O uso de padrões pode favorecer coesão e acoplamento, métricas diretamente ligadas à reusabilidade de software, inclusive para LPS. Portanto, a aplicação de padrões de projeto pode ajudar a obter uma ALP mais flexível, comprehensível e propícia a acomodar novas características durante a manutenção ou evolução da LPS. Apesar desses benefícios serem de grande valor no projeto de software, não foram encontrados trabalhos que aplicam padrões de projeto automaticamente na otimização de projeto de ALPs. Contudo, existem evidências de que operadores de mutação que aplicam padrões de projeto ou estilos arquiteturais na otimização de projeto de LPS podem contribuir para obter ALPs com mais qualidade [Colanzi and Vergilio 2012a, Räihä et al. 2011].

Considerando a importância e desafios para a aplicação automática de padrões no projeto de ALPs com abordagens baseadas em busca, bem como a complexidade para automatizar essa aplicação, este trabalho tem como objetivo discutir alguns fatores, tais como: a identificação automática de escopos propícios de aplicação de padrões e a definição de um operador de mutação. Neste trabalho, um escopo propício é uma parte de um diagrama de classes que possui elementos arquiteturais que satisfazem requisitos mínimos para a aplicação de um determinado padrão. Portanto, apesar de este trabalho utilizar algoritmos com abordagens automáticas, padrões de projeto só são aplicados em escopos propícios e quando existem reais vantagens de utilização para a melhoria da ALP [Gamma et al. 1995], caso contrário, anomalias de projeto podem ser geradas.

A principal contribuição deste trabalho é a proposta de um operador de mutação que aplica padrões de projeto automaticamente em ALPs a ser utilizado com algoritmos de otimização evolutivos, seguindo a abordagem baseada em busca para a otimização de ALPs apresentada em [Colanzi 2012]. Para tornar isto possível, é introduzida uma representação em forma de metamodelo para escopos propícios de aplicação de padrões de projeto em ALPs. Para exemplificar o uso do operador proposto, são descritos métodos de aplicação automática para dois padrões de projeto do catálogo GoF *Strategy* e *Bridge*. Ao final um exemplo de uso do operador e do padrão *Strategy* é apresentado. Este exemplo ilustra como o uso de padrões pode contribuir para melhorar a ALP de uma LPS.

O artigo está organizado como segue. A Seção 2 revê abordagem utilizada, baseada em algoritmos multiobjetivos para otimização de ALP. A Seção 3 introduz o operador de mutação e aspectos de implementação. A Seção 4 apresenta um exemplo de aplicação do padrão de projeto *Strategy* em uma ALP real. A Seção 5 descreve trabalhos relacionados. A Seção 6 conclui o artigo e apresenta trabalhos futuros.

2. Uma abordagem de otimização multiobjetivo para projeto de ALP

Abordagens baseadas em busca podem possibilitar a descoberta automática de bons projetos de ALP e tornam o projeto menos dependente de arquitetos de software. Nesse contexto, Colanzi (2012) propôs uma abordagem de otimização multiobjetivo que visa a: (a) alcançar ALPs o mais extensível possível; (b) reduzir esforço de manutenção e evolução; e (c) obter alta reusabilidade. Sendo assim, algoritmos evolutivos multiobjetivos parecem ser a melhor alternativa para o problema de otimização de ALP já que eles têm encontrado os melhores resultados quando há diferentes objetivos a serem alcançados [Coello et al. 2007]. Essa abordagem engloba quatro atividades, explicadas a seguir.

1. *Construção da Representação de ALP*: A partir do diagrama de classes contendo o projeto da ALP, uma representação computacional é gerada para a ALP contendo todos os elementos arquiteturais, com seus respectivos relacionamentos, variabilidades e interesses associados [Colanzi and Vergilio 2013].

2. *Definição do Modelo de Avaliação*: Há diferentes medidas que permitem avaliar uma ALP e nesta atividade o arquiteto seleciona as medidas que serão utilizadas no processo de otimização. Métricas convencionais, como coesão e acoplamento, podem ser utilizadas juntamente com métricas específicas para LPS, tais como extensibilidade e modularidade de LPS e métricas orientadas a características.

3. *Otimização Multiobjetivo*: A representação da ALP obtida na primeira atividade é otimizada de acordo com as restrições estabelecidas pelo arquiteto. Cada alternativa de ALP é avaliada seguindo o modelo de avaliação (segunda atividade). Um conjunto de representações de ALP é gerado como saída e ele consiste nas soluções que têm o melhor *trade-off* entre os objetivos (métricas utilizadas).

4. *Transformação e Seleção*: O conjunto de representações da ALP é convertido em uma visão legível para o arquiteto, que deve selecionar uma das alternativas para adotar como a ALP da LPS de acordo com as prioridades organizacionais.

O operador de mutação proposto neste trabalho é utilizado na atividade *Otimização Multiobjetivo* visando a melhorar a reusabilidade da ALP. Portanto, a aplicação de padrões proposta aqui deve ser automática, pois só assim ela será compatível com a abordagem de Colanzi (2012). A abordagem já contém outros operadores de mutação que têm por objetivo melhorar a modularização de características e a extensibilidade da ALP [Colanzi and Vergilio 2012b].

3. Proposta de aplicação automática de padrões na otimização de ALPs

Nesta seção é descrita uma proposta para aplicação automática de padrões de projeto em otimização de ALPs, mais particularmente no contexto da abordagem apresentada na última seção. Neste contexto, padrões de projeto são aplicados automaticamente por meio de um operador de mutação, apresentado na Seção 3.2. Para a definição do operador de mutação, alguns requisitos devem ser satisfeitos: i) o padrão deve ser aplicado em um escopo propício; ii) a aplicação do padrão deve ser coerente e não deve trazer nenhuma anomalia para a arquitetura (assim como assegurado em [Räihä et al. 2011]); iii) o padrão deve ser efetivamente aplicado em forma de mutação no processo evolutivo; e iv) o processo de identificação de escopos e aplicação do padrão deve ser totalmente automático, ou seja, aplicar o padrão sem nenhuma interferência do usuário.

De modo a satisfazer esses requisitos, antes de tudo, é necessário definir um modo de lidar com escopos propícios para a aplicação de cada padrão de projeto durante o processo evolutivo. Portanto, na Seção 3.1 é proposto um metamodelo genérico capaz de representar escopos de ALPs propícios para a aplicação de padrões, bem como os critérios para a identificação desses escopos. Tal modelo deve ser instanciado para um dado padrão. Os critérios são particularidades que moldam esses escopos e são implementados por métodos de verificação, descritos na Seção 3.3. Esses métodos verificam a adequação de um escopo em receber um determinado padrão de projeto.

Além disso, um método de aplicação *apply()* (Seção 3.3) é necessário para cada padrão de projeto com a função de aplicar o padrão em um escopo informado. Esse método adiciona um estereótipo referente ao padrão aplicado em todos os elementos arquiteturais mutados que fazem parte da estrutura do padrão. Isso é utilizado para prevenir que outras mutações deixem incoerente a estrutura do padrão aplicado.

3.1. Identificação de Escopos Propícios

Se um escopo de uma ALP é suscetível à aplicação de um determinado padrão de projeto, então ele é chamado de “*Pattern application Scope in Product Line Architecture*” (PS-PLA) para este padrão. A notação “PS-PLA<Nome do Padrão>” é utilizada para nomear um escopo propício para a aplicação de um padrão específico, por exemplo, PS-PLA<Strategy>. Isso implica que o escopo satisfaz todos os requisitos mínimos para a aplicação do padrão. Além disso, um escopo pode ser um PS-PLA para mais que um padrão de projeto, caso em que qualquer um destes padrões de projeto pode ser aplicado.

O metamodelo ilustrado na Figura 1 representa graficamente o conceito de PS-PLA. Um PS-PLA é um escopo composto por ao menos um elemento arquitetural, que por sua vez pode estar presente em vários PSs-PLA. De fato, para um escopo ser considerado um PS-PLA para um determinado padrão de projeto, ele deve satisfazer todos os requisitos que este padrão de projeto exige. Como já mencionado, esses requisitos são incorporados nos métodos de verificação específicos para cada padrão. Além disso, quando um padrão de projeto é aplicado ele pode influenciar algumas métricas arquiteturais que são usadas pelos algoritmos evolutivos para calcular o valor de *fitness* da solução.



Figura 1. Metamodelo representando o conceito de PS-PLA

3.2. Operador de Mutação Proposto

O Algoritmo 1 apresenta o operador de mutação proposto. Primeiramente, um padrão de projeto DP é aleatoriamente selecionado em um conjunto de padrões aplicáveis (linha 4). Depois disso, o operador de mutação utiliza a função $f_s(A)$ para obter um escopo S da arquitetura A (linha 5). Uma possível implementação para a função $f_s(A)$ é a seleção aleatória de um número aleatório de classes e interfaces de A . Assim como a seleção aleatória de padrões de projeto, selecionar aleatoriamente os elementos arquiteturais mantém

o aspecto aleatório da mutação no processo evolutivo. Entretanto, isso pode ser redefinido pelo usuário, sendo possível a criação de regras para a seleção de escopos e padrões.

