UNIVERSIDADE ESTADUAL DO CEARÁ

CAMILA LOIOLA BRITO MAIA

UMA ABORDAGEM INTEGRADA, INTERATIVA E MULTI-OBJETIVA PARA OS PROBLEMAS DE SELEÇÃO, PRIORIZAÇÃO E ALOCAÇÃO DE CASOS DE TESTE

FORTALEZA - CEARÁ
2011

CAMILA LOIOLA BRITO MAIA

UMA ABORDAGEM INTEGRADA, INTERATIVA E MULTI-OBJETIVA PARA OS PROBLEMAS DE SELEÇÃO, PRIORIZAÇÃO E ALOCAÇÃO DE CASOS DE TESTE

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Sistemas de Informação

Orientador: Prof. Dr. Jerffeson Teixeira de Souza

FORTALEZA - CEARÁ
2011

M217a Maia, Camila Loiola Brito

Uma abordagem integrada, interativa e multi-objetiva para os problemas de seleção, priorização e alocação de casos de teste — Fortaleza, 2011.

104 p. : il.

Orientador: Prof. Dr. Jerffeson Teixeira de Souza

Dissertação do Mestrado Acadêmico em Ciência da Computação – Universidade Estadual do Ceará, Centro de Ciência e Tecnologia.

1. Seleção de casos de teste. 2. Priorização de casos de teste. 3. Alocação de casos de teste. 4. Metaheurísticas multi-objetivo. 5. *Search-Based Software Engineering*. I. Universidade Estadual do Ceará, Centro de Ciência e Tecnologia.

CDD: 001.6

CAMILA LOIOLA BRITO MAIA

UMA ABORDAGEM INTEGRADA, INTERATIVA E MULTI-OBJETIVA PARA OS PROBLEMAS DE SELEÇÃO, PRIORIZAÇÃO E ALOCAÇÃO DE CASOS DE TESTE

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Sistemas de Informação

Aprovada em ____ / ____ / _____.

BANCA EXAMINADORA

_____

Prof. Dr. Jerffeson Teixeira de Souza (Orientador)
Universidade Estadual do Ceará - UECE

_____

Prof. Dr. Arilo Claudio Dias Neto
Universidade Federal do Amazonas - UFAM

_____

Profa. Dra. Mariela Inés Cortés
Universidade Estadual do Ceará - UECE

Às minhas doces princesas Lorena e Larissa, ao meu amado marido Carlos Rosemberg e aos melhores pais do mundo, Sônia e Oliner.

AGRADECIMENTOS

Antes de tudo agradeço a Deus, por me dar saúde e força para manter uma rotina de estudante, profissional, filha, esposa e mãe. E especialmente por me permitir concluir esse sonho.

Agradeço aos meus pais, os melhores do mundo, por sempre incentivarem meus estudos e acreditarem em mim. Ao meu paizinho, por pagar o combustível do meu carro durante toda a faculdade, e minha mãezinha por me dar força em qualquer momento que eu preciso.

Ao meu marido Berg, meu amor, por sempre me dar força e vários e vários conselhos, além de me apoiar incondicionalmente. E também por cuidar sozinho das nossas duas princesas, nos momentos que eu precisei me ausentar da doce presença delas. Por último, pela sua paciência nos últimos dias antes da entrega da dissertação, pois você deu um show de compreensão e apoio. Sem você eu não teria conseguido, meu anjo. Obrigada! :-)

Não poderia deixar de fora as minhas duas filhas lindas, Lorena e Larissa, que são os maiores amores da minha vida. Obrigada simplesmente por existirem e fazerem a minha vida ser tão feliz. Aprendi muitas coisas e continuo aprendendo todos os dias com vocês. Amo muito vocês, minhas princesas!

Agradeço também aos meus queridos irmãos, Felipe e Danilo, e minhas cunhadinhas que adoro, Ana Eliza, Bárbara e Patrícia, que sempre me apóiam quando preciso. À minha querida sobrinha Babinha por me proporcionar momentos de alegria, e à minha querida sogrinha, Maria, por participar tão ativamente da nossa felicidade, ajudando e nos fazendo rir da vida.

Aos meus amigos, que quase esqueceram de mim, de tanto que fiquei ausente durante o mestrado, principalmente esses últimos dias.

Ao SERPRO, pela liberação de tempo para o mestrado, e às pessoas da minha equipe, das quais sempre tive todo o apoio possível, em especial minha líder Lialda.

Agradeço ao meu orientador, Jerffeson, por toda a atenção e dedicação que teve comigo, todos os conselhos e o empurrãozinho no meu futuro. Em especial, por acreditar em mim, que eu poderia fazer um bom trabalho com o GOES.Uece. Muito mais do que esse trabalho, aprendi coisas que serão lembradas a vida inteira. Obrigada por tudo, inclusive pelos puxões de orelha! :-)

Aos professores Arilo Cláudio, Mariela Inés e Gustavo Campos, pelas maravilhosas observações em relação ao meu trabalho.

Agradeço também aos amigos que fiz na Uece, principalmente os do GOES.Uece (Grupo de Otimização em Engenharia de Software), em especial ao Fabrício, Márcia, Thiago Albuquerque, Léo e Thiago Nascimento.

A mente que se abre a uma nova idéia jamais voltará ao seu tamanho original.

Albert Einstein

RESUMO

Os problemas de seleção, priorização e alocação de casos de teste podem ser considerados como difíceis, devido ao grande número de soluções possíveis que devem ser consideradas na resolução desses problemas e os diversos fatores que podem influenciar na busca dessas soluções. Existem vários trabalhos que utilizam técnicas de otimização na busca por soluções para problemas difíceis da engenharia de software, na recente área de pesquisa conhecida como *Search-Based Software Engineering* (SBSE). Dentro desse contexto, esta dissertação propõe uma abordagem integrada, interativa e multi-objetiva dos problemas de seleção, priorização e alocação de casos de teste. Ela é integrada porque os três problemas são tratados de forma sequencial, em três fases, onde a saída de uma fase é entrada para a fase seguinte. A saída de cada fase é um conjunto de soluções, e um usuário deve escolher uma dessas soluções antes que a próxima fase seja iniciada, por isso a abordagem é interativa. Além disso, a abordagem é multi-objetiva, pois considera mais de um fator para compor o conjunto de funções a ser otimizado, para cada problema. Assim, para cada um desses problemas, é apresentada uma descrição formal seguida de uma formulação matemática completa. Três experimentos foram projetados e implementados: o primeiro, com o objetivo de investigar qual metaheurística pode ser aplicada à abordagem, implementa três algoritmos multi-objetivos, NSGA-II, MOCell e SPEA2, e compara os mesmos com um algoritmo randômico. O SPEA2 gerou as melhores soluções para os três problemas, com custo computacional relativamente alto comparado às outras soluções. Já o MOCell gerou soluções em um tempo menor. O segundo experimento realiza uma análise de sensibilidade, onde o comportamento esperado dos algoritmos com a variação dos parâmetros de entrada da abordagem foi confirmado experimentalmente. Por último, o terceiro experimento realizou uma análise de competitividade, confirmando a competitividade dos algoritmos multi-objetivos em relação aos resultados gerados por usuários humanos, para todos os humanos.

Palavras–Chave: Seleção de casos de teste. Priorização de casos de teste. Alocação de casos de teste. Metaheurísticas multi-objetivo. *Search-based software engineering*.

ABSTRACT

The test cases selection, prioritization and allocation problems can be considered difficult because of the large number of possible solutions that should be considered in solving these problems and the many factors that can influence the search for these solutions. In this context, this dissertation proposes an integrated, interactive and multi-objective approach to the test cases selection, prioritization and allocation problems. It is integrated because the three problems are treated sequentially in three phases, where the output of one stage is input to the next phase. The output of each phase is a set of solutions, and a user must choose one of these solutions before the next phase is initiated, so the approach is interactive. Moreover, the approach is multi-objective, since it considers more than one factor to compose the set of functions to be optimized for each problem. Thus, for each of these problems, it is presented a formal description followed by a complete mathematic formulation. Three experiments where designed and implemented: the first, in order to investigate which metaheuristic performs better when applied to solve the proposed approach, implementing three multi-objective algorithms, NSGA-II, SPEA2 and MOCell, and compares them with a random algorithm. The SPEA2 generated the best solutions to three problems, with relatively high computational cost compared to other solutions. Yet MOCell generated solutions in a shorter time. The second experiment performs a sensibility analysis, where the expected behavior of the algorithms with the variation of input parameters was confirmed experimentally. Finally, the third experiment conducted a competitiveness analysis, confirming the competitiveness of the multi-objective algorithms in relation to the results generated by human users, to all humans.

Keywords: *Test case selection. Test case prioritization. Test case allocation. Multi-objective metaheuristics. Search-based software engineering.*

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

addtl	Additional Function Coverage Prioritization
Addtl-diff	Additional Binary-Diff Function Coverage Prioritization
APBC	Average Percentage of Block Coverage
APDC	Average Percentage of Decision Coverage
APFD	Average Percentage of Fault Detected
APSC	Average Percentage of Statement Coverage
cGA	Cellular Genetic Algorithm
GRASP	Greedy Randomized Adaptive Search Procedure
MOCell	Multi-Objective Cellular Genetic Algorithm
NSGA-II	Non-dominated Sorting Genetic Algorithm II
SBSE	Search-Based Software Engineering
total	total function coverage prioritization
total-diff	total binary-diff function coverage prioritization

SUMÁRIO

1 INTRODUÇÃO

Diversos problemas relacionados à Engenharia de Software podem ser considerados como complexos, como, por exemplo, alocação de tarefas, escolha de requisitos que devem ser implementados no próximo *release* e seleção de casos de teste. Esses problemas podem apresentar um número alto de possíveis soluções e alta complexidade, tornando-os grandes candidatos a serem resolvidos por métodos automáticos ou semi-automáticos. Se for possível contar com esses métodos para auxiliar no processo de decisão, os usuários do sistema, neste caso gerentes de projeto ou engenheiros de software, podem se focar em aplicar sua experiência no processo de decisão, após a resposta do método utilizado ou durante a execução do mesmo, orientando a busca de soluções.

Uma área relativamente nova, unindo Engenharia de Software e Pesquisa Operacional, chamada *Search-Based Software Engineering* (HARMAN e JONES, 2001), ou SBSE, tem obtido grande interesse recente da comunidade acadêmica. A idéia principal é aplicar técnicas de otimização em problemas da Engenharia de Software que podem ser formulados matematicamente. Uma ou mais funções objetivo são definidas, bem como as restrições do problema, e as técnicas de busca são aplicadas a fim de encontrar soluções próximas de uma solução ótima.

Metaheurísticas como Têmpera Simulada (KIRKPATRICK et al., 1983), Algoritmos Genéticos (HOLLAND, 1975), Busca Tabu (GLOVER e LAGUNA, 1993), NSGA-II (DEB et al., 2002), MOCell (NEBRO et al., 2009) e outras têm sido aplicadas com sucesso a problemas da Engenharia de Software (HARMAN e JONES, 2001). Porém, segundo (HARMAN, 2007), independentemente da técnica de busca empregada, é a função objetivo que captura a informação crucial, e será utilizada para diferenciar boas soluções das ruins.