Algoritmo 1: Pseudocódigo do operador de mutação proposto

```

1 Entrada:  $A$  - Arquitetura a ser mutada;  $\rho_{mutation}$  - Probabilidade de mutação.
2 Saída: a arquitetura mutada  $A$ .
3 Início
4    $DP \leftarrow$  seleção aleatória de um padrão de projeto em um conjunto de padrões aplicáveis;
5    $S \leftarrow f_s(A)$ ;
6   se  $DP.verification(S)$  e  $\rho_{mutation}$  for alcançada então
7     |    $DP.apply(S)$ ;
8   fim se
9   retorna  $A$ ;
10 Fim
```

Depois disso, usando os métodos de verificação, o escopo S é verificado como um PS-PLA para a aplicação do padrão de projeto DP (linha 6, método $verification()$). Se o método de verificação retornar *verdadeiro* e a probabilidade de mutação $\rho_{mutation}$ for alcançada, então o método $apply()$ é utilizado para aplicar o padrão de projeto DP no escopo S (linha 7). Os métodos $verification()$ e $apply()$ possuem implementação variável, específica para o tipo de padrão selecionado. A seguir são descritas implementações destes métodos. Por restrições de espaço, somente dois padrões do catálogo GoF são utilizados como exemplo: *Bridge* e *Strategy*. Uma descrição resumida destes padrões é apresentada na Tabela 1 de acordo com [Gamma et al. 1995].

Tabela 1. Padrões de projeto utilizados

Padrão	Descrição
Bridge	Padrão estrutural que separa uma abstração da sua implementação, de modo que as duas possam variar independentemente.
Strategy	Padrão comportamental que define uma família de algoritmos, encapsulando-os e tornando-os intercambiáveis. Assim o algoritmo pode variar independentemente de seus clientes.

3.3. Métodos de Verificação de Escopo e Aplicação de Padrões

Esta seção descreve os métodos de verificação e de aplicação dos padrões *Bridge* e *Strategy*, juntamente com exemplos genéricos. Cada método de verificação recebe um escopo S como parâmetro e fazendo diversas verificações em seus elementos. Assim, cada método de verificação verifica se o escopo S é propício para a aplicação do padrão de projeto DP . Se sim, o método $apply()$ de aplicação do padrão DP pode aplicá-lo em S . O método $apply()$ é o método que faz a mutação propriamente dita, uma vez que ele aplica o padrão no escopo mudando, removendo e/ou adicionando elementos arquiteturais.

É importante mencionar alguns aspectos de implementação. Primeiramente, foi utilizado o conceito de interesse (*concern*) em alguns métodos de verificação e aplicação para identificar características relacionadas. Na engenharia de LPS, interesses são utilizados na implementação de características, sendo que uma característica pode ser considerada um interesse a ser realizado. Neste trabalho, interesses são mapeados em elementos de um diagrama de classes por meio do uso de estereótipos. Por motivos de espaço, as operações de adição de estereótipos de cada padrão estão implícitas na descrição dos métodos, e atributos, métodos e tipos de retornos foram omitidos nos diagramas

de exemplo. Por fim, a notação PLUS [Gomaa 2004] foi utilizada para a representação de variantes em diagramas UML.

O método de verificação (*verification()*) do *Strategy* verifica se: i) existe um conjunto de elementos arquiteturais que fazem parte de uma família de algoritmos; ii) os elementos do item i) são todos utilizados por pelo menos um elemento em comum (que possui o papel de *context*); iii) o elemento *context* utiliza os elementos da família de algoritmos com um mesmo tipo de relacionamento de uso; iv) existe apenas uma família de algoritmos no escopo; v) o elemento *context* é um ponto de variação; e vi) os algoritmos que *context* utiliza são variantes. No contexto deste trabalho, uma família de algoritmos pode ser caracterizada se ela possuir: i) ao menos dois elementos arquiteturais com um prefixo ou sufixo comum em seus nomes; ii) pelo menos dois elementos arquiteturais com ao menos dois métodos com um mesmo sufixo ou prefixo em seus nomes e um mesmo tipo de retorno. Apesar de uma família de algoritmos ser caracterizada dessa maneira neste trabalho, essa definição depende da interpretação, experiência e compreensão de projeto do projetista.

A Figura 2 apresenta um exemplo de mutação utilizando o método *apply()* do padrão *Strategy*. A Figura 2.a ilustra um exemplo de PS-PLA<Strategy> antes da aplicação do padrão. Apesar de o elemento “Cliente” ser um ponto de variação isso não é visível no diagrama porque a notação PLUS não representa pontos de variação explicitamente em diagramas de classes. O elemento “Client” no escopo possui o papel de *context* e as classes “AlgorithmA” e “AlgorithmB” são algoritmos de uma família de algoritmos. Como resultado da mutação (Figura 2.b), a interface nomeada “AlgorithmStrategy” é criada, pois ainda não existe uma interface para essa família. Essa interface define a família de algoritmos, portanto declara todos os métodos de ambos os algoritmos. Uma vez que as classes “AlgorithmA” e “AlgorithmB” implementam a interface *Strategy*, o elemento “Client” não precisa mais utilizá-las diretamente, basta utilizar a interface “AlgorithmStrategy” com o mesmo tipo de relacionamento. Apesar do relacionamento entre *context* e a interface *Strategy* ser de agregação na estrutura padrão do *Strategy* [Gamma et al. 1995], o relacionamento é mantido o mesmo de antes da mutação para preservar o comportamento dos elementos. Além disso, todas as classes e interfaces que utilizavam diretamente os algoritmos da família, passam a utilizar a interface *Strategy* depois da mutação.

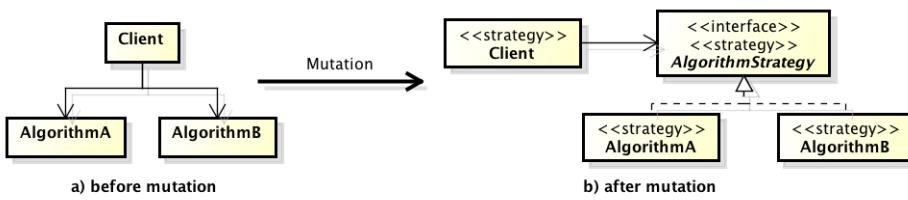


Figura 2. Exemplo de mutação utilizando o padrão *Strategy*

O método de verificação do *Bridge* checa se: i) existe ao menos um conjunto de elementos arquiteturais que fazem parte de uma família de algoritmos; ii) os elementos do item i) são todos utilizados por pelo menos um elemento em comum (que possui o papel de *context*); iii) o elemento *context* utiliza os elementos da família de algoritmos com um mesmo tipo de relacionamento de uso; iv) ao menos dois elementos de uma mesma família de algoritmos do escopo possuem ao menos um interesse associado; v) o elemento *context* é um ponto de variação; e vi) os algoritmos que *context* utiliza são variantes.

A Figura 3 apresenta um exemplo genérico de mutação para o padrão *Bridge*. De fato, a Figura 3.a apresenta um PS-PLA<Bridge> conforme descrito anteriormente. O elemento “Client” possui o papel de *context* e as classes “AlgorithmA” e “AlgorithmB” são algoritmos de uma família de algoritmos com um interesse *x* associado a eles. Por realizarem o mesmo interesse e fazerem parte de uma mesma família de algoritmos, essas classes implementam operações similares, o que possibilita a abstração de ambas. A mutação cria uma classe abstrata “XAbstraction” (papel de *abstraction*), uma classe concreta padrão “ConcreteXAbstraction” estendendo a classe *abstraction*, e uma interface “XImplementation” (papel de *implementation*) que agora define os algoritmos da família de algoritmos. Para cada interesse em comum entre os algoritmos da família de algoritmos, uma interface de implementação é criada. Além disso, todos os métodos similares dos algoritmos são declarados na interface *implementation* e na classe abstrata *abstraction*. O elemento *context* e todos os elementos que antes utilizavam os algoritmos diretamente, passam a utilizar a classe *abstraction* depois da mutação.

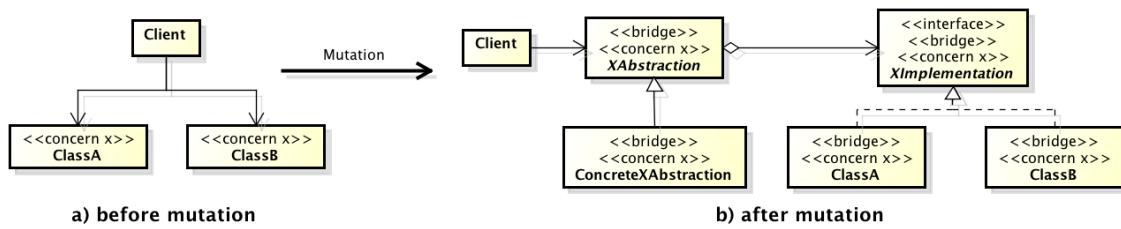


Figura 3. Exemplo de mutação usando o padrão *Bridge*

Implementar esses dois padrões no contexto de pontos de variação permite um fácil manuseio de variantes. Por exemplo, para remover uma variante de um produto em um ponto de variação que possui a implementação do *Bridge*, basta que o projetista remova da estrutura a classe referente à variante. Os elementos que utilizam essa variante não sentirão essa mudança, pois acessam tais funcionalidades por meio da classe *abstraction*.

4. Exemplo de Uso do Operador Proposto

De modo a apresentar a aplicabilidade do operador de mutação proposto, essa seção contém um exemplo de aplicação com o uso da ALP *Microwave Oven Software* [Gomaa 2004]. As seguintes entradas de dados são informadas: A = a ALP *Microwave Oven Software*, e $\rho_{mutation} = 1$.

Primeiramente um padrão deve ser selecionado aleatoriamente. Supõe-se que o *Strategy* foi selecionado. O próximo passo consiste em selecionar um escopo S de A utilizando a função $f_s(A)$. A Figura 4 ilustra os elementos arquiteturais aleatoriamente selecionados pela função $f_s(A)$ e armazenados no escopo S .

Depois do escopo ter sido selecionado, ele é verificado como sendo um PS-PLA<Strategy> ou não, de acordo com o método de verificação apresentado na Seção 3.3. A família de algoritmos é composta pelos elementos “Lamp”, “Turntable”, “Display”, “HeatingElement” e “Beeper”. Eles são caracterizados como uma família porque possuem métodos em comum (*on()* e *off()*). O elemento com o papel de *context* é a classe “MicrowaveOvenProductLineSystem” que utiliza os elementos da família de algoritmos diretamente com um mesmo tipo de relacionamento. Além disso, essa classe é um ponto de variação, apesar de não estar explícito na imagem. Neste exemplo, os estereótipos

PLUS «optional» e «kernel» são utilizados para representar variantes e partes comuns, respectivamente. Portanto, o método *Strategy.verification(S)* retorna *verdadeiro*, uma vez que o escopo *S* é de fato um PS-PLA<Strategy>.

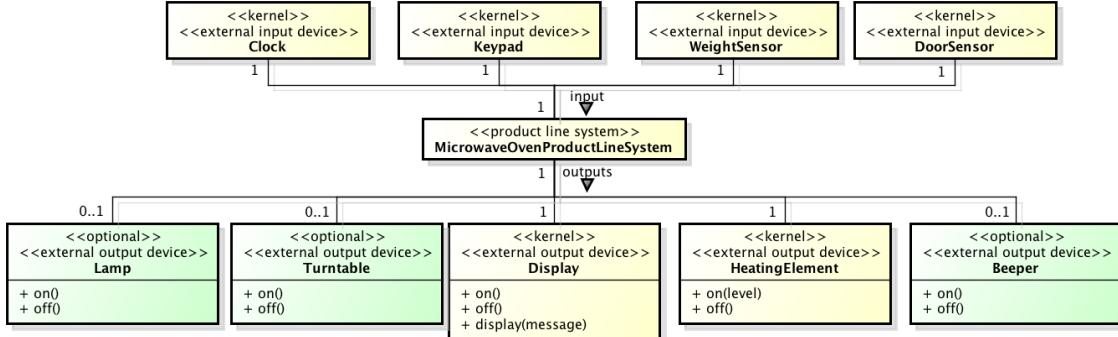


Figura 4. Escopo selecionado para a mutação pela função $f_s(A)$

Após verificado, o escopo recebe a aplicação do *Strategy* por meio da execução do método *Strategy.apply(S)*, cujo resultado é apresentado na Figura 5. O padrão *Strategy* é apropriado para essa estrutura, pois ele pode criar uma abstração para as cinco classes concretas de *output* e desacoplar delas a classe “MicrowaveOvenProductLineSystem”.

Neste exemplo, a interface “OutputStrategy” foi criada para definir a estrutura dos algoritmos. Ela possui o papel principal: de interface *Strategy* e torna-se o ponto de variação depois da mutação; é implementada pelas classes “Lamp”, “Turntable”, “Display”, “HeatingElement” e “Beeper”, e consequentemente, todas essas classes implementam todos os métodos da interface. Apesar da classe “HeatingElement” ter um parâmetro para o método *on()* e as outras não, esse método é considerado o mesmo das outras, pois seu retorno e nome são idênticos para todas as classes. Algumas das classes não utilizarão todos os métodos, como por exemplo *display(message)*. Como descrito em [Gamma et al. 1995], essa é uma situação normal para este padrão, uma vez que a interface *Strategy* deve definir todos os métodos de todas as classes da família de algoritmos, independentemente da complexidade destas classes. Se uma classe não utiliza um método ou um parâmetro de um método, ela deve apenas ignorar os itens não utilizados.

O operador de mutação retorna a arquitetura *A* da Figura 5. Por adicionar a interface “OutputStrategy” e forçar “MicrowaveOvenProductLineSystem” a usá-la ao invés das cinco classes concretas, o projeto alcança um menor nível de acoplamento entre *con-*

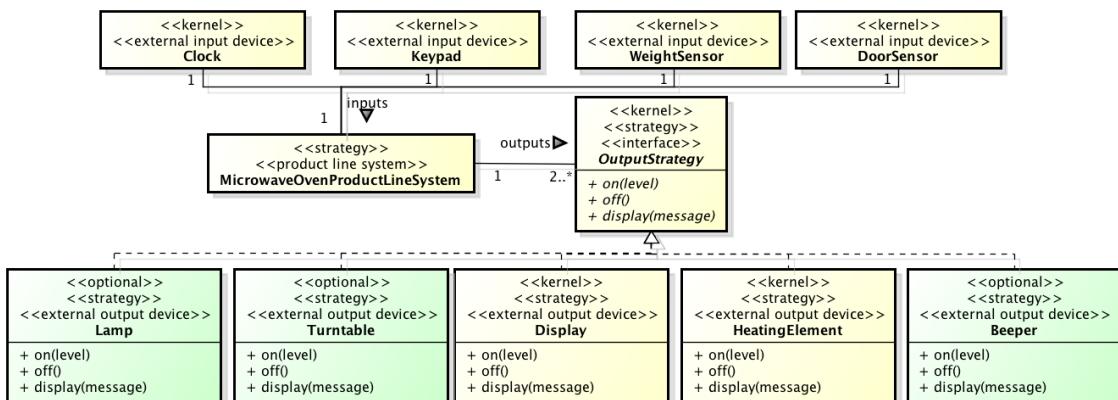


Figura 5. Escopo mutado com a aplicação do padrão de projeto *Strategy*

text e os elementos da família de algoritmos, e um maior nível de coesão para a classe “MicrowaveOvenProductLineSystem”. Portanto, a estrutura alcançada pode ser reutilizada e mantida de uma forma mais fácil. Por exemplo, adicionar uma variante à ALP agora requer apenas a criação da classe e a realização da interface. Isso diminui o esforço do projetista em modificar variabilidades.

5. Trabalhos Relacionados

Existem alguns trabalhos nos quais algoritmos de busca têm sido usados para aplicar padrões no projeto de software. Uma abordagem semi-automática para aplicar padrões de projeto do catálogo GoF em sistemas utilizando refatoração de código é proposta em [Cinnéide and Nixon 2001]. Entretanto, essa abordagem delega ao projetista a decisão de quais padrões aplicar e onde aplicá-los, focando apenas em remover do projetista o processo tedioso e propenso a erros de reorganização de código.

Räihä *et al.* (2011) aplicam os padrões *Façade*, *Adapter*, *Strategy*, *Template Method* e *Mediator* na síntese de arquiteturas de software com algoritmos genéticos. Os padrões são aplicados por operadores de mutação a fim de construir soluções mais facilmente modificáveis e eficientes durante o processo evolutivo. Apesar da proposta dos autores garantir uma aplicação automática coerente e válida, isso não significa necessariamente que os padrões estão sendo aplicados em escopos propícios. Além disso, a representação da arquitetura adotada não comporta características específicas de ALPs.

Para aplicar padrões em LPS, Keepence e Mannion (1999) desenvolveram um método que usa padrões de projeto para modelar variabilidades em modelos orientado a objetos. Eles definiram três padrões para modelar grupos de variantes mutuamente exclusivas, grupos de variantes inclusivas e variantes opcionais. Entretanto, não é possível identificar automaticamente escopos para a aplicação destes padrões usando apenas diagrama de classes. Além disso, alguns padrões do catálogo GoF oferecem os mesmos tipos de soluções em um nível mais abstrato, como por exemplo o *Strategy* [Gamma *et al.* 1995].

Ziadi *et al.* (2003) propõem uma abordagem baseada no uso do padrão *Abstract Factory* para derivar modelos de produtos a partir de ALPs modeladas com UML. A abordagem inclui a especificação de uso do padrão e um algoritmo usando OCL (*Object Constraint Language*) para derivar os modelos. Apesar de ter sido comprovada como uma boa solução para modelar variabilidades, não é possível identificar, através desta abordagem, escopos propícios para a aplicação automática do padrão.

Ainda que estes trabalhos envolvam a aplicação de padrões de projeto em ALPs, eles não se preocupam em identificar escopos propícios à aplicação automática de padrões de projeto. Adicionalmente, até então, não há na literatura um estudo sobre a aplicação totalmente automática de padrões de projeto para aprimorar o projeto de ALPs, principalmente considerando o projeto de LPS baseado em busca.