O termo *Search-Based Software Engineering* foi criado em 2001 por Harman e Jones (HARMAN e JONES, 2001), e desde então uma grande quantidade de

trabalhos têm sido desenvolvido nesta área. Alguns exemplos de áreas da Engenharia de Software que já possuem aplicação destas técnicas são: engenharia de requisitos (BAGNALL et al., 2001), análise e projeto (LI et al., 2010), codificação (KUPERBERG e OMRI, 2009), teste de software (LI et al., 2007), manutenção (SCHMIDT, 2009), gerência de projetos (DI PENTA et al., 2007) e métricas (ANTONIOL et al., 2009).

No estudo realizado em (HARMAN et al., 2009) pode-se verificar que a maioria dos trabalhos em *Search-Based Software Engineering* é relacionado a teste e depuração, com cerca de 59% dos trabalhos até então. Destes, grande parte é referente à geração de dados de teste caixa branca. Porém, há também trabalhos em priorização de casos de teste (WALCOTT et al., 2006) (LI et al., 2007) (MAIA et al., 2010), seleção de casos de teste (YOO e HARMAN, 2007) (MANSOUR et al., 2001) (MAIA et al., 2009), minimização de suíte de testes (TALLAM e GUPTA, 2006), testes não funcionais (ALANDER et al., 1997) (BRIAND et al., 2006), dentre outros. O fato de haver mais trabalhos em teste pode ter relação com a questão histórica, pois o primeiro trabalho publicado na área de SBSE foi sobre geração de dados de teste (MILLER e SPOONER, 1976), e ocorreram outros trabalhos de SBSE com testes antes de haver trabalhos em outras áreas da engenharia de software.

O principal objetivo do processo de testes é encontrar o maior número de erros possível (BASTOS et al., 2007), para que possam ser corrigidos e o cliente possa receber uma versão confiável do sistema. Para que isso ocorra, é importante haver um planejamento da execução dos testes, de modo a garantir que os testes mais significativos sejam executados.

O planejamento da execução dos testes consiste em selecionar e/ou priorizar os casos de teste a serem executados, e alocá-los aos testadores disponíveis para sua execução. Normalmente, são levados em consideração nesses processos o tempo disponível para a execução dos casos de teste, e fatores como a importância dos requisitos para o negócio do cliente, experiência dos testadores, dentre outros. Pode parecer um problema simples, porém quanto mais requisitos, casos de teste e

testadores existem, maior a complexidade para analisar os dados e encontrar uma solução aceitável para o planejamento da execução dos testes.

Mais especificamente, a seleção de casos de teste consiste em escolher quais casos de teste serão executados, a priorização de casos de teste consiste em priorizar os casos de teste de acordo com algum critério (MATHUR, 2008), de modo que os mais significativos sejam executados antes, e a alocação de casos de teste consiste em escolher quais testadores irão executar quais casos de teste.

Nos trabalhos (COLARES et al., 2009), (WALCOTT et al., 2006) e (LI et al., 2007) foi demonstrado que as técnicas de busca superam outras técnicas já utilizadas para resolução de problemas da engenharia de software modelados matematicamente. O trabalho (SOUZA et al., 2010) mostra que as técnicas de busca superam também a resposta humana para alguns problemas da engenharia de software, comprovando a competitividade dessas técnicas naqueles cenários.

Sendo assim, a proposta deste trabalho é formular matematicamente, de forma multi-objetiva, integrada e iterativa, os problemas de seleção, priorização e alocação de casos de teste e aplicar técnicas baseadas em busca, mais especificamente metaheurísticas, na resolução dos mesmos. A formulação será multi-objetiva porque leva em consideração diversos fatores que podem ser otimizados, nos três problemas. É também integrada porque os problemas são modelados de forma a serem executados sequencialmente, ou seja, o resultado de um é entrada para a execução do processo seguinte. Além disso, a abordagem é iterativa, pois quando cada um desses processos exibe um conjunto de soluções para o usuário, é este usuário que escolherá qual das soluções apresentadas será a entrada para o processo seguinte.

Na abordagem proposta por esta dissertação, serão aplicadas metaheurísticas multi-objetivo para a resolução destes problemas. Como os problemas acima possuem vários fatores a serem levados em consideração, podemos formulá-los matematicamente como problemas multi-objetivos e aplicar técnicas de otimização para encontrar soluções no mínimo próximas de ótimas. A FIGURA 1 a seguir mostra a visão geral da solução proposta.

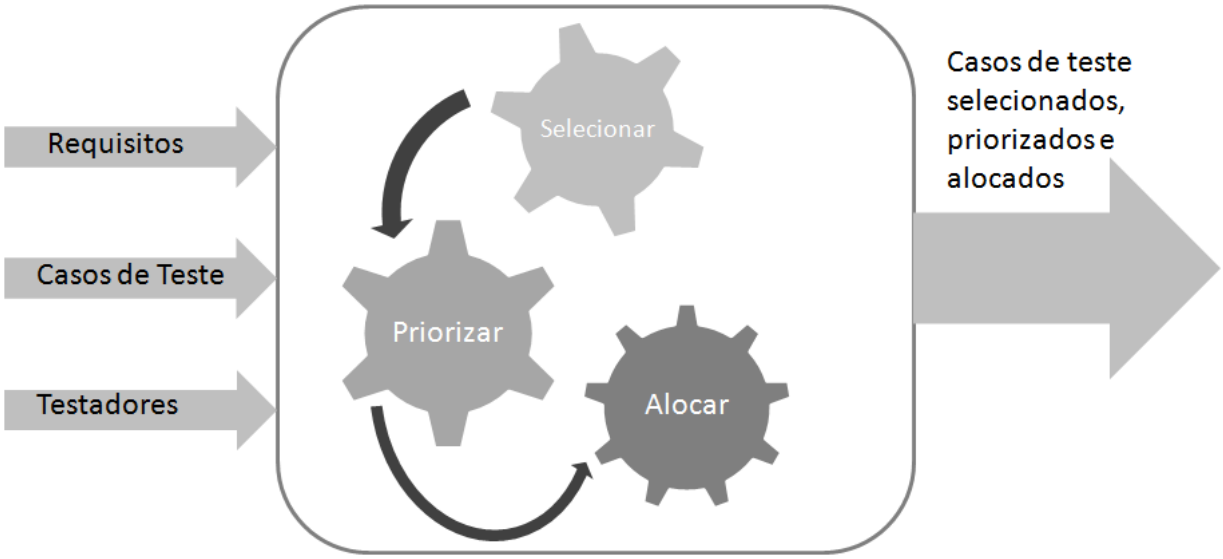

FIGURA 1 – Visão geral da solução proposta

As entradas para as técnicas de otimização a serem utilizadas serão: dados dos requisitos do sistema, dados dos casos de teste e dados dos testadores disponíveis para o projeto de software.

1.1 Motivação

A área de Teste de Software tem uma grande importância no desenvolvimento. Algumas das vantagens de se testar um software antes que ele entre em produção são:

1. O produto testado possui mais qualidade que um produto não testado, uma vez que este foi exercitado com a finalidade de detectar erros antes de sua entrega ao cliente;

2. O custo de correção de um defeito encontrado na fase de testes é muito menor que o custo de uma falha encontrada quando o sistema já está em uso pelo cliente (MYERS, 2004);

3. Verifica se os requisitos especificados pelo cliente foram implementados de forma correta.

Segundo (BASTOS et al., 2007), os defeitos existentes em um software representam riscos tanto para o negócio quanto para a imagem do cliente. Torna-se necessário, então, entregar para o cliente uma versão o mais bem testada possível, de modo a garantir uma qualidade mínima ao produto desenvolvido. Porém, dada a crescente complexidade dos sistemas atuais e o surgimento constante de novas tecnologias, o esforço necessário para testar estas aplicações é cada vez maior.

Para que o processo de testes realmente seja um diferencial no desenvolvimento do software, contribuindo para a boa qualidade do produto final, é necessário um bom planejamento dos testes, inclusive da execução dos casos de teste.

Os três problemas anteriormente definidos são de difícil solução, uma vez que possuem vários fatores que devem ser levados em consideração e analisados quando a solução estiver sendo buscada, por exemplo, tempo de execução do caso de teste, prioridade do requisito testado, habilidade para testar um determinado caso de teste, dentre outros. Além disso, à medida que o número de requisitos, casos de teste e testadores aumenta, a resolução dos problemas se torna mais complexo, devido ao aumento do espaço de busca.

A alocação dos casos de teste deve sempre ser realizada em um projeto cujos testes são executados manualmente. A seleção e/ou a priorização também, porém às vezes realizamos as duas atividades, e outras vezes escolhemos selecionar ou priorizar. A abordagem proposta considera que deve-se selecionar primeiro e posteriormente priorizar os casos de teste selecionados.

1.2 Objetivo Geral

Criar uma abordagem integrada, iterativa e multi-objetiva para os problemas de seleção, priorização e alocação de casos de teste.

1.3 Objetivos Específicos

Para alcançar o objetivo geral descrito acima, os seguintes objetivos específicos foram definidos:

a) Formular matematicamente e de maneira integrada os problemas de seleção, priorização e alocação de casos de teste de sistema, levando em consideração informações sobre os requisitos do sistema, casos de teste e dos testadores disponíveis para alocação, e de modo a permitir a participação do usuário no processo de busca por soluções.

b) Aplicar técnicas de otimização multi-objetivo para resolver os problemas formulados anteriormente (seleção de casos de teste, priorização de casos de teste e alocação de casos de teste para os testadores).

c) Comparar os resultados obtidos com respostas de especialistas da área.

1.4 Organização do Trabalho

Este trabalho está organizado em 6 capítulos.

O Capítulo 2 apresenta uma breve fundamentação teórica necessária para a compreensão da abordagem proposta.

O Capítulo 3 detalha alguns trabalhos relacionados aos problemas de seleção, priorização e alocação de casos de teste, mais precisamente na aplicação de técnicas de otimização a estes problemas.

O Capítulo 4 detalha os problemas de seleção, priorização e alocação de casos de teste, e exibe a formulação proposta de modo integrado para os três.

O Capítulo 5 apresenta os resultados dos experimentos e exibe a avaliação de forma detalhada, enquanto o Capítulo 6 cita as conclusões e discorre sobre possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos necessários para o completo entendimento da abordagem proposta nesta dissertação. São apresentados conceitos sobre teste de software, otimização, metaheurísticas e *Search-Based Software Engineering* (SBSE).

2.1 Teste de Software

2.1.1 Introdução

O objetivo principal das empresas que desenvolvem software é satisfazer a expectativa do usuário. Em outras palavras, entregar o produto que o cliente espera receber.

Uma das maneiras de se minimizar os defeitos do software e melhorar a qualidade deste é realizando teste de software, ou seja, verificar, antes de entregar o produto para o cliente, se o que foi desenvolvido está de acordo com o que o cliente solicitou. Segundo (PRESSMAN, 2001), o teste representa a revisão final das fases de especificação, projeto e implementação.

Os casos de teste são os cenários a serem executados para verificar se a implementação está de acordo com a especificação (BASTOS et al., 2007). São compostos por pré-condições, pós-condições, passo a passo de como executar o teste e as configurações necessárias para que o teste ocorra. Os casos de teste são originados da especificação de requisitos e regras de negócio, representando o que deve ser testado no sistema. Um conjunto de casos de teste é chamado de suíte de testes.

Em (PRESSMAN, 2001), o autor afirma que o Princípio de Pareto pode ser aplicado ao teste de software: 80% dos problemas encontrados estão em 20% dos seus componentes. Sendo assim, é importante saber quais casos de teste tem a maior probabilidade de detectar esses problemas. Caso seja necessário escolher ou ordenar

esses casos de teste para a execução, isso deve ser levado em consideração, para que esses casos de teste pertencentes aos 20% que detectam mais problemas possam ser selecionados e/ou priorizados. Vários fatores podem influenciar nesta probabilidade de detectar problemas, dentre os quais pode-se destacar as características dos requisitos e características dos casos de teste.