6. Conclusão

Este trabalho apresentou um operador de mutação para a aplicação automática de padrões de projeto do catálogo GoF em LPS, a ser utilizado em uma abordagem de projeto baseada em busca. Os padrões são aplicados apenas quando um escopo da arquitetura é propício para a sua aplicação. Deste modo, evita-se a introdução de anomalias de projeto e mantém-se a integridade da arquitetura.

O operador de mutação proposto aplica aleatoriamente um padrão de projeto em um escopo proveniente da arquitetura. Métodos de verificação e aplicação para dois padrões foram descritos. No exemplo de aplicação, a introdução de padrões de projeto propiciou a obtenção de uma ALP mais extensível e comprehensível e essas melhorias puderam ser detectadas por algumas métricas de software, como a coesão e o acoplamento.

Neste sentido, pretende-se automatizar a aplicação do operador de mutação para viabilizar a realização de estudos empíricos que permitam analisar se esses benefícios são válidos para ALPs reais. Além disso, pretende-se realizar uma análise da viabilidade da utilização de todos os padrões do catálogo GoF. Para os que se mostrarem viáveis, deverão ser definidos os métodos *verification()* e *apply()*.

Referências

- Cinnéide, M. and Nixon, P. (2001). Automated software evolution towards design patterns. *International Workshop on Principles of Software Evolution*, pages 162–165.
- Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer-Verlag New York, USA, 2 edition.
- Colanzi, T. E. (2012). Search based design of software product lines architectures. In *International Conference on Software Engineering (ICSE), Doctoral Symposium*, pages 1507–1510.
- Colanzi, T. E. and Vergilio, S. R. (2012a). Applying Search Based Optimization to Software Product Line Architectures: Lessons Learned. In *International Symposium on Search Based Software Engineering*, pages 259–266, Riva del Garda.
- Colanzi, T. E. and Vergilio, S. R. (2012b). Direções de Pesquisa para a Otimização de Arquiteturas de Linhas de Produto de Software Baseada em Busca. In *III Workshop de Engenharia de Software Baseada em Busca*.
- Colanzi, T. E. and Vergilio, S. R. (2013). Representation of Software Product Lines Architectures for Search-based Design. In *I Workshop of International Conference on Software Engineering (CMSBSE) - ICSE'2013*.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.
- Gomaa, H. (2004). Designing Software Product Lines with UML. In *Clinical trials London England*, number 6(8). Addison-Wesley Professional, Boston, MA.
- Keepence, B. and Mannion, M. (1999). Using patterns to model variability in product families. *IEEE Software*, (August):102–108.
- Linden, F. v. d., Schmid, F., and Rommes, E. (2007). *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer.
- Räihä, O., Koskimies, K., and Mäkinen, E. (2011). Generating software architecture spectrum with multi-objective genetic algorithms. In *2011 Third World Congress on Nature and Biologically Inspired Computing (NaBIC)*, pages 29–36. IEEE.
- Ziadi, T., Jézéquel, J., and Fondement, F. (2003). Product Line Derivation with UML. In *Groningen Workshop on Software Variability Management*.

A caminho de uma abordagem baseada em buscas para minimização de conflitos de *merge*

Gleiph Ghiotto¹, Leonardo Murta¹, Marcio Barros²

¹Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passos da Pátria 158, São Domingos – Niterói – RJ – Brasil

²Programa de Pós-graduação em Sistemas de Informação – UNIRIO
Av. Pasteur 458, Urca – Rio de Janeiro – RJ – Brasil

{gmenezes, leomurta}@ic.uff.br, marcio.barros@uniriotec.br

Abstract. Merge algorithms are widely used to conciliate changes performed in parallel over software artifacts. However, traditional 3-way merge algorithms, based on diff3, are unable to solve some conflicts because they only consider two basic operations: add and remove. These operations cannot accurately describe developer's intention. Moreover, these algorithms ignore intermediate versions, using only the two head versions and a common base version to perform the merge. This paper introduces a search-based approach that aims at minimizing the number of conflicts generated during merge. Our approach considers the developer's intention and the development history during merge, finding alternative merge sequences to reduce conflicts.

Resumo. Algoritmos de merge são amplamente utilizados para conciliar alterações realizadas em paralelo sobre artefatos de software. Entretanto, algoritmos tradicionais de merge em três vias, baseados no diff3, não são capazes de solucionar alguns conflitos por considerar apenas duas operações básicas: adição e remoção. Essas operações não expressam precisamente a intenção do desenvolvedor. Além disso, esses algoritmos ignoram versões intermediárias, usando somente as duas versões cabeça e a versão base na execução do merge. Este artigo introduz uma abordagem baseada em buscas que visa minimizar o número de conflitos gerados pelo merge físico. Para tal, a intenção do desenvolvedor e o histórico de desenvolvimento são considerados na execução de merge, encontrando sequências alternativas de merge que reduzem os conflitos.

1. Introdução

Algoritmos de merge são amplamente utilizados para conciliar alterações realizadas em paralelo sobre artefatos de software. Segundo Mens (2002), o suporte para merge de software é necessário durante todo o ciclo de desenvolvimento. Essa necessidade surge em especial no desenvolvimento de software complexo em larga escala, onde pode haver um grande número de desenvolvedores distribuídos geograficamente. Este cenário motiva a criação de diversas linhas de desenvolvimento que são frequentemente mescladas, visando consolidar alterações realizadas em paralelo.

A importância dos algoritmos de merge se tornou ainda maior com a popularização de sistemas de controle de versão (SCV) distribuídos, como, por exemplo, o Git [Chacon 2009] e o Mercurial [O'Sullivan 2009]. Nesses sistemas, não há

opção da utilização de políticas de controle de concorrência pessimistas, baseadas em bloqueio. Sendo assim, cada desenvolvedor clona para o seu espaço de trabalho uma cópia completa do repositório. Durante seu trabalho, são realizados *commits* locais, que compõem um ramo em relação aos demais desenvolvedores. Ao final, as contribuições locais são reintegradas ao repositório de origem ou a outros repositórios por meio de comandos de *push* e *pull*, requerendo *merges* frequentes.

Quando o algoritmo de *merge* não consegue conciliar automaticamente as contribuições feitas em paralelo, são reportados conflitos para serem tratados manualmente pelos desenvolvedores. Desta forma, o número de conflitos reportados e a complexidade desses conflitos se tornam parâmetros importantes para classificar a qualidade de um algoritmo de *merge*. Por exemplo, os algoritmos de *merge* de duas vias (do inglês, *two-way merge*) consideram somente as duas versões mais recentes de cada linha de desenvolvimento na execução do *merge*. Como consequência, um elevado número de conflitos é gerado por não ser possível diferenciar a intenção de adição por parte de um desenvolvedor da intenção de remoção por parte do outro. Por outro lado, algoritmos de *merge* de três vias (do inglês, *three-way merge*) também levam em conta uma versão base comum aos ramos que devem ser combinados, permitindo uma melhor percepção da intenção dos desenvolvedores, levando a um menor número de conflitos.

Os SCV atuais fazem amplo uso de algoritmos de *merge* de três vias, pois, por guardarem o histórico da evolução do software, sempre têm a versão base disponível. Contudo, esses algoritmos não utilizam todo o histórico de desenvolvimento do projeto e, consequentemente, o conhecimento presente nas versões intermediárias, entre a base e as cabeças dos ramos, é perdido. Essa decisão leva a menos alternativas no momento de resolver automaticamente um conflito. Além disso, os algoritmos de *merge* tradicionais mapeiam as alterações em operações de adição e remoção. Essa estratégia restringe a capacidade de identificação precisa da real intenção dos desenvolvedores.

Diante dessas limitações, é proposta uma abordagem baseada em buscas que tem como objetivo minimizar o número de conflitos gerados pela execução do *merge* físico. Esta abordagem, denominada Kraken, se diferencia das demais por utilizar, na execução do processo de *merge*, o histórico de versões intermediárias de cada ramo, heurísticas para a identificação da real intenção do desenvolvedor e técnicas de busca para identificação de diferentes sequências de aplicação de deltas visando a minimização do número de linhas de código em conflito. Por considerar todo o histórico de versões intermediárias entre a versão base e as cabeças dos ramos, essa abordagem tende a explorar um espaço de busca de ordem fatorial ao número de deltas, representando um cenário propício para aplicação de Engenharia de Software Baseada em Buscas [Harman 2007].

Este trabalho está organizado em seis seções incluindo esta seção de introdução. Na Seção 2 é apresentado um exemplo motivacional que consiste em um cenário de *merge* onde as abordagens tradicionais não são capazes de obter automaticamente uma solução livre de conflitos. Na Seção 3 é dada uma visão geral do estado da arte em *merge* de software e técnicas de busca que podem ser aplicadas para auxiliar na minimização dos conflitos de *merge*. Na Seção 4 é apresentada uma visão geral da abordagem Kraken, caracterizando-a como um problema de otimização, e uma discussão das atividades realizadas por ela para minimizar o número de conflitos. Na

Seção 5 é apresentado como a Kraken soluciona o exemplo motivacional da Seção 2. Finalmente, na Seção 6 são apresentadas as conclusões e discutidos trabalhos futuros.

2. Exemplo motivacional

Conforme discutido na Seção 1, os algoritmos de *merge* tradicionais não são capazes de solucionar, sem conflitos, alguns cenários de *merge* que poderiam ser resolvidos automaticamente. Nesta seção é apresentado um exemplo que reforça a necessidade de uma nova abordagem de *merge*.

Considere a classe `Transformacao`, apresentada na Figura 1, que possui três métodos: um método `celsiusToKelvin`, que está implementado na classe, e outros dois métodos que estão sem conteúdo. Tais métodos foram projetados para realizar transformação de temperatura entre as escalas Kelvin, Fahrenheit e Celsius.

```
class Transformacao {
    public double kelvinToFahrenheit(double kelvin) {
    }
    public double celsiusToKelvin(double celsius) {
        return celsius + 273;
    }
    public double fahrenheitToCelsius(double fahrenheit) {
    }
}
```

Figura 1. Versão base das alterações.

Considere também que, a partir desse artefato mostrado na Figura 1, dois desenvolvedores realizaram alterações em seus ramos. O primeiro desenvolvedor, no ramo 1, realizou duas modificações, como apresentado na Figura 2: Δ_1 , que consiste em mover o método `fahrenheitToCelsius` para antes do método `celsiusToKelvin`, e Δ_2 , que consiste em adicionar conteúdo ao método `kelvinToFahrenheit`. Em paralelo, o desenvolvedor 2, no ramo 2, realiza duas modificações, como apresentado na Figura 3: Δ_3 , que adiciona conteúdo no método `fahrenheitToCelsius`, e Δ_4 , que move o método `kelvinToFahrenheit` para depois do método `celsiusToKelvin`.

```
class Transformacao {
    public double kelvinToFahrenheit(double kelvin) {
        return (9 * (kelvin - 273) / 5) + 32;           ( $\Delta_2$ )
    }
    public double fahrenheitToCelsius(double fahrenheit){ ( $\Delta_1$ )
    }                                                 ( $\Delta_1$ )
    public double celsiusToKelvin(double celsius) {
        return celsius + 273;
    }
    public double fahrenheitToCelsius(double fahrenheit) { ( $\Delta_1$ )
    }                                                 ( $\Delta_1$ )
}
```

Figura 2. Versão com alterações do Desenvolvedor 1 (ramo 1).

Analizando a Figura 2 e a Figura 3 é possível visualizar que as alterações realizadas sobre o artefato original resultam em conflitos. Por exemplo, a remoção da linha com o conteúdo “`}`” pelo Δ_4 e a adição da linha com o conteúdo “`return (9 * (kelvin - 273) / 5) + 32;`” pelo Δ_2 representa um conflito porque ambos os desenvolvedores editaram a mesma região de um artefato. Este cenário de conflitos pode ser visualizado na Figura 4, onde o Git foi utilizado para realizar *merge* entre os

ramos 1 e 2. Neste cenário é demarcada a versão do *ramo1* e do *ramo2*, pois o algoritmo de *merge* do Git não foi capaz de resolver os conflitos existentes. Deste modo, fica a cargo do desenvolvedor combinar as versões.

```
class Transformacao {
    public double kelvinToFahrenheit(double kelvin) {          ( $\Delta_4$ )
    +
    public double celsiusToKelvin(double celsius) {           ( $\Delta_4$ )
        return celsius + 273;
    }
    public double kelvinToFahrenheit(double kelvin) {          ( $\Delta_4$ )
    }
    public double fahrenheitToCelsius(double fahrenheit) {   ( $\Delta_4$ )
        return (5 * (fahrenheit - 32)) / 9;                   ( $\Delta_3$ )
    }
}
```

Figura 3. Versão com alterações do Desenvolvedor 2 (ramo 2).

```
class Transformacao {
<<<<< ramo1
    public double kelvinToFahrenheit(double kelvin) {
        return (9 * (kelvin - 273) / 5) + 32;
    }
    public double fahrenheitToCelsius(double fahrenheit) {
    }
    public double celsiusToKelvin(double celsius) {
        return celsius + 273;
    }
=====
    public double celsiusToKelvin(double celsius) {
        return celsius + 273;
    }
    public double kelvinToFahrenheit(double kelvin) {
    }
    public double fahrenheitToCelsius(double fahrenheit) {
        return (5 * (fahrenheit - 32)) / 9;
    }
>>>>> ramo2
}
```

Figura 4. Conflito resultante do *merge* dos ramos 1 e 2.

3. Trabalhos relacionados

A resolução de conflitos é tratada por diferentes áreas da computação, como, por exemplo, banco de dados e engenharia de software. Na área de banco de dados, a resolução de conflitos é necessária para combinar transações realizadas em paralelo sobre réplicas de um banco de dados central [Berlage e Genau 1993, Edwards et al. 1997, Kermarrec et al. 2001]. Para realizar essas combinações são exploradas satisfatoriamente diversas estratégias, como, por exemplo, intercalação da ordem das transações, de forma que a combinação das transações seja realizada.

Na área de engenharia de software, foco deste trabalho, a resolução de conflitos é uma preocupação constante em *merge* de software, onde os desenvolvedores realizam alterações sobre artefatos de software que devem ser combinados no futuro. Pesquisas em *merge* de software são realizadas sobre diversos tipos de artefatos, como, por exemplo, modelos [Koegel et al. 2010, Murta et al. 2008], arquivos binários [Silva Junior et al. 2012] e arquivos texto [Mens 2002, Shen e Sun 2004]. O escopo deste artigo é voltado para resolução de conflitos em *merges* textuais, independentemente da estrutura sintática utilizada. Esse tipo de *merge* é de especial relevância por ser

genérico, podendo estar incorporado em SCV e apoiar projetos de software independentemente da linguagem adotada. Portanto, essa seção trata desse tipo de *merge*.

Como discutido anteriormente, os SCV atuais empregam de forma extensiva o *merge* textual de duas e três vias. O *merge* de duas vias [Mens 2002] é capaz de reconhecer que dois artefatos são diferentes, extrair o delta que representa essa diferença e apoiar na combinação dos dois artefatos. Contudo, nenhum conhecimento a mais pode ser extraído devido à ausência de um referencial sobre a forma que ocorreu a evolução. Para solucionar esse problema, o *merge* de três vias [Mens 2002] utiliza um ancestral comum às versões que serão combinadas. Neste caso, o ancestral comum serve como parâmetro para que as operações de adição ou remoção sejam mapeadas. Por exemplo, supondo que uma linha de código exista no ancestral e em uma das versões, mas não exista mais na outra, então esta linha é marcada como removida. Por outro lado, a adição pode ser identificada por uma linha de código que não exista no ancestral e também não exista em uma das versões, mas passou a existir na outra versão.

Com o intuito de melhorar as abordagens de *merge* tradicionais Shen e Sun (2004) desenvolveram uma abordagem baseada em operações que realiza o *merge* de dois ramos. Para tal, é utilizada uma correção no local que haveria conflito, permitindo a combinação de ambas as alterações em diferentes ordens. Apesar dessa abordagem ser capaz de conciliar alguns conflitos que são reportados por abordagens de *merge* de três vias, ela pode gerar falsos negativos em algumas situações. Esses falsos negativos, que ocorrem quando conflitos reais não são reportados, são causados quando as edições em paralelo têm o mesmo propósito. Por exemplo, caso haja na versão base uma expressão “ $a = b;$ ”, que foi editada em paralelo para “ $a = 1 + b;$ ” e para “ $a = b + 1;$ ”, nenhum conflito seria reportado e o resultado final seria “ $a = 1 + b + 1;$ ”, o que provavelmente é equivocado. Além disso, por não capturar a real intenção dos desenvolvedores durante a evolução, esta abordagem não seria capaz de resolver conflitos como o mostrado na Seção 2.

Diante do exposto, é possível identificar que o histórico de desenvolvimento dos projetos e a intenção do desenvolvedor são recorrentemente ignorados na literatura. Ao descartar o histórico de desenvolvimento, perde-se o conhecimento sobre as pequenas mudanças que, quando combinadas, geram a alteração como um todo. Essas pequenas mudanças refletem as intenções originais do desenvolvedor. Por outro lado, ao mapear a intenção do desenvolvedor somente em operações de adição e remoção, as abordagens perdem conhecimento que poderia ser usado no processamento do *merge*. Além disso, é válido ressaltar que as abordagens descritas nesta seção não utilizam qualquer técnica de busca.

4. Kraken

A Kraken é uma abordagem baseada em busca que tem como objetivo minimizar o número de conflitos reportados para o desenvolvedor na execução do *merge*. Para isso, são utilizadas técnicas de buscas/otimização, heurísticas para a resolução de conflitos e o histórico de desenvolvimento para que as decisões quanto a existência ou não de conflitos possam ser tomadas com base na intenção do desenvolvedor. O problema consiste na geração de diversas sequências de deltas que serão avaliadas quanto a sua ordem de aplicação objetivando alcançar o menor número de conflitos.