2.1.2 Processo de Teste

O processo de testes é composto, geralmente, pelas seguintes fases: planejamento, projeto e implementação de casos de teste, execução, avaliação dos resultados e gerência de defeitos.

Na fase de planejamento, o Plano de Testes é elaborado com o objetivo de definir estratégias e tipos de teste a serem realizados, identificar os riscos relacionados aos testes e elaborar um cronograma para a execução.

As atividades de seleção, priorização e alocação de casos de teste podem ser realizadas no início do projeto de testes, na fase de planejamento, quando se trata de teste de regressão ou quando os casos de teste a serem executados já são conhecidos. Quando o cenário é o desenvolvimento de um sistema desde o início, o planejamento de execução pode ser realizado após o projeto dos casos de teste, quando os mesmos são conhecidos.

É na fase de projeto e implementação que os casos de teste são elaborados ou alterados, e implementados, e o ambiente de testes é planejado e montado. A fase de execução engloba a execução dos testes e o registro dos defeitos encontrados pelos testadores. Por último, a avaliação dos resultados ocorre em paralelo com a execução dos testes, e é na realidade o acompanhamento da resolução dos defeitos encontrados na execução dos testes.

2.1.3 Níveis de Teste

Ao longo do processo de desenvolvimento, podemos ter os seguintes níveis de teste: teste unitário, teste de integração, teste de sistema e teste de aceitação.

O teste unitário, ou teste de unidade, verifica a menor unidade do software (PRESSMAN, 2001), que pode ser um método, classe, ou o que for considerado como unidade. Os próprios desenvolvedores executam este tipo de teste, enquanto estão implementando as unidades, por isso este tipo de teste normalmente é considerado como um apêndice ao passo de codificação (PRESSMAN, 2001). O objetivo é testar apenas uma unidade específica, e não sua integração com as demais.

Após os testes de unidade, podem-se realizar os testes de integração, quando duas ou mais unidades são testadas em conjunto para verificar a integração entre elas.

Quando juntamos todas as unidades, são realizados os testes de sistema, com a finalidade de exercitar por completo o sistema (PRESSMAN, 2001). Por último, o teste de aceitação também exercitará o sistema por completo, porém sob a visão do cliente, que é o executor deste nível de teste.

2.1.4 Tipos de Teste em Relação ao Conhecimento do Código

O teste pode ser executado na forma caixa branca ou caixa preta. No teste caixa branca, o código do sistema é examinado, e casos de teste são feitos para testar os caminhos lógicos do código, exercitando a lógica do programa exaustivamente (PRESSMAN, 2001).

Por outro lado, no teste caixa preta não é necessário conhecer o código da aplicação, pois os testes são conduzidos a partir da interface do sistema. O objetivo é demonstrar que as funções estão operacionais, que o sistema está processando corretamente as entradas e exibindo o resultado esperado (PRESSMAN, 2001), de acordo com a especificação de requisitos.

2.1.5 Teste de Regressão

Os testes de regressão voltam a testar funcionalidades que já haviam sido testadas ou que já estavam em uso pelo cliente. Geralmente, ocorre quando o cliente solicita alterações no sistema, como adição de novas funcionalidades ou alteração de

alguma já existente, e pretende-se garantir que as funcionalidades anteriores continuam funcionando como deveriam.

Em outras palavras, é quando um programa é testado de modo a garantir que não somente as novas funcionalidades estão corretas, mas também que as funcionalidades não alteradas da versão anterior continuam se comportando adequadamente (MATHUR, 2008).

Para liberar uma versão inicial do programa para o cliente, é necessário apenas desenvolver, testar e finalmente entregar a *release*. Porém, para uma versão posterior do programa, o processo consiste em realizar as modificações solicitadas na versão já testada, gerando a nova versão, testar as novas funcionalidades, testar as funcionalidades da versão anterior não alteradas, e finalmente entregar a *release* do programa (MATHUR, 2008).

Para garantir que a versão anterior continua funcionando da mesma maneira, a estratégia de re-executar todos os casos de teste pode ser aplicada. Por um lado, esta estratégia praticamente elimina os riscos de algum problema não ser descoberto, porém sua grande desvantagem é o tempo de execução necessário para executar todos os testes (MATHUR, 2001), já que geralmente não há tempo suficiente para isso.

2.2 Otimização

Um problema de otimização pode ser definido como a busca pelo valor ótimo de uma função (COLLETTE e SIARRY, 2003) ou um conjunto de funções, chamadas de função objetivo. Uma boa função objetivo é determinante para o sucesso da busca pelas soluções (MCMINN, 2004). Além da função objetivo existem as restrições, que são condições que uma solução deve satisfazer para ser considerada válida. Caso uma solução não satisfaça uma restrição do problema, esta não é uma solução válida para este problema.

Existem duas categorias de problemas de otimização: mono-objetivo e multi-objetivo. Um problema mono-objetivo possui apenas uma função a ser otimizada. A definição formal para este tipo de problema é na forma:

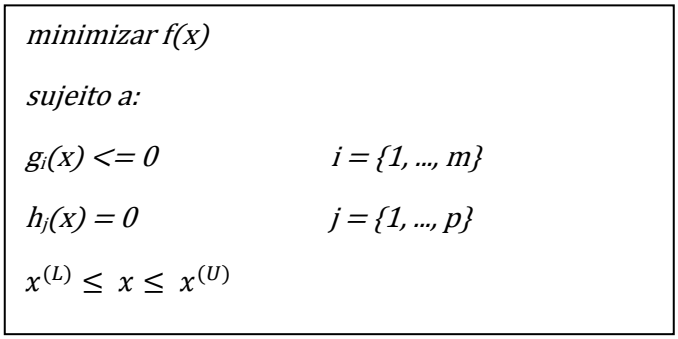

FIGURA 2 – Definição de um problema de otimização mono-objetivo

Na FIGURA 2, a função *f(x)* pode ser minimizada ou maximizada, dependendo do objetivo. As duas equações seguintes são as restrições do problema, sendo a primeira uma restrição de desigualdade e a segunda uma restrição de igualdade. Um problema pode ter várias restrições de igualdade e desigualdade, representadas por g_i(x) e h_j(x).

Um problema multi-objetivo consiste em otimizar um conjunto de funções objetivo simultaneamente (COELLO et al., 2006), ou seja, encontrar um vetor de variáveis X_i de decisão que otimize esse conjunto de funções, respeitando o conjunto de restrições do problema. Essas funções normalmente são conflitantes entre si.

A definição formal de um problema multi-objetivo segue o formato a seguir:

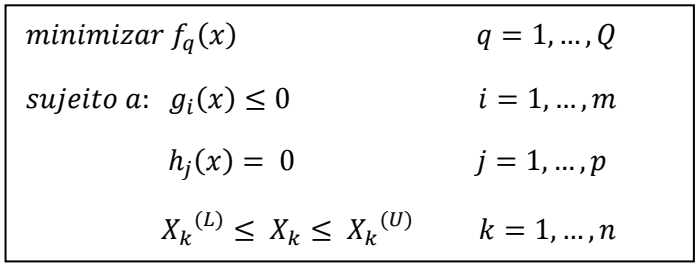

FIGURA 3 – Definição de um problema de otimização multi-objetivo

Deseja-se minimizar (ou maximizar) um conjunto de funções, representadas por f_q, onde q = 1 a Q. No problema multi-objetivo temos as mesmas restrições de igualdade e desigualdade encontradas no problema mono-objetivo.

Na otimização de problemas mono-objetivo, onde apenas uma função deve ser otimizada, a ordenação das soluções é total, ou seja, na comparação de quaisquer duas soluções é possível determinar qual é a melhor ou se elas são iguais. Porém, na otimização multi-objetivo, quando mais de uma função objetivo deve ser otimizada, nem sempre é possível determinar qual entre duas soluções é a melhor, pois há mais de um critério de comparação: a solução S_1 pode ser melhor que a solução S_2 em um objetivo, porém a solução S_2 pode ser melhor que S_1 no outro objetivo. Assim, a ordenação das soluções geradas nesta abordagem é dita ser parcial, e é baseada no conceito de dominância.

Uma solução S_1 domina outra solução S_2 quando: (DEB, 2008)

- A solução S_1 não é pior que S_2 em todos os objetivos;
- A solução S_1 é melhor em pelo menos um dos objetivos.

Para ilustrar este conceito, visualizemos a FIGURA 4, que representa um conjunto de 5 soluções e duas funções-objetivo: f_1 e f_2, que devem ser minimizadas.

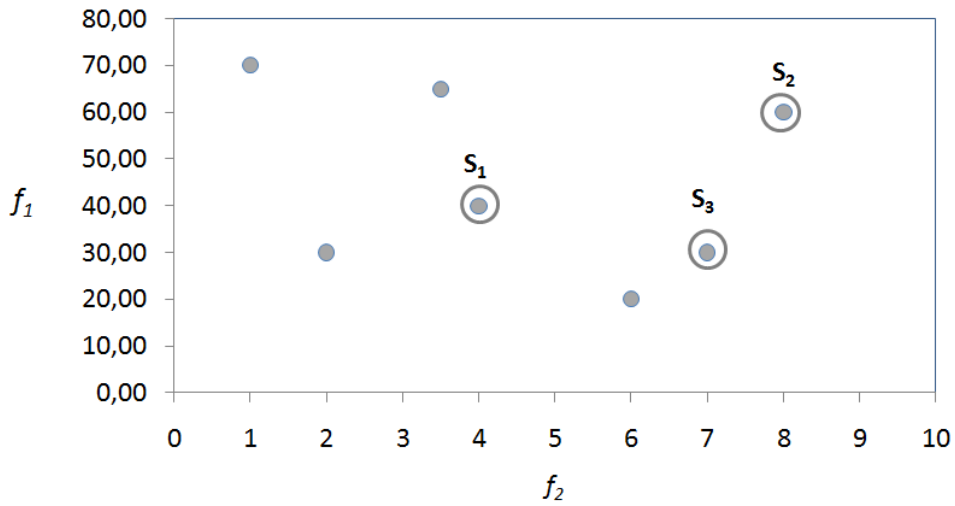

FIGURA 4 – Exemplo de dominância

Sejam as soluções S_1, S_2 e S_3 selecionadas na FIGURA 4. É possível observar que a solução S_1 possui valores menores do que S_2 tanto para a função f_1 quanto para f_2. Como as duas funções devem ser minimizadas, S_1 é melhor que S_2 nos dois objetivos. Então, S_1 domina S_2. Pelo mesmo motivo, S_3 domina S_2. Tomemos como exemplo agora as soluções S_1 e S_3. Para a função f_1, S_3 possui valor menor (e melhor), e para a função f_2, S_1 possui valor menor (e melhor). Neste caso, cada uma das soluções é melhor relativamente a uma das funções. Logo, não há relação de dominância entre S_1 e S_3.

Quando falamos em dominância, podemos definir quatro áreas na projeção de soluções no espaço de objetivos (COLLETTE e SIARRY, 2003), representadas na FIGURA 5. Por exemplo, tomemos como base uma solução S_1, que está selecionada. Se a solução a ser comparada com S_1 estiver nas áreas brancas do gráfico, a comparação é indiferente, ou seja, não há relação de dominância entre S_1 e a outra solução. Caso a solução a ser comparada esteja na área superior direita, S_1 domina esta solução. Em contrapartida, se esta solução estiver na área inferior esquerda do gráfico, S_1 é dominada por esta solução.