Na Tabela 1 é apresentado um levantamento realizado em 6 projetos de código aberto que justificam a aplicação de uma técnica de busca dado o tamanho dos espaços de busca, ou seja, o número de sequências que podem ser geradas. Esta tabela possui dados como: tamanho médio de cada ramo, dado em função do número de deltas, o desvio padrão no tamanho dos ramos e os cenários de *merge* que possuem maior número de deltas para serem combinados. Além disso, é calculado o tamanho do espaço de busca para cada um desses cenários complexos.

Tabela 1. Análise de projetos open-source.

Projeto	Ramos		Cenário mais complexo		
	Tamanho médio	Desvio padrão	Ramo 1	Ramo 2	Espaço de busca
Git	171,3	750,6	4734	4612	$9,93 \times 10^{2810}$
Jenkins	22,4	75,5	76	2623	$1,08 \times 10^{149}$
Linux	76,8	185,8	1704	1621	$4,13 \times 10^{998}$
Perl	16,6	87,6	214	60	$2,05 \times 10^{61}$
Storm	10,9	19,4	95	19	$7,99 \times 10^{20}$
Voldemort	10,4	20,4	205	69	$7,90 \times 10^{65}$

A Kraken recebe como entrada um conjunto de ramos de um SCV que são processados até que os deltas possam ser aplicados a uma versão base. Esse processamento inclui a identificação dos deltas, o enriquecimento das operações e a aplicação de mecanismos de busca para a identificação das sequências de deltas com o menor número possível de conflitos. No restante dessa seção é apresentada a formalização do problema e são detalhadas as atividades que compõem a Kraken.

4.1. Formalização do problema

Conforme discutido anteriormente, a Kraken recebe como entrada um conjunto de ramos que, por sua vez, é formado por uma sequência de deltas, como apresentado na Figura 5. Desta forma, considere que n é o número de ramos, $R = \{r_i \mid 1 \leq i \leq n\}$ o conjunto de todos os ramos, e $r_i = (\Delta_{i,1}, \Delta_{i,2}, \dots, \Delta_{i,t(r_i)})$ a tupla que representa a sequência de deltas de um ramo r_i de comprimento $t(r_i)$. O problema em questão consiste na geração de uma sequência de deltas m , que contém todos os deltas de todos os ramos, respeitando a ordenação imposta por cada ramo, tal que o número de linhas de código em conflito, calculado pela função $f(m)$, seja minimizado.

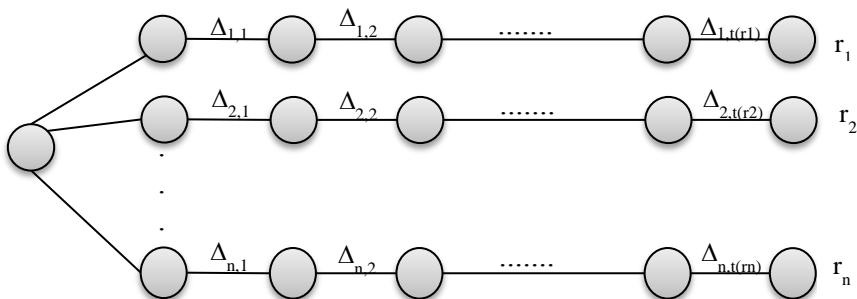


Figura 5. Ilustração dos ramos e deltas.

Para gerar as sequências de deltas candidatas para solucionar o problema, os ramos são combinados de modo que os deltas sejam preservados na mesma ordem em que aparecem em cada ramo. Ou seja, uma sequência de deltas $(\dots, \Delta_{i,p}, \dots, \Delta_{i,q}, \dots)$ é válida se, e somente se, $\forall r_i \in R, 1 \leq p < q \leq t(r_i)$. Por exemplo, suponha um cenário composto por dois ramos, r_1 e r_2 , onde r_1 é composto por $\Delta_{1,1}$ e $\Delta_{1,2}$, e r_2 é composto por

$\Delta_{2,1}$ e $\Delta_{2,2}$ é possível a geração de 6 sequências de deltas válidas: $(\Delta_{1,1}, \Delta_{1,2}, \Delta_{2,1}, \Delta_{2,2})$, $(\Delta_{1,1}, \Delta_{2,1}, \Delta_{1,2}, \Delta_{2,2})$, $(\Delta_{1,1}, \Delta_{2,1}, \Delta_{2,2}, \Delta_{1,2})$, $(\Delta_{2,1}, \Delta_{1,1}, \Delta_{2,2}, \Delta_{1,2})$, $(\Delta_{2,1}, \Delta_{1,1}, \Delta_{1,2}, \Delta_{2,2})$ e $(\Delta_{2,1}, \Delta_{2,2}, \Delta_{1,1}, \Delta_{1,2})$.

Esse problema possui espaço de busca com uma tendência de crescimento factorial. O problema está associado a uma sucessão de combinações que são aplicadas de modo que os deltas possam ser alocados em locais diferentes da sequência e o número de combinações possíveis é dado pela fórmula $\frac{[t(r_1) + t(r_2) + \dots + t(r_n)]!}{t(r_1)!t(r_2)! \dots t(r_n)!}$. Alguns exemplos estão representados na Tabela 1.

4.2. Resolução do problema

Para atingir o objetivo de minimizar o número de conflitos gerados, a Kraken foi organizada em cinco atividades, conforme representado no diagrama de atividades da Figura 6. Cada atividade possui objetivos específicos que serão descritos no decorrer desta seção.

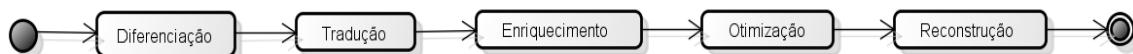


Figura 6. Visão geral da Kraken.

A Kraken recebe como entrada ramos que devem ser mesclados e a primeira atividade consiste em identificar todos os deltas presentes neles. Nesta atividade é utilizado um algoritmo de *diff* tradicional, aplicado para identificar as diferenças entre versões consecutivas (Figura 7), neste exemplo é possível visualizar o *diff* resultante do Δ_4 (Figura 3). Ou seja, é feita uma varredura entre as versões, que são analisadas duas a duas, para identificação dos deltas, em modo texto, que serão interpretados durante a atividade de tradução.

```

@@ -1,9 +1,9 @@
class Transformacao {
- public double kelvinToFahrenheit(double kelvin) {
- }
+ public double celsiusToKelvin(double celsius) {
    return celsius + 273;
}
+ public double kelvinToFahrenheit(double kelvin) {
+ }
public double fahrenheitToCelsius(double fahrenheit) {
}
  
```

Figura 7. Delta em modo texto.

A tradução é a atividade responsável por dar semântica aos deltas encontrados pela diferenciação. Os deltas identificados na atividade anterior não possuem semântica alguma para a Kraken, por se tratarem apenas de texto. Portanto, durante a tradução são identificadas as linhas que foram alteradas e essas linhas são traduzidas para operação de adição ou remoção, ainda conforme os algoritmos de *diff* tradicionais são capazes de identificar.

Com base nas operações de adição e remoção obtidas durante a tradução, a atividade de enriquecimento é executada para que a intenção do desenvolvedor seja mapeada de forma mais precisa. A principal contribuição desta atividade é identificar operações de edição, quando uma linha tem o seu conteúdo alterado, e movimentação,

quando um conjunto de linhas é movido de uma região para outra. Portanto, nessa atividade são realizadas buscas por operações que removem e adicionam o mesmo conteúdo, caracterizando uma movimentação de um bloco de linhas, e operações que substituem o conteúdo (remoção e adição na mesma linha), caracterizando uma edição.

De posse de um conjunto de deltas Enriquecidos, a atividade de otimização é realizada. Nessa atividade são geradas sequências de deltas que combinam os ramos, sempre respeitando a ordem original de cada ramo. Em seguida, os resultados do *merge* são avaliados visando a minimização do número de linhas de código conflitantes. Como o número de sequências possíveis pode ser muito grande, são utilizados algoritmos de busca, como, por exemplo, algoritmos genéticos, *hill climbing* e *simulated annealing* [Glover e Kochenberger 2003], para explorar o espaço de busca de forma eficiente, resultando em sequências que se aproximem do resultado ótimo.

O resultado da atividade de otimização consiste em uma sequência com o menor número possível de conflitos dentre as alternativas investigadas. Essa sequência é materializada através da atividade de reconstrução. Logo, dada a sequência de deltas e uma versão base (ancestral comum dos ramos) a atividade de reconstrução aplica os deltas na ordem escolhida sobre a versão base. Deste modo, o desenvolvedor recebe a versão final do *merge* com o menor número de conflitos que a Kraken foi capaz de identificar.

5. Exemplo de utilização

Na Seção 2 é discutido um exemplo onde as abordagens tradicionais de *merge* não conseguem realizar a combinação de forma automática e reportam conflitos. Alguns conflitos existentes em abordagens tradicionais ocorrem por deficiências no mapeamento da intenção do desenvolvedor e falta de heurísticas para a resolução de conflitos. Portanto, nessa seção o exemplo da Seção 2 é retomado para ilustrar como a Kraken é capaz de solucionar conflitos. Nesse exemplo, são ilustrados os resultados após cada uma das etapas e são destacadas as atividades de Enriquecimento e otimização.