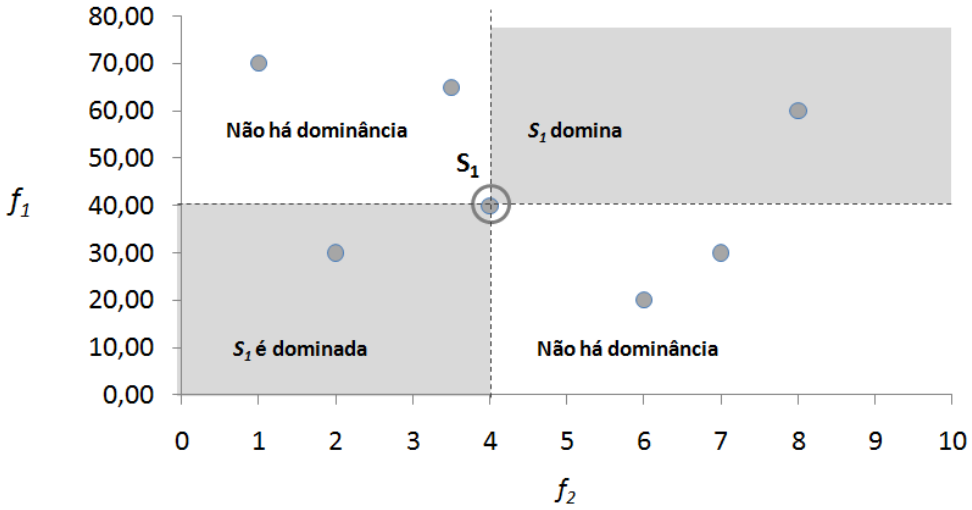

FIGURA 5 – Relações de dominância

A partir da projeção das soluções no espaço de objetivos, é possível classificar as soluções como sendo soluções não dominadas, ou seja, que não são dominadas por qualquer outra solução, e soluções dominadas, isto é, as soluções que são dominadas por pelo menos uma outra solução. O conjunto de todas as soluções não dominadas formam a Frente de Pareto (ou Fronteira de Pareto), que representa o conjunto de soluções eficientes para o problema. Essas soluções pertencentes à Frente de Pareto são igualmente boas porque não existe relação de dominância entre elas.

Então, para um problema multi-objetivo, é desejável encontrar soluções o mais próximo possível da Frente de Pareto real do problema, e também o mais diversas possível (DEB, 2008).

Existem três métodos para resolver problemas multi-objetivos: métodos à priori, métodos à posteriori e métodos interativos. No método à priori (COELLO et al., 2006), o usuário informa suas preferências antes do processo de busca de soluções. Um exemplo deste método é a composição de várias funções objetivo em uma só, onde cada objetivo recebe um peso informado pelo usuário, que representa sua influência no valor da função composta. Desta maneira, o problema é tratado como mono-objetivo. No método à posteriori (COELLO et al., 2006), o usuário informa suas preferências após o processo de busca ter sido concluído, ou seja, ele recebe um conjunto de soluções boas e pode escolher qual objetivo priorizar durante a escolha da solução. No método interativo (TAKAGI, 2001), o usuário informa suas preferências durante o processo de busca, guiando esse processo de busca de soluções.

2.3 Metaheurísticas

Várias técnicas de otimização podem ser utilizadas nos problemas que são formulados matematicamente, dentre elas métodos exatos, métodos aproximativos, heurísticas e meta-heurísticas.

Técnicas de busca metaheurísticas são algoritmos que encontram soluções ótimas ou próximo de ótimas para problemas de otimização (GLOVER e

KOCHENBERGER, 2003). Segundo (REEVES, 1993), metaheurísticas são técnicas utilizadas para encontrar soluções boas para problemas difíceis, utilizando custo computacional razoável.

As metaheurísticas são abordagens de busca genéricas, podendo ser facilmente adaptadas para resolver vários tipos de problemas, principalmente poblemas considerados difíceis, especialmente aqueles pertencentes às classes NP-completo e NP-difícil.

As metaheurísticas podem resolver problemas mono e multi-objetivos. A seguir são descritas algumas metaheurísticas bastante aplicadas a problemas da engenharia de software. Nesta pesquisa serão utilizadas as metaheurísticas NSGA-II, MOCell e SPEA2, apresentadas nas subseções 2.3.2, 2.3.3 e 2.3.4, respectivamente. O Algoritmo Genético será apresentado porque as demais metaheurísticas são baseadas no mesmo.

2.3.1 Algoritmo Genético

Um algoritmo genético é uma variante do algoritmo evolucionário, e seu conceito foi introduzido por (HOLLAND, 1975) em seu livro *Adaptation in Natural and Artificial Systems*, de 1975.

O conceito básico de um algoritmo evolucionário é que cada solução é um indivíduo em uma população (conjunto de soluções), e deve haver uma representação para esses indivíduos ou cromossomos. Os indivíduos se combinam e formam novos indivíduos, chamados herdeiros ou descendentes. Os indivíduos mais “fortes”, ou seja, as soluções que possuem melhor valor para a função objetivo, sobrevivem e passam a fazer parte da próxima geração da população. Após a seleção de quais indivíduos serão os “pais”, há a recombinação e a mutação. A recombinação (ou *crossover*) envolve dois o mais indivíduos, gerando um ou mais herdeiros, enquanto a mutação envolve apenas um indivíduo, que sofrerá uma alteração e gerará um novo indivíduo (EIBEN e SMITH, 2003). A recombinação e a mutação estão exemplificadas na FIGURA 6.

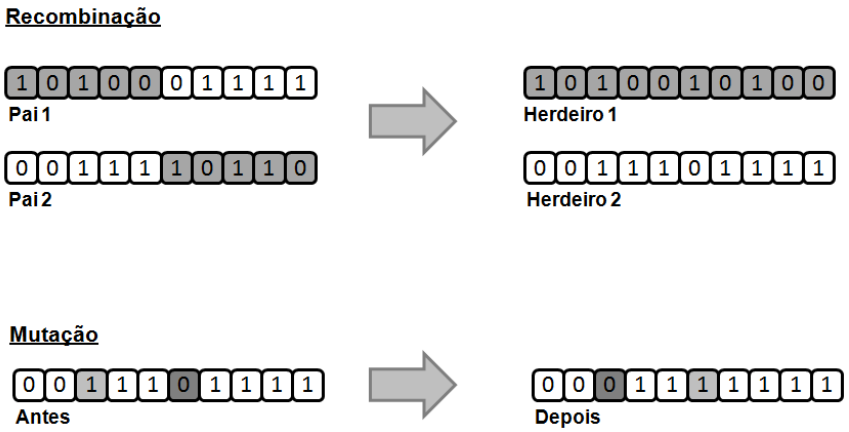

FIGURA 6 – Exemplo das operações de recombinação e mutação do algoritmo genético

O exemplo da recombinação possui um ponto de corte, que será a base para a formação dos herdeiros. A primeira parte do primeiro indivíduo (pai 1) é combinada com a segunda parte do segundo indivíduo (pai 2), formando um novo herdeiro. Da mesma forma, gera-se o segundo herdeiro combinando-se a primeira parte do segundo indivíduo com a segunda parte do primeiro indivíduo. Já o exemplo da mutação considera a troca de posição dois elementos em um determinado indivíduo. É importante citar que há diversos tipos de recombinação e de mutação, como pode ser visto em (EIBEN e SMITH, 2003).

O algoritmo genético foi inicialmente um estudo sobre a adaptabilidade, inspirado na teoria da seleção natural de Darwin (DARWIN, 1859). Um GA básico possui uma representação, uma técnica de seleção de indivíduos baseada em sua função objetivo, uma probabilidade baixa de mutação e uma recombinação baseada nas características dos pais (EIBEN e SMITH, 2003). A representação dos indivíduos, porém, pode ser realizada de diversas formas, não somente como binária, mas como inteira, real, dentre outras opções.

A FIGURA 7 apresenta o algoritmo genético. Após a geração da população inicial, repete-se o seguinte processo até que um critério de parada seja atingido: são selecionados pelo menos dois indivíduos (pais), e em seguida é realizada a recombinação entre eles e a mutação, com as devidas probabilidades. Ao final, os herdeiros gerados são adicionados à população.

```
Procedimento AlgoritmoGenetico

1.   P1 ← geraPopulacaoInicial()
2.   Repetir
3.      P2 ← selecionaPais(P1)
4.      P3 ← recombinacao(P2)
5.      P3 ← mutacao (P3)
6.      P1 ← P3
7.   até CriterioDeParada
8.   Retornar (s)
Fim
```

FIGURA 7 – Algoritmo Genético

2.3.2 NSGA-II

NSGA-II (*Nondominated Sorting Genetic Algorithm* II) é uma versão multi-objetiva de um algoritmo genético, proposto em (DEB et al., 2002), e possui duas fases.

A primeira, chamada *non-dominated sorting*, separa as soluções em grupos que representam o nível de não-dominância para estas soluções, ou seja, um grupo com soluções que não são dominadas por outras soluções, um segundo grupo com soluções não dominadas por outras soluções com exceção das soluções do primeiro grupo, e assim por diante. Uma população inicial de tamanho *N* é gerada, normalmente de forma aleatória, e posteriormente uma segunda população é gerada, baseada na primeira, aplicando os operadores genéticos de seleção, recombinação e mutação. Então, uma população combinada de tamanho *2N* é formada baseada nestas duas primeiras populações. Depois, grupos de soluções são formados baseados em seu valor de não-dominância. As soluções que não são dominadas por qualquer outra solução compõem o grupo *F1*, que representa a primeira Frente de Pareto. O grupo *F2* é formado pelas soluções que não são dominadas por nenhuma outra, não considerando os elementos de *F1*. O grupo *F3* é formado por soluções que não são dominadas por outras, não considerando os elementos de *F1* e *F2*, e assim por diante.

Como pode ser visto na Figura 8, a população P_{t+1} deve ser formada por apenas *N* elementos. Os elementos da população de tamanho *2N* são adicionados na

população P_{t+1}, da seguinte maneira: o primeiro grupo formado anteriormente (F1) é adicionado. Em seguida, o segundo grupo (F2) é adicionado, e assim por diante. Quando um grupo não couber na nova população, a segunda fase do algoritmo, chamada de *crowding-distance sorting* é executada. Nesta fase, a diversidade das soluções do grupo é considerada, para que seja possível selecionar apenas alguns elementos do grupo que possam "entrar" na população P_{t+1}.

A estimativa de densidade das soluções geradas é calculada levando em consideração a distância média de dois pontos em ambos os lados ao longo de cada objetivo. O valor do perímetro do cubo formado usando os vizinhos mais próximos como vértices é calculado. O valor de *crowding-distance* para uma dada solução é a soma dos valores de distância individuais de cada objetivo. Este valor de *crowding-distance* é usado para selecionar os elementos que farão parte da população P_{t+1} de tamanho *N*, como pode ser visto na FIGURA 8.

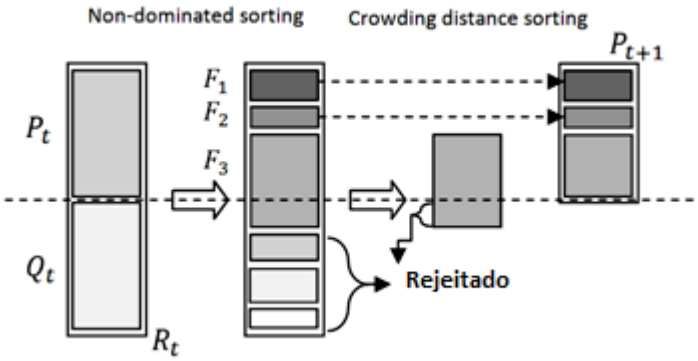

FIGURA 8 – Procedimento NSGA-II – adaptado de (DEB et al., 2002)

Os parâmetros a serem setados para o NSGA-II são: a estratégia de seleção dos indivíduos, a estratégia e a taxa de recombinação, a estratégia e a taxa de mutação, o tamanho da população e a quantidade de avaliações.

2.3.3 MOCell

O MOCell, proposto em (NEBRO et al., 2009), é uma adaptação de um cGA (*cellular genetic algorithm*) canônico para abordagens multi-objetivas. Em um algoritmo genético celular, um indivíduo pode reproduzir somente com seus vizinhos

do círculo de reprodução (NEBRO et al., 2009), diferentemente de um algoritmo genético tradicional, onde os indivíduos podem se combinar com qualquer outro indivíduo para reproduzir. A FIGURA 9 mostra o algoritmo do MOCell.

Inicialmente, uma Frente de Pareto vazia é gerada pelo algoritmo. Os indivíduos são organizados em *grids* bidimencionais, e as operações genéticas são então aplicadas sucessivamente sobre eles, até que uma condição de parada seja atingida (NEBRO et al., 2009). Um herdeiro é gerado por vez, e após a sua geração (recombinação dos pais e mutação), ele é avaliado. Caso não seja dominado por outro indivíduo, ele é adicionado à Frente de Pareto e a uma população auxiliar. Ao final de cada iteração, a população atual é substituída pela população auxiliar, que conterá um conjunto de soluções não dominadas.

O procedimento *Feedback*, na linha 13 do algoritmo, é responsável por sobrescrever randomicamente um determinado número de indivíduos na população por elementos pertencentes a um arquivo de Frente de Pareto.

Um estimador de densidade baseado no algoritmo de *crowding distance* é utilizado para inserir soluções no arquivo de Frente de Pareto, com o propósito de obter um conjunto diverso. Este estimador é utilizado para remover soluções do arquivo de Frente de Pareto quando este se torna cheio (limite atingido).

```
Procedimento mocell(parametros)

1.  ParetoFront = criaFronteParetoVazia()
2.  Enquanto (!condicaoTermino()) do
3.    For individuo ← 1 a parametros.tamanhoPopulacao do
4.      n_list ← Neighborhood(parametros,posicao(individuo));
5.      pais ← selecao(n_list);
6.      herdeiro ← recombinacao(parametros.taxaCrossover,pais);
7.      herdeiro  ← mutacao(parametros.taxaMutacao,herdeiro);
8.      avaliaFuncaoFitness(herdeiro);
9.
    adiciona(posicao(individuo),herdeiro,parametros,populacaoAuxiliar);
10.     adicionaFrontePareto(individuo);
11.   end For
12.   populacao ← populacaoAuxiliar;
13.   populacao ← Feedback(parametros,ParetoFront);
14. end Enquanto
Fim
```

FIGURA 9 – Algoritmo MOCell – adaptado de (NEBRO et al., 2009)

Os parâmetros a serem setados para o MOCell são: a estratégia de seleção dos indivíduos, a estratégia e a taxa de recombinação, a estratégia e a taxa de mutação, o tamanho da população e a quantidade de avaliações.

2.3.4 SPEA2

O SPEA2, proposto em (ZITZLER et al., 2001), é uma evolução do SPEA - *Strength Pareto Evolutionary Algorithm* (ZITZLER e THIELE, 1999).

Uma população e um arquivo externo (archive) são utilizados para armazenar os indivíduos. Inicialmente, todos os indivíduos que não são dominados são copiados para o arquivo (ZITZLER et al., 2001). Caso a quantidade de soluções não dominadas supere o número de soluções permitidas no arquivo externo, um método é utilizado para truncar essas soluções, ou seja, excluir algumas soluções de modo a respeitar o limite de tamanho do arquivo. Neste método para truncar, os limites das extremidades são respeitados, com o objetivo de não diminuir o espaço de busca.

Um valor chamado *strength*, ou *S(i)* é associado para cada indivíduo , tanto para a população regular quanto para o arquivo externo. Este valor é igual ao número de indivíduos na população que são dominados ou são iguais ao indivíduo selecionado.

O processo de seleção envolve apenas indivíduos do arquivo externo. Posteriormente, os operadores de recombinação e mutação são aplicados.

Os parâmetros a serem setados para o SPEA2 são: a estratégia de seleção dos indivíduos, a estratégia e a taxa de recombinação, a estratégia e a taxa de mutação, o tamanho da população e a quantidade de avaliações.

2.4 Search-Based Software Engineering

A engenharia de software possui problemas de solução difícil, com muitas soluções possíveis (espaço de busca grande), soluções não conhecidas e muitas vezes com objetivos conflitantes. Para esses problemas, soluções ótimas são impossíveis ou é impraticável tentar encontrá-las (HARMAN e JONES, 2001).

Contudo, uma nova área da engenharia de software está emergindo e vem ao encontro dessa necessidade de respostas boas e aceitáveis para os problemas da engenharia de software. Esta area está sendo chamada de SBSE: *Search-Based Software Engineering* (HARMAN e JONES, 2001), ou Engenharia de Software Baseada em Buscas.

A área SBSE consiste em aplicar técnicas de otimização aos problemas da engenharia de software, com o objetivo de gerar soluções próximo de ótimas. Para que isso seja possível, é necessário que o problema seja formulado matematicamente.

Conforme dito no Capítulo 1, várias subáreas da engenharia de software têm utilizado técnicas de otimização para resolver seus problemas. Alguns exemplos são: engenharia de requisitos (BAGNALL et al., 2001), análise e projeto (LI et al., 2010), codificação (KUPERBERG e OMRI, 2009), teste de software (LI et al., 2007), manutenção (SCHMIDT, 2009), gerência de projetos (DI PENTA et al., 2007) e métricas (ANTONIOL et al., 2009). Porém, a área de teste de software possui a maioria dos trabalhos de pesquisa: cerca de 59% dos trabalhos (HARMAN et al., 2009).

Apesar de o nome *Search-Based Software Engineering* ter surgido em 2001, a idéia de aplicar técnicas de otimização a problemas da engenharia de software ocorreu bem antes desta época. O primeiro trabalho dessa área aconteceu em 1976 (MILLER e SPOONER, 1976), e consistia em gerar dados de teste. Além desse, haviam outros trabalhos aplicando otimização à engenharia de software, principalmente na área de teste de software. Porém, foi após a publicação de (HARMAN e JONES, 2001) que houve um explosão de trabalhos na área. A FIGURA 10 mostra a evolução do número de publicações na área SBSE.

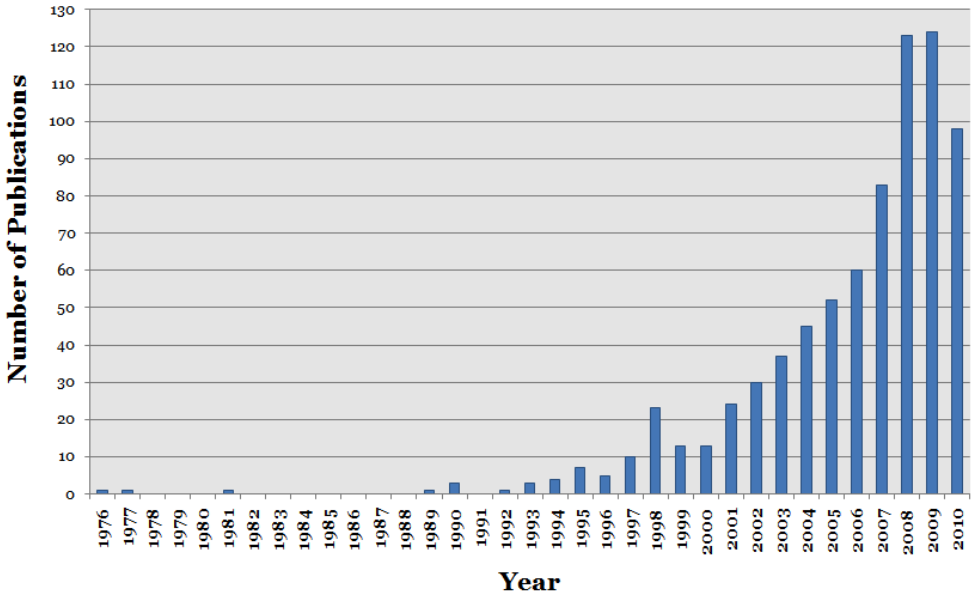

FIGURA 10 – Evolução de publicações em SBSE (SBSE Repository, 2011)

Dentro da área de teste de software, há vários trabalhos relacionados à geração de dados de teste (DAVIES et al., 1994), seleção de casos de teste (YOO e HARMAN, 2007) (MANSOUR et al., 2001) (MAIA et al., 2009), priorização de casos de teste (WALCOTT et al., 2006) (LI et al., 2007) (MAIA et al., 2010), testes não funcionais (ALANDER et al., 1997) (BRIAND et al., 2006), minimização de suíte de testes (TALLAM e GUPTA, 2006), dentre outros.

Por ser difícil encontrar soluções para esses problemas, a engenharia de software é uma área favorável ao uso de metaheurísticas. Porém, antes de aplicar as metaheurísticas para a resolução dos problemas de engenharia de software, é necessário formular estes problemas como um problema baseado em busca. Para que isso ocorra, deve-se definir (HARMAN e JONES, 2001):

a) Uma representação para o problema, a fim de possibilitar a manipulação dos dados durante a aplicação da metaheurística escolhida.

b) Uma ou mais funções objetivo, que representam os objetivos que serão maximizados ou minimizados.

c) Um conjunto de operadores que serão utilizados nas técnicas de otimização. Um exemplo seria o operador de mutação para os algoritmos genéticos.

Uma das principais técnicas de otimização aplicadas é a metaheurística, que pode ser aplicada tanto a problemas mono-objetivos como multi-objetivos. Algoritmos de otimização exata, como programação linear, geralmente não são aplicáveis aos problemas de engenharia de software, porque estes problemas possuem funções objetivo que não são caracterizadas por equações lineares, pois muitas vezes possuem mais de um objetivo e funções complexas (HARMAN, 2007), além do espaço de busca grande.

2.5 Considerações Finais

O presente capítulo apresentou uma breve revisão teórica relativa aos principais conceitos de teste de software, otimização, metaheurísticas e *Search-Based Software Engineering*. De maneira geral, os conceitos apresentados ressaltam a importância da área de teste de software e a emergente *Search-Based Software Engineering*.

A presente pesquisa é aplicada na fase de planejamento de testes, para o nível de teste de sistema, com o teste sendo realizado de forma caixa preta. Serão utilizadas três metaheurísticas multi-objetivo para implementar a abordagem: NSGA-II, MOCell e SPEA2. O motivo da escolha destes três algoritmos se deve ao fato de os mesmos serem bastante utilizados nos trabalhos em SBSE.

No capítulo seguinte serão apresentados os trabalhos relacionados a esta pesquisa.