Na Seção 2 é possível identificar que os deltas estão todos escritos em função de adição e remoção. Porém, existem operações que podem ser reescritas de forma mais precisa, representando melhor a intenção do desenvolvedor. Aplicando o Enriquecimento sobre este exemplo, o Δ_1 passa a ser mapeado como uma movimentação das linhas do método `fahrenheitToCelsius` do local original, abaixo do método `celsiusToKelvin`, para as linhas imediatamente acima deste método. Essa detecção é possível pois o conteúdo removido é igual ao conteúdo adicionado. Por sua vez, Δ_2 é mantido como uma adição, dado que não possui operações que se encaixam nos perfis de edição e nem de movimentação. No outro ramo, os dois deltas também são Enriquecidos. Enquanto Δ_3 é mantido como uma adição, o Δ_4 , passa a ser mapeado como uma movimentação das linhas do método `kelvinToFahrenheit` para baixo do método `celsiusToKelvin`.

Com as operações Enriquecidas, a atividade de otimização gera as sequências possíveis e calcula a função de *fitness* para cada uma delas. Durante a otimização, são geradas diversas sequências para buscar por aquela que possua o menor número de conflitos possível que, neste caso, é a sequência ($\Delta_3, \Delta_1, \Delta_2, \Delta_4$). Para que as alterações sejam aplicadas corretamente, as operações são reescritas respeitando o contexto gerado

pelos outros deltas e decorrentes do enriquecimento. Ou seja, cada um dos deltas é aplicado sobre o resultado da aplicação do delta anterior.

No caso da sequência ($\Delta_3, \Delta_1, \Delta_2, \Delta_4$), a operação de movimentação é responsável por mover linhas adicionadas em outros ramos para que o resultado obtido reflita a intenção do desenvolvedor. O processamento dessa sequência inicia aplicando Δ_3 , que consiste na adição de uma linha de código sobre a versão base. Como esse é o primeiro delta a ser aplicado, ele não sofre qualquer interferência de outros deltas. Em seguida é aplicado o Δ_1 , que consiste em mover o bloco onde o Δ_3 foi inserido. Portanto, esse delta é reescrito para que o conteúdo adicionado por Δ_3 , que está entre o início e o fim do bloco definido em Δ_1 , também seja também movido. Após a aplicação do Δ_3 , o Δ_2 é aplicado, que também consiste na adição de uma linha de código. Embora já tenham acontecido alterações sobre a versão base, essa operação não é afetada, pois está fora raio de ação dos deltas anteriores. Finalmente, Δ_4 consiste na movimentação do método `kelvinToFahrenheit`, e é alterado para incluir a linha de código adicionada pelo Δ_2 e para adaptar o destino da movimentação para compensar a ação dos deltas anteriores. Com o resultado obtido no passo de otimização, o resultado do *merge* é produzido, como mostrado na Figura 8.

```
class Transformacao {  
    public double fahrenheitToCelsius(double fahrenheit) {  
        return (5 * (fahrenheit - 32)) / 9;  
    }  
    public double celsiusToKelvin(double celsius) {  
        return celsius + 273;  
    }  
    public double kelvinToFahrenheit(double kelvin) {  
        return (9 * (kelvin - 273) / 5) + 32;  
    }  
}
```

Figura 8. Resultado com utilização da Kraken.

6. Conclusão

Neste artigo é apresentada a Kraken, uma abordagem baseada em buscas para encontrar a melhor sequência de aplicação dos deltas de forma que o número de conflitos seja minimizado. Este artigo apresenta como contribuições a formalização do problema do *merge* e uma estrutura, organizada em atividades, que é utilizada para atingir o objetivo de minimização do número que conflitos na execução do *merge*. Nessa estrutura é possível ressaltar as atividades de enriquecimento e otimização, que são inovadoras no que tange a resolução de conflitos de *merge*.

Kraken está sendo implementada utilizando a linguagem Java e usando como fonte de dados o SCV Git. Neste momento estão sendo realizados experimentos iniciais sobre projetos de código aberto, considerando *merges* que de fato ocorreram. Como trabalhos futuros serão realizados (1) experimentos confrontando o resultado obtido pela Kraken com resultados de abordagens tradicionais e com os gabaritos, *merges* reais dos repositórios; (2) experimentos para identificar o algoritmo de busca que resolve melhor este problema, iniciando por algoritmos como *Hill Climbing* e evoluindo para Algoritmos Genéticos ou Colônia de Formigas, possibilitando a escolha do algoritmo que utilize menos recurso computacional, por exemplo; e (3) a extensão da abordagem para considerar a sintaxe de linguagens de programação na solução de conflitos, caracterizando assim como *merge* sintático.

Agradecimentos

Os autores gostariam de agradecer à CAPES, ao CNPq e à FAPERJ pelo auxílio financeiro.

Referências

- Berlage, T. and Genau, A. (1993). A framework for shared applications with a replicated architecture. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*. , Symposium on User Interface Software and Technology (UIST).
- Chacon, S. (2009). *Pro Git*. 1. ed. Berkeley, CA, USA: Apress.
- Edwards, W. K., Mynatt, E. D., Petersen, K., et al. (1997). Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*. , Symposium on User Interface Software and Technology (UIST).
- Glover, F. W. and Kochenberger, G. A. [Eds.] (2003). *Handbook of Metaheuristics*. 1. ed. New York, Boston, Dordrecht, London, Moscow: Springer.
- Harman, M. (2007). The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*.
- Kermarrec, A.-M., Rowstron, A., Shapiro, M. and Druschel, P. (2001). The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. , Symposium on Principles of Distributed Computing (PODC).
- Koegel, M., Herrmannsdoerfer, M., Von Wesendonk, O. and Helming, J. (2010). Operation-based conflict detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*. , International Workshop on Model Comparison in Practice (IWMCP).
- Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE transactions on software engineering*, v. 28, n. 5, p. 449–462.
- Murta, L. G. P., Corrêa, C. K. F., Prudêncio, J. G. and Werner, C. M. L. (2008). Towards odyssey-VCS 2: improvements over a UML-based version control system. In *International Workshop on Comparison and Versioning of Software Models (CVSM)*.
- O’Sullivan, B. (2009). *Mercurial : the definitive guide*. 1. ed. Sebastopol CA: O'Reilly Media.
- Shen, H. and Sun, C. (2004). A complete textual merging algorithm for software configuration management systems. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*.
- Silva Junior, J. R., Pacheco, T., Clua, E. and Murta, L. (2012). A GPU-based Architecture for Parallel Image-aware Version Control. In *European Conference on Software Maintenance and Reengineering (CSMR)*.

Usando Hill Climbing para Identificação de Componentes de Software Similares

Antonio Ribeiro¹, Leila Silva¹, Glêdson Elias²

¹Departamento de Computação – Universidade Federal de Sergipe (UFS)
Aracaju – SE – Brazil

antoniorsj@dcomp.ufs.br, leila@ufs.br

²Centro de Informática – Universidade Federal da Paraíba (UFPB)
João Pessoa – PB – Brazil

gledson@ci.ufpb.br

Abstract. Component software reuse contributes to reduce software development time. These components are stored in repositories and search systems are needed to perform the selection of components. This paper proposes a method to reduce space storage in indexes of software assets repositories, based on applying Hill Climbing to identify similar components.

Resumo. O reuso de componentes de software contribui para a redução do tempo de desenvolvimento de aplicações. Estes componentes são armazenados em repositórios e sistemas de busca são necessários para efetuar a seleção dos mesmos. Este artigo propõe um método para economizar o espaço de armazenamento em índices de repositórios, baseado na aplicação de Hill Climbing para a identificação de componentes similares.

1. Introdução

Repositórios de componentes de software têm um grande potencial de melhorar o reuso de software. Para a busca e seleção de artefatos de software dentro do repositório faz-se necessário a construção de sistemas de buscas os quais indexam artefatos de software baseado em metadados. Estes metadados são utilizados para descrever as características sintáticas e semânticas dos artefatos.

Este trabalho propõe uma abordagem para reduzir o espaço de armazenamento de índices de repositórios de componentes de software, possibilitando otimizar o tempo de busca de artefatos para reuso. A proposta consiste em identificar grupos de artefatos de software similares. Para cada grupo gera-se um elemento representativo do grupo e é este elemento que será indexado e acessado quando uma busca por artefatos for realizada no repositório. Para realizar o agrupamento, técnicas de *Search Based Software Engineering* (SBSE) são investigadas, dado que o problema é similar ao problema de particionamento em grafos e portanto, NP-difícil.

Convém observar que este trabalho baseia-se no trabalho de Mancoridis *et al* (1998) no contexto de modularização de software. Assim, propõe-se o mapeamento do problema aqui tratado no problema investigado em Mancoridis *et al* (1998), bem como na aplicação de Hill Climbing [Clarke *et al.* 2003] para a geração dos grupos de componentes similares.

2. Geração dos Grupos de Componentes de Software Similares

Técnicas de agrupamento consistem em três estágios básicos: (i) extração de características que expressam o comportamento dos elementos a serem agrupados; (ii) definição da métrica de similaridade para comparar os elementos; e (iii) adoção de um algoritmo de agrupamento. Neste trabalho fixou-se o modelo de descrição de componente X-ARM [Elias et al. 2006] e adotou-se a abordagem proposta em [Paixão et al. 2011] nos estágios (i) e (ii). No contexto deste artigo, basta mencionar que são considerados cinco tipos de artefatos, a saber, especificações de componentes dependentes e independentes, especificações de interfaces dependentes e independentes e implementação de componente. Para cada tipo de artefato, a métrica de similaridade definida por [Paixão et al. 2011] resulta em um valor inteiro, chamado *distância* (d), no intervalo de 0 a 300, que expressa a similaridade entre artefatos. Quanto mais próximo de zero, mais similares são os artefatos.