3 TRABALHOS RELACIONADOS

O presente capítulo descreve alguns trabalhos relacionados referentes aos problemas de seleção, priorização e alocação de casos de teste, principalmente no contexto da SBSE.

Serão relacionadas algumas abordagens utilizadas para tratar os três problemas definidos anteriormente. Por último, a Seção 3.2 descreve as considerações finais deste capítulo.

3.1 Abordagem dos Problemas

3.1.1 Seleção de Casos de Teste

Os autores de (MANSOUR et al., 2001) comparam cinco algoritmos para o problema de seleção de casos de teste: Têmpera Simulada, Reduction (HARROLD et al., 1993), Slicing (AGRAWAL et al., 1993), Dataflow (GUPTA et al., 1996), e Firewall (MANSOUR et al., 2001). A função objetivo utilizada para a comparação é baseada na cobertura de código. A comparação é baseada em oito critérios quantitativos e qualitativos: número de casos de teste, tempo de execução, precisão, inclusividade, processamento dos requisitos, tipo de manutenção, nível de teste e tipo de abordagem. Os algoritmos foram executados para 15 programas, que possuíam de 21 a 381 linhas de código. Não houve conclusão de qual algoritmo é o melhor, visto que cada critério utilizado na comparação teve algoritmos diferentes como sendo o melhor. Os autores sugerem que a escolha da melhor solução depende de qual critério deve ser priorizado.

A maior contribuição do trabalho (YOO e HARMAN, 2007) é que o mesmo foi o primeiro a tratar o problema de seleção de casos de teste como um problema multi-objetivo. Os autores propõem duas aborgadens: a primeira considera cobertura de código e custo como funções objetivo, e a segunda considera cobertura de

código, custo e histórico de detecção de falhas dos casos de teste como funções objetivo. Além da formulação multi-objetivo para o problema, os autores implementam três algoritmos: uma reformulação do Algoritmo Guloso, o NSGA-II (*Non Dominating Sorting Genetic Algorithm*), e uma variante do NSGA-II, chamada vNSGA-II, com o objetivo de comparar o algoritmo guloso com as técnicas metaheurísticas. Cinco programas foram utilizados nos experimentos: uma parte dos programas da suite Siemens (printtokens, printtokens2, schedule e schedule2), considerados programas pequenos, e o programa space, da European Space Agency, o maior dos programas. Para os programas pequenos, o NSGA-II obteve o melhor desempenho, seguido pelo algoritmo vNSGA-II. Para os programas maiores, o algoritmo guloso teve o melhor desempenho.

Uma extensão deste trabalho pode ser observada em (YOO e HARMAN, 2010), onde as mesmas representações do problema (com dois e três objetivos) são utilizadas. Porém, o trabalho trata de minimização de suíte de testes, ou seja, são selecionados casos de teste considerados redundantes que devem ser retirados da suíte de testes. Os autores apresentam dois algoritmos que foram utilizados para resolver o problema: uma reformulação do algoritmo guloso mono-objetivo e uma variante híbrida do NSGA-II, chamada HNSGA-II. No algoritmo HNSGA-II, a população inicial é gerada por um algoritmo chamado Guloso Adicional, ou seja, um elemento é adicionado à solução quando faz o valor da função objetivo ser mais otimizado naquele momento. Esta população inicial é então entrada para o algoritmo NSGA-II, que executa normalmente. Nos experimentos, foram utilizados 5 programas do "mundo real": space, da European Space Agency, e 4 programas de código aberto (flex, grep, gzip e sed). Para os programas flex e gzip, as soluções foram melhoradas devido ao algoritmo HNSGA-II, que também obteve melhor desempenho que o algoritmo guloso mono-objetivo.

Em (MAIA et al., 2009), foi apresentada uma nova abordagem multi-objetivo para o problema de seleção de casos de teste. Uma das contribuições deste trabalho é considerar a participação do cliente ao selecionar os casos de teste, visto que uma das funções a serem otimizadas é a importância do caso de teste para o cliente. As

funções objetivo otimizadas neste trabalho são o tempo total de execução (a ser minimizada), o risco dos casos de teste (a ser minimizado) e a importância dos casos de testes (a ser maximiza). Como restrições, são levados em consideração o tempo de execução dos casos de teste selecionados e o tempo disponível para testes de cada testador. O algoritmo NSGA-II foi comparado apenas com um algoritmo randômico, obtendo melhor desempenho.

3.1.2 Priorização de Casos de Teste

O trabalho (ELBAUM et al., 2004) analisa algumas técnicas de priorização não baseadas em otimização, tentando determinar qual técnica é melhor para cenários específicos e suas condições. As técnicas investigadas são: *total function coverage prioritization* (*total*), a*dditional function coverage priotitization* (*addtl*), *total binary-diff function coverage prioritization* (*total-diff*) e *additional binary-diff function coverage prioritization* (*addtl-diff*). A técnica *total* ordena os casos de teste de acordo com o valor da função de cobertura, enquanto a *total-diff* ordena segundo o valor de cobertura das funções que estão diferentes em relação ao código. A técnica *addtl* adiciona a cada iteração o caso de teste com melhor cobertura, e os casos de teste ainda não selecionados têm sua informação de cobertura atualizada, de modo a computar somente a cobertura ainda não atingida pela solução (funções ainda não cobertas pela solução). Finalmente, a técnica *addtl-diff* é semelhante à técnica *addtl*, porém a função para ordenação é a cobertura em relação às funções ainda não cobertas pela solução.

Uma heurística gulosa foi utilizada apenas para calcular uma solução ótima para a métrica APFD (*Average Percentage of Fault Detected*), a fim de comparar com as soluções encontradas pelas técnicas selecionadas. Todas as técnicas foram aplicadas a cada um dos 8 programas utilizados nos experimentos, bem como um algoritmo randômico. As técnicas que possuem feedback (additional function coverage prioritization e additional binary-diff function coverage prioritization), ou seja, as que atualizam a cobertura dos casos de teste ainda não selecionados baseado nos já selecionados, obtiveram melhor desempenho, às vezes encontrando a solução ótima.

As técnicas total function coverage prioritization e total binary-diff function coverage prioritization foram iguais ou piores que o algoritmo randômico.

O trabalho (ROTHERMEL et al., 2001) foi utilizado como base para muitos dos trabalhos relacionados à priorização de casos de teste. O objetivo deste trabalho é investigar se a priorização dos casos de teste aumenta a probabilidade de detectar erros no início dos testes. Oito técnicas de priorização de casos de teste foram utilizadas no experimento, sendo implementadas utilizando algoritmos gulosos. As suítes de teste priorizadas por estas técnicas foram comparadas com suítes de teste não priorizadas. As técnicas são: "*random prioritization"*, onde os casos de teste são priorizados randomicamente, "*optimal prioritization"*, onde os casos de teste são ordenados de acordo com sua taxa de detecção de falhas, "*total statement coverage* prioritization", onde os casos de teste são priorizados de acordo com a cobertura de comandos, "*additional statement coverage prioritization"*, onde os casos de teste são ordenados de acordo com o número de comandos ainda não cobertos, "*total branch coverage prioritization"*, onde os casos de teste são ordenados de acordo com sua cobertura de *if/else,* "*additional branch coverage prioritization"*, onde os casos de teste são priorizados de acordo com a cobertura ainda não atingida de *if/else,* "*total fault-exposing-potential"*, onde a ordenação ocorre de acordo com o potentcial dos casos de teste de encontrar falhas, e "*additional fault-exposing-potential"*, que ordena os casos de teste tendo em vista seu potencial de encontrar falhas ainda não detectadas. Os autores mostraram que todas as técnicas melhoraram a taxa de detecção de falhas, justificando a aplicação de técnicas de priorização de casos de teste.

Os autores de (WALCOTT et al., 2006) propuseram uma técnica de priorização de casos de teste baseada em algoritmo genético que reordena a suíte de teste considerando dois objetivos: restrições de tempo para testes e cobertura de código. A função objetivo implementa a técnica Average Percentage Block Coverage (APBC). A técnica proposta foi comparada com as seguintes técnicas: ordem inicial, onde os casos de teste são ordenados de acordo com a ordem em que foram escritos, ordem reversa, onde os casos de teste são ordenados de forma descendente de acordo com a ordem em que foram escritos, e *fault-aware*, onde os casos de teste são

ordenados de acordo com a função APBC, até que o tempo limite para execução seja atingido. A técnica proposta pelos autores obteve melhor desempenho que as outras técnicas para um estudo de caso. Para outro estudo de caso, este fato não ocorreu porque os casos de teste deste estudo de caso eram mais intercambiáveis (trocados facilmente por outros). Segundo os autores, é mais fácil para os métodos de priorização randômicos ter valores de APFD maiores à medida que mais casos de teste da suíte original possam ser executados.

Em (LI et al., 2007), os autores comparam cinco algoritmos para o problema da priorização de casos de teste: Algoritmo Guloso, Additional Greedy, 2-Optimal, Hill Climbing, e Algoritmo Genético. Os autores separaram as suítes de teste em pequenas (8 a 155 casos de teste) e grandes (228 a 4.350 casos de teste). Seis programas em C foram usados nos experimentos. A análise foi realizada separadamente para programas pequenos e grandes. Como função objetivo, foram consideradas as seguintes métricas de cobertura, separadamente (três abordagens mono-objetivo): *Average Percentage Block Coverage* (APBC), *Average Percentage Decision Coverage* (APDC) e *Average Percentage Statement Coverage* (APSC). Para os programas pequenos, o algoritmo genético foi melhor que os demais, e o Algoritmo Guloso obteve o pior desempenho. O algoritmo Additional Greedy teve o segundo melhor desempenho, e os autores mostraram que a diferença entre ele e o Algoritmo Guloso, que foi o melhor, não é significante (calculado pelo *t-test*). Para os programas grandes, os dois melhores algoritmos foram Additional Greedy e 2-Optimal, porém não houve diferença significativa entre eles.

Baseado neste trabalho, em (MAIA et al., 2008), os autores implementaram a metaheurítica GRASP Reativo para o problema da priorização de casos de teste. A metaheurística proposta foi então comparada com o Algoritmo Guloso, Additional Greedy e o Algoritmo Genético, em termos de tempo e desempenho, e foram utilizados nos experimentos todos os programas pequenos utilizados em (LI et al., 2007). Para a criação das instâncias, a técnica utilizada foi a seleção dos casos de teste dos programas, realizada de forma randômica, e considerando percentuais diferentes de quantidade e casos de teste: 1%, 2%, 3%, 5%, 10%, 20%, 30%, 40%, 50%, 60%,

70%, 80%, 90% e 100%. Por exemplo, se o percentual considerado for 5%, então, randomicamente, 5% dos casos de teste da suíte serão considerados para a comparação entre os algoritmos. O algoritmo Additional Greedy, chamado neste trabalho de Algoritmo Guloso, obteve o melhor desempenho, de modo semelhante ao trabalho de (LI et al., 2007), seguido do algoritmo GRASP Reativo, porém sem diferença significante. O Algoritmo Genético obteve o pior desempenho, resultado diferente do reproduzido em (LI et al., 2007). Uma desvantagem do GRASP Reativo foi o seu tempo de execução, sendo o mais lento dos algoritmos.