Diferentemente do trabalho de [Paixão et al. 2011], este trabalho propõe a utilização de técnicas de SBSE realizar o estágio (iii). Mais especificamente, propõe-se um mapeamento do problema em questão para o trabalho de Mancoridis *et al* (1998), a fim de que seja usado a mesma função objetivo para realizar o agrupamento. O trabalho de Mancoridis *et al* (1998) visa identificar subsistemas de software que devem ser modularizados em um mesmo pacote, por possuírem dependência de dados. Para tal, os subsistemas são considerados vértices em um dígrafo. Dois subsistemas são ligados por uma aresta dirigida se eles possuem pelo menos uma relação de dependência, que pode ser, por exemplo, uma chamada de método. O problema reside então em encontrar um “bom particionamento” para o dígrafo D . Um bom particionamento é aquele em que subsistemas mais similares (vértices com alto grau de conexão) formem grupos e subsistemas não conectados residam em grupos distintos.

No nosso problema, um dígrafo é construído da forma descrita a seguir. Cada vértice v_i , corresponde a um artefato A_i , $1 \leq i \leq n$, onde n é o número total de artefatos de um dado tipo. Artefatos são conectados por arestas de acordo com a métrica de similaridade adotada, observando-se um limiar de corte (lc) definido pelo usuário. Assim, dados dois artefatos A_i e A_j , $i \neq j$, se a distância entre A_i e A_j for menor que lc , adiciona-se a D as arestas (v_i, v_j) e (v_j, v_i) . Caso contrário, nenhuma aresta é adicionada entre v_i e v_j .

O espaço de busca deste problema pode ser definido como todos os agrupamentos possíveis dado um conjunto de artefatos, que é claramente exponencial. Assim, a utilização de metaheurísticas é necessária para minimizar o esforço de se encontrar uma solução satisfatória. Analogamente a [Mancoridis et al. 1998], este trabalho adota o conceito de partição vizinha, NP , de uma partição P . Considera-se partição vizinha de P uma partição que difere de P somente pela junção de dois grupos distintos em P . A técnica de exploração do espaço de busca considera somente as partições vizinhas durante a busca de soluções satisfatórias. Essa exploração é guiada pela função objetivo MQ , que inclui as medidas de intraconectividade (IC) e interconectividade (E), como expresso na figura 1, onde μ_i é o número de arestas entre elementos do grupo i com N_i elementos, $\varepsilon_{i,j}$ é o número de arestas entre dois grupos distintos de N_i e N_j elementos e k é o número de grupos.

O objetivo da otimização é achar um agrupamento que ao mesmo tempo que minimiza a interconectividade, maximiza a intraconectividade. O valor obtido pela MQ é um

$$(a) \quad IC_i = \frac{\mu_i}{N_i^2} \quad (b) \quad E_{i,j} = \begin{cases} 0 & \text{se } i = j, \\ \frac{\varepsilon_{i,j}}{2N_i N_j} & \text{se } i \neq j \end{cases} \quad (c) \quad MQ = \begin{cases} \frac{1}{k} \sum_{i=1}^k IC_i - \frac{1}{\frac{k(k-1)}{2}} \sum_{i,j=1}^k E_{i,j} & \text{se } k > 1, \\ IC_1 & \text{se } k = 1 \end{cases}$$

Figura 1. Fórmulas de (a) Intraconectividade; (b) Interconectividade; (c) MQ

número real entre -1 e 1 , -1 indica a falta de coesão entre os grupos, ou seja, os grupos são formados majoritariamente por elementos não tão similares entre si, já um valor de 1 indica a ausência de acoplamento entre os grupos, que demonstra que os grupos são formados por elementos que são bastante similares entre si. O objetivo nessa otimização é portanto alcançar um valor mais próximo de 1 possível.

Para realizar o agrupamento, o algoritmo Hill Climbing foi utilizado. Há várias variações do Hill Climbing, para este trabalho adotou-se a variante padrão, na qual o algoritmo não analisa todos os vizinhos da solução atual, apenas escolhe um vizinho aleatório e então compara o resultado da função objetivo do estado corrente com o resultado do vizinho. Caso o resultado do vizinho seja melhor, este é considerado o estado corrente na próxima iteração.

3. Resultados

Na análise do desempenho do algoritmo, foi utilizada uma base sintética gerada aleatoriamente utilizando-se um gerador automático como em [Paixão et al. 2011]. O método de agrupamento proposto foi executado sobre essa base de dados, para realizar o agrupamento dos artefatos similares. Para cada grupo gerado, os respectivos elementos representativos foram construídos.

O repositório foi submetido a diferentes execuções do algoritmo, mudando apenas os parâmetros de limiar de corte e quantidade de iterações. O ganho da quantidade de componentes a indexar é diferente para cada tipo de artefato, considerando-se diferentes limiares de corte e quantidade de iterações. Observa-se que quanto menor for o número de elementos representativos, mais rápido se dará a busca e seleção dos componentes, já que somente estes elementos irão compor o índice do repositório.

Considerando-se a base de dados investigada, observou-se que para 100 iterações a redução de elementos não foi significativa para a maioria dos tipos de artefatos, mas para 250 e 500 iterações a redução foi de maior magnitude. Por exemplo, para o artefato de implementação de componentes esta redução foi de 8 para 100 iterações, 50 para 250 iterações e 92 para 500 iterações considerando o limiar de corte de 150.

Para analisar a qualidade do agrupamento obtido uma métrica externa foi adotada. Esta métrica conhecida como Silhouette, calcula a qualidade do agrupamento baseada na distância entre os artefatos originais e é largamente empregada em trabalhos de agrupamento. Este índice assume valores no intervalo de -1 e 1 , onde um valor mais próximo de 1 significa que o agrupamento é formado por grupos bem coesos, em relação à métrica de similaridade adotada, e um resultado próximo de -1 significa que há uma falta de coesão dentro dos grupos. A figura 2(b) ilustra o resultado do índice Silhouette do artefato de

implementação de componente, observando-se assim que os resultados obtidos com 250 iterações são os de melhor qualidade.

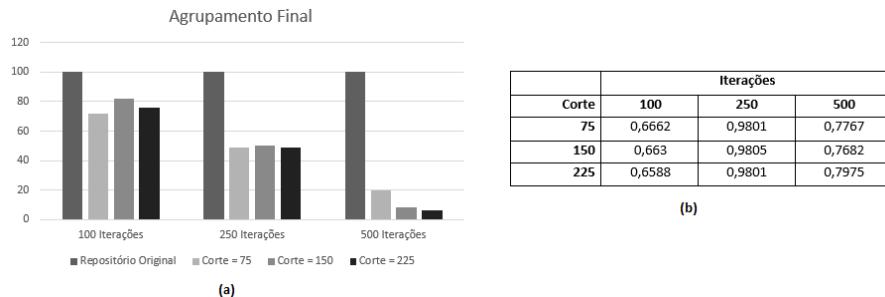


Figura 2. Artefatos de implementação: (a) Elementos a indexar (b) Silhouette

4. Conclusões

Este trabalho apresentou uma investigação preliminar do problema de identificação de componentes similares em repositórios de componentes de software. A abordagem proposta baseou-se em métricas de similaridades definidas anteriormente pelos autores e na exploração do espaço de busca baseada no trabalho de Mancoridis *et al* (1998) para o problema de modularização de software. O diferencial do trabalho está em estabelecer uma conexão entre os domínios de modularização de software e o de desenvolvimento de software baseado em componentes no contexto de SBSE.

Apesar de considerarmos o trabalho em estágio inicial e a base de dados pequena, os índices de qualidade de agrupamentos obtidos, aferidos por uma métrica externa, indicam que os resultados são promissores. No entanto, este trabalho pode ser ainda melhorado. Como trabalho futuro pretendemos investigar outras bases de dados, heurísticas e variações no mapeamento entre problemas aqui estabelecido.

Agradecimentos Este trabalho foi apoiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES), financiado pelo CNPq (Proc. 573964/2008-4).

Referências

- Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. (2003). Reformulating software engineering as a search problem. *Software, IEEE Proceedings*, 150(3):161–175.
- Elias, G., Schuenck, M., Negócio, Y., Dias, Jr, J., and Filho, S. M. (2006). X-arm: an asset representation model for component repository systems. In *Proceedings of the, SAC '06*, pages 1690–1694, New York, NY, USA. ACM.
- Mancoridis, S., Mitchell, B. S., and Rorres, C. (1998). Using automatic clustering to produce high-level system organizations of source code. In *In Proceedings of 6th Intl. Workshop on Program Comprehension*, pages 45–53.
- Paixão, M. P., Silva, L., Elias, G., and Brito, T. (2011). Clustering large-scale, distributed software component repositories. In *Proceedings of The Third International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 124–129.