3.1.3 Alocação de Casos de Teste

Existem diversos trabalhos que tratam de alocação de tarefas utilizando-se de técnicas de otimização, porém não tratam especificamente de alocação de casos de teste para testadores. Há alguns trabalhos que tratam de alocação de recursos para a execução de testes, porém esses recursos englobam outros fatores além de mão de obra, como tempo de CPU e número de casos de teste. Serão listados aqui alguns trabalhos relacionados à alocação de tarefas e de recursos.

Em (DUGGAN et al., 2004) é descrito um otimizador de alocação de tarefas para construção de software (apenas para a fase de implementação). Algoritmos genéticos e evolucionários são utilizados para encontrar possíveis soluções. As entradas para o otimizador são as características do projeto e as experiências de cada desenvolvedor.

(ANTONIOL et al., 2005) aplica Algoritmos Genéticos, Hill Climbing e Têmpera Simulada em duas representações diferentes para o problema de alocação de recursos. As alocações de “pacotes de tarefas” são feitas para um grupo de trabalho, e não para um indivíduo. O Algoritmo Genético gerou as melhores soluções.

No trabalho (DI PENTA et al., 2007), os autores exploram o impacto da Lei de Brooks (BROOKS, 1975) sobre as alocações nos projetos, utilizando dados de projetos reais nos experimentos. Uma abordagem baseada em algoritmos genéticos é proposta para estudar o efeito do *overhead* de comunicação da equipe do projeto no tempo de conclusão do projeto e na alocação das pessoas nas equipes. Os resultados

dos experimentos confirmaram que a abordagem utilizada na alocação das pessoas pode aumentar o *overhead* de comunicação, fazendo com que o projeto atrase. Além disso, o Algoritmo Genético implementado mostrou como soluções várias equipes pequenas (com poucas pessoas) ao invés de equipes maiores. Com menos pessoas alocadas nas equipes e tarefas mais paralelizadas, o *overhead* de comunicação fica baixo e o risco de atraso no projeto é menor.

Em (BARRETO et al., 2008) os autores levam em consideração as características das atividades de um projeto, os recursos humanos disponíveis e as restrições para a resolução do problema de alocação de tarefas, definidas pela empresa desenvolvedora do software. O problema é modelado como um constraint satisfaction problem, e a implementação do algoritmo proposto envolve *backtracking* (BITNER e REINGOLD, 1975), *forward checking* (HARALICK e ELLIOTT, 1980) e *branch and bound* (LAWLER e WOOD, 1966) . Em uma ferramenta criada pelos autores, o gerente de projeto seleciona qual função ele quer otimizar (maximizar ou minimizar). Os experimentos indicaram bom desempenho da abordagem proposta.

Uma nova abordagem para o problema de alocação de tarefas é apresentado em (DI PENTA et al., 2009), baseada em dependências, fragmentação, conflitos e especialização das tarefas. Os algoritmos implementados para a avaliação da solução foram: Têmpera Simulada, Algoritmo Genético mono e multi-objetivo e *Hill Climbing*. Os dados utilizados nos experimentos são de projetos reais. Os autores buscam minimizar o tempo de conclusão das tarefas e também sua fragmentação, respeitando as dependências (executadas pelo mesmo time) e experiência das pessoas envolvidas. Os algoritmos Têmpera Simulada e Algoritmo Genético obtiveram melhor desempenho que os demais. Um teste de sensibilidade foi realizado, aumentando o número de pessoas alocadas ao projeto, e observou-se que, à medida que o número de pessoas crescia, o tempo para completar as tarefas era reduzido.

No trabalho (HOU et al., 1996), os autores investigam dois problemas de alocação para testes de módulo (testes unitários): minimização do número de falhas ainda não detectadas no sistema após a alocação de recursos e minimização do total de

recursos alocados dado um número de falhas não detectadas após os testes. Esses recursos incluem mão de obra humana, número de casos de teste, tempo de CPU, dentre outros. Dois algoritmos (heurísticas) são propostos para a resolução destes problemas.

Finalmente, o trabalho (DAI et al., 2003) também considera vários tipos de recursos a serem alocados para os testes, não somente as pessoas. Os autores propõem um algoritmo genético para a alocação de recursos para os testes de sistemas modulares.

3.1.4 Resumo dos Trabalhos Relecionados

A TABELA 1 mostra, de forma resumida, os trabalhos relacionados descritos na seção anterior.

TABELA 1 – Trabalhos Relacionados

PROBLEMA	REFERÊNCIA	ALGORITMOS	ABORDAGEM	RESULTADO
Seleção	(MANSOUR et al., 2001)	1. Têmpera Simulada 2. Reduction 3. Slicing 4. Dataflow 5. Firewall	Cobertura de código	Não houve conclusão de qual algoritmo é melhor. Depende do critério.
Seleção	(YOO e HARMAN, 2007)	1. Algoritmo Guloso 2. NSGA-II 3. vNSGA-II	1. cobertura de código + custo 2. cobertura de código + custo + histórico de detecção de falhas	Melhor algoritmo: 1. Programas pequenos: NSGA-II 2. Programas maiores: Algoritmo Guloso
Seleção	(YOO e HARMAN, 2010)	1. Reformulação do Algoritmo Guloso Mono-Objetivo 2. HNSGA-II	1. cobertura de código + custo 2. cobertura de código + custo + histórico de detecção de falhas	Para dois programas (dos cinco), o HNSGA-II obteve melhores soluções.
Seleção	(MAIA et al., 2009)	1. NSGA-II 2. Randômico	Tempo de de execução, risco do caso de teste e importância do caso de teste	NSGA-II obteve melhor desempenho que o Randômico
Priorização	(ELBAUM et al., 2004)	Heurística gulosa (apenas para calcular	APFD	A comparação foi realizada entre técnicas

		uma solução ótima)		que não são baseadas em busca
Priorização	(ROTHERMEL et al., 2001)	-	-	O objetivo do trabalho é avaliar se técnicas de priorização melhoram a taxa de detecção de falhas
Priorização	(WALCOTT et al., 2006)	Algoritmo Genético	APBC	Algoritmo Genético foi melhor, comparado com outras técnicas de priorização não baseadas em busca
Priorização	(LI et al., 2007)	1. Algoritmo Guloso 2. Additional Greedy 3. 2-Optimal 4. Hill Climbing 5. Algoritmo Genético	1. APBC 2. APDC 3. APSC	Melhor algoritmo: 1. Programas pequenos: Algoritmo Genético 2. Programas maiores: Additional Greedy e 2-Optimal
Priorização	(MAIA et al., 2008)	1. GRASP Reativo 2. Algoritmo Guloso 3. Additional Greedy 4. Algoritmo Genético	1. APBC 2. APDC 3. APSC	O melhor desempenho foi obtido pelo Additional Greedy, seguido do GRASP Reativo
Alocação	(DUGGAN et al., 2004)	Algoritmos genéticos e evolucionários	Não especificado com detalhes	Resultado satisfatório para a aplicação destes algoritmos
Alocação	(ANTONIOL et al., 2005)	1. Algoritmos Genéticos 2. Hill Climbing 3. Têmpera Simulada	Alocações de pacotes de tarefas para grupos de trabalho	O Algoritmo Genético obteve melhor desempenho
Alocação	(BARRETO et al., 2008)	Algoritmo envolvendo *backtracking, forward checking* e *brach and bound*.	O gerente de projeto seleciona na ferramenta qual função deseja otimizar	Boa performance da abordagem proposta
Alocação	(DI PENTA et al, 2009)	1. Têmpera	Tempo de	Têmpera

		Simulada 2. Algoritmo Genético mono-objetivo 3. Algoritmo genético multi-objetivo 4. Hill Climbing	conclusão das tarefas + fragmentação	Simulada e Algoritmo Genético obtiveram melhores resultados
Alocação	(HOU et al., 1996)	Duas heurísticas foram propostas	Número de falhas ainda não detectadas + total de recursos	Boa performance dos algoritmos
Alocação	(DAI et al., 2003)	Algoritmo Genético	Não detalhado	Boa performance do algoritmo

3.2 Considerações Finais

Este capítulo apresentou algumas das principais contribuições encontradas na literatura para os problemas de seleção, priorização e alocação de casos de teste, no contexto da SBSE.

Em geral, para os problemas de seleção e priorização de casos de teste, a maioria dos trabalhos utiliza a cobertura de código como função objetivo, ou seja, é necessário conhecer todo o código do sistema, e seu relacionamento com os casos de teste. É mais fácil que isso ocorra para os testes unitários, porém, para as pessoas envolvidas nos testes de sistema, nem sempre é possível conhecer esse código, seja por conta do processo da empresa, o tempo disponível para os testes ou simplesmente a metodologia utilizada pela equipe de testes. Usualmente, é necessário conhecer apenas as características do sistema para testá-lo. A abordagem proposta nesta dissertação considera como objetivos funções que dependem de dados dos requisitos e casos de teste, não dependendo do código da aplicação.

Um outro ponto que pode ser observado em praticamente todos os trabalhos listados é que poucos consideram a opinião do cliente, já que deixam de considerar alguns fatores importantes para o mesmo, como a importância de uma funcionalidade. O motivo para esta realidade pode ser a necessidade de automação do processo de testes. Geralmente, somente o custo é considerado. Na abordagem proposta,

consideramos a importância dada pelo cliente a cada requisito do sistema, no processo de seleção de casos de teste.

Nenhum dos trabalhos citados realiza a comparação do desempenho das técnicas propostas com respostas de especialistas (humanos). O trabalho (SOUZA et al., 2010) realiza uma análise de competitividade humana de alguns trabalhos já publicados na área de SBSE, porém só inclui o problema de seleção de casos de teste, dentre os três problemas envolvidos nesta pesquisa. Neste trabalho, fizemos uma comparação das soluções encontradas pelas técnicas de otimização com as soluções encontradas por alguns engenheiros de software.

Em relação ao problema de alocação de casos de teste, conforme dito anteriormente, não há trabalhos publicados na área de SBSE. Há diversos trabalhos referentes a alocação de tarefas, mas nenhum específico para a atividade de teste de software. Além disso, os trabalhos que envolvem alocação de recursos para testes, dentre esses recursos a alocação de pessoas, são muito pouco detalhados. Este é o primeiro trabalho na área de SBSE que considera a alocação de casos de teste para execução.

Finalmente, não há trabalhos publicados na área de SBSE envolvendo os três problemas (seleção, priorização e alocação de casos de teste) de forma integrada, sendo isto uma contribuição desta dissertação.

4 ABORDAGEM INTEGRADA DOS PROBLEMAS DE SELEÇÃO, PRIORIZAÇÃO E ALOCAÇÃO DE CASOS DE TESTE

Neste capítulo é apresentada a formulação para os três seguintes problemas de teste de software, de forma integrada: seleção, priorização e alocação de casos de teste.

4.1 Visão Geral da Abordagem Integrada e dos Problemas

O diagrama exibido na FIGURA 11 apresenta, de forma gráfica, a abordagem proposta. A abordagem consiste em selecionar, priorizar e alocar casos de teste, de forma integrada, interativa e multi-objetiva.

A saída da primeira fase é um conjunto de soluções, onde cada solução possui um subconjunto de casos de teste, selecionados para a execução. O usuário seleciona uma das soluções sugeridas na fase 1, e então o processo de ordenação dos casos de teste, a fase 2, é iniciado. A saída da segunda fase é um conjunto de soluções, onde cada solução contém os mesmos casos de teste selecionados na fase 1, porém ordenados para execução. Novamente, o usuário seleciona uma das soluções sugeridas, e a fase 3 inicia. A saída desta última fase é um conjunto de soluções, onde cada solução contém a alocação dos casos de teste que já foram selecionados e priorizados, ou seja, conterá a informação de quem testará o que.

A seguir, cada uma das fases é detalhada.

FASE 1) Seleção de Casos de Testes

Consiste em selecionar quais casos de teste serão executados. A abordagem leva em consideração a importância dos requisitos do sistema para o cliente, o tempo máximo permitido para a execução dos testes e a precedência dos casos de teste. A seleção dos requisitos é baseada na visão do cliente, visto que o valor de importância dado a cada requisito é atribuído pelo cliente.

Para realizar a seleção dos casos de teste, os seguintes objetivos serão almejados pelas técnicas de otimização:

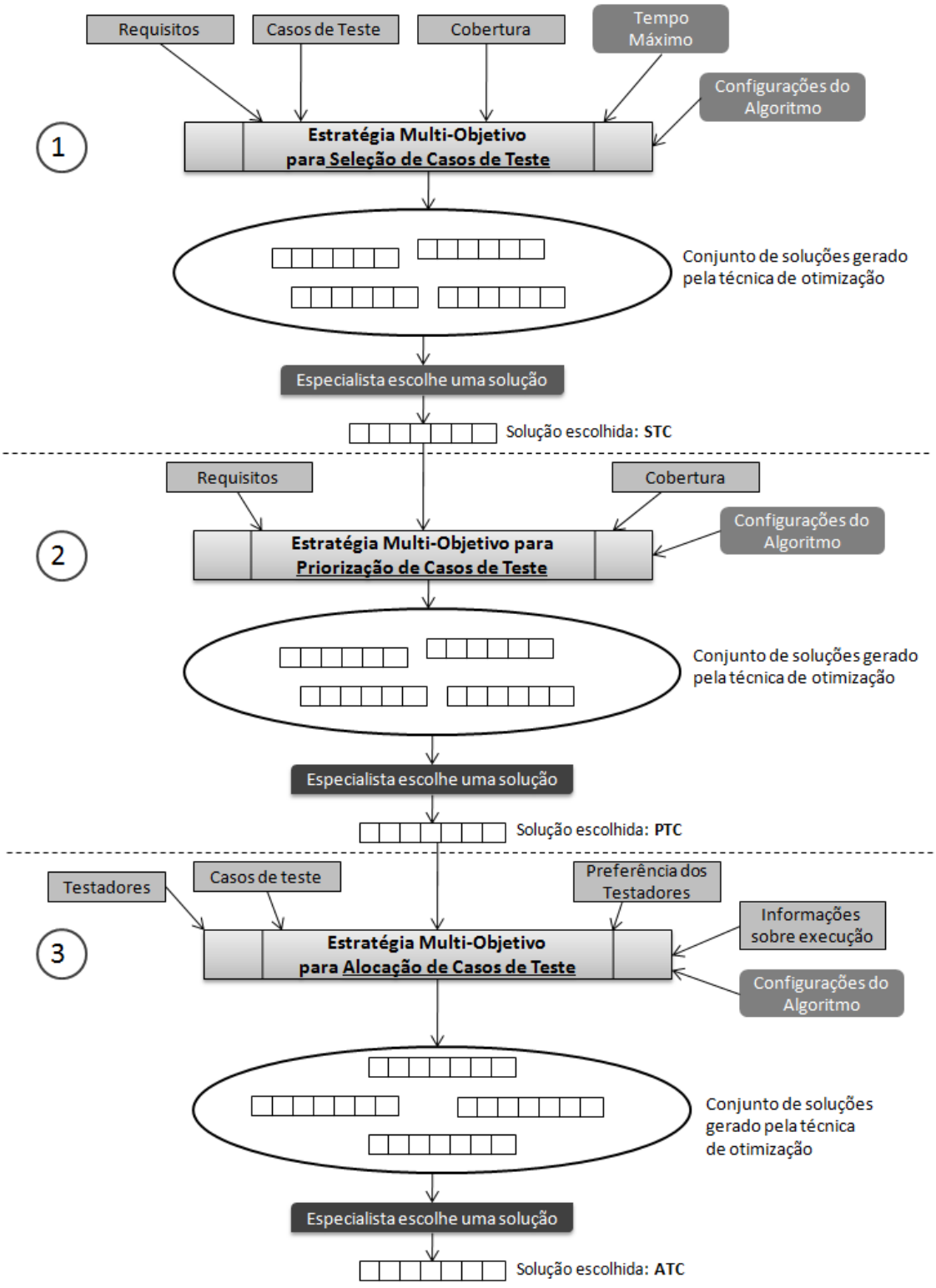

FIGURA 11 – Integração dos três problemas

1) Selecionar um subconjunto de casos de teste, de modo que seja maior a quantidade de requisitos cobertos, visto que quanto mais requisitos forem testados, melhor;

2) Selecionar os casos de teste que cobrem os requisitos mais importantes, ou seja, testar os requisitos que têm maior valor para o negócio do cliente.

Além disso, a soma do tempo de execução dos casos de teste selecionados não deve exceder o tempo máximo permitido para a execução dos testes, e as precedências dos casos de teste devem ser respeitadas, ou seja, caso exista um caso de teste que possua um precedente, este precedente deve ser selecionado também.

O resultado final desta fase é um conjunto de soluções, sendo que cada solução é composta por um conjunto de casos de teste. Estas soluções são apresentadas a um especialista para que ele possa escolher uma das soluções. A solução escolhida será utilizada como entrada para a segunda fase do processo.

FASE 2) Priorização de Casos de Testes

Consiste em ordenar os casos de teste que serão executados. A abordagem leva em consideração a volatilidade e a complexidade dos requisitos, além do conjunto de casos de teste selecionados na fase anterior. Nesta fase, a ordenação é realizada baseada na visão da empresa desenvolvedora, que atribui os valores de volatilidade e complexidade dos requisitos.

Para a ordenação dos casos de teste, os seguintes objetivos serão almejados pelas técnicas de otimização:

1) Antecipar a execução dos casos de teste que cobrem os requisitos mais voláteis, visto que quanto maior a volatilidade do requisito maior a probabilidade de haver erros ou esquecimentos em sua implementação;

2) Antecipar a execução dos casos de teste que cobrem os requisitos mais complexos, visto que quanto mais complexa é a implementação de um requisito maior a probabilidade de haver erros em sua implementação.

Adicionalmente, existe uma restrição de precedência dos casos de teste, ou seja, caso exista um caso de teste x que possua um precedente y, este precedente y deve ser executado obrigatoriamente antes de x.

O resultado final desta fase é um conjunto de soluções, sendo que cada solução é composta por um conjunto de casos de teste já ordenados para a execução. Estas soluções são apresentadas a um especialista para que ele possa escolher uma das soluções. A solução escolhida será utilizada como entrada para a terceira fase do processo.

FASE 3) Alocação de Casos de Testes

Consiste em distribuir os casos de teste que serão executados para os testadores disponíveis. A abordagem leva em consideração a habilidade dos testadores, a preferência dos testadores pelos casos de teste, o histórico de execução de testes (quem testou quais casos de teste), a habilidade mínima necessária para executar cada caso de teste e o conjunto de casos de teste já ordenados para execução.

Para alocar os casos de teste para os testadores, os seguintes objetivos serão almejados pelas técnicas de otimização:

1) Atribuir a cada testador casos de teste que ele executou menos vezes, visto que quanto mais vezes um testador executa um caso de teste manualmente, maior a probabilidade de haver vícios ou falta de atenção durante a execução;

2) Atribuir a cada testador os casos de teste que ele prefere executar;

3) Maximizar a habilidade do testador alocado para executar cada caso de teste. Caso um caso de teste seja alocado para um testador que não possua a habilidade mínima exigida para sua execução, uma penalidade deve ser aplicada, a fim de diminuir o valor deste objetivo na solução.

Além disso, a restrição de precedência dos casos de teste deve ser respeitada, ou seja, caso exista um caso de teste x que possua um precedente y, este precedente y deve ser executado antes pelo mesmo testador que executar x.

O resultado final desta fase é um conjunto de soluções, sendo que cada solução é composta por um conjunto de casos de teste já alocados para os testadores. Estas soluções são apresentadas a um especialista para que ele possa escolher uma das soluções.

4.2 Formalização dos Problemas

A seguir são apresentadas algumas definições formais que permitirão a formulação matemática da abordagem proposta.

4.2.1 Requisitos

Requisitos são funcionalidades e regras de negócio que foram implementadas ou alteradas, e devem ser testadas, de modo a garantir que o sistema está funcionando como deveria.

Seja R o conjunto de requisitos de um sistema. O conjunto R contém N requisitos, ou seja, $R = \{r_i \mid i = 1, \dots, N\}$, e cada requisito possui os seguintes atributos:

- $importance_i$ [1]- Importância do requisito i. Um valor de importância é associado pelo cliente a cada requisito, representando sua importância para o negócio e recebe um valor entre 1 e 100.

- $volatility_i$ – Volatilidade do requisito i. O valor de volatilidade de cada requisito é atribuído pela equipe de desenvolvimento, e representa a variação que os requisitos sofreram desde o início de sua existência, devido a fatores externos, imaturidade do cliente ou à inexperiência dos analistas de requisitos que detalharam os requisitos. Nesta abordagem, considerou-se valores entre 1 e 100, porém, quando não for possível atribuir valores para esta métrica, pode-se utilizar como métrica o número de vezes que o requisito foi alterado, pois é um dado que

[1] Os termos da definição formal da abordagem estão em inglês para facilitar a divulgação da abordagem em fóruns internacionais.

geralmente as empresas armazenam. Valores maiores para a volatilidade representam mais modificações no requisito.

- $complexity_i$ – Complexidade do requisito i. O valor de complexidade também é atribuído pela equipe de desenvolvimento, e representa o quão complexa foi a implementação de um requisito. A complexidade é maior quando são utilizadas novas tecnologias, quando a regra de negócio do cliente é complexa ou quando é necessário montar uma estrutura complexa para implementar e testar o requisito, dentre outros fatores. Valores maiores representam complexidade maior do requisito. Nesta aboradgem, considerou-se valores entre 1 e 100, porém, quando não for possível atribuir valores para esta métrica, pode-se utilizar como métrica o tamanho do requisito em pontos de função (ALBRECHT, 1979), ou outra métrica de medição de complexidade que seja utilizada pela empresa desenvolvedora.

4.2.2 Casos de Teste

Os casos de teste representam as condições de um teste a ser realizado, conforme definido anteriormente. Ele contém o procedimento que deve ser realizado para que o teste seja executado, as entradas necessárias para o teste e a saída esperada, dentre outras informações.

Seja C o conjunto de todos os casos de teste do sistema. C possui M casos de teste, ou seja, $C = \{ c_j \mid j = 1, \dots, M \}$.

Cada caso de teste possui os seguintes atributos:

- $requiredExperience_j$ – Experiência requerida do testador para que o mesmo possa executar o caso de teste j. Um valor entre 1 e 100 representa o nível de experiência mínima requerido do testador. Quanto maior o valor para este atributo, maior é a experiência requerida para executar o caso de teste.

- *executionTime*$_j$ – Tempo de execução do caso de teste *j*, em milisegundos. O tempo total necessário para que o caso de teste possa ser executado por um testador humano (teste manual). Caso seja um novo caso de teste, o tempo de execução do mesmo é estimado pela equipe responsável pelos testes do sistema. Se o caso de teste não for novo, pode-se utilizar o histórico de execuções de teste para descobrir seu tempo de execução.

- *precedence*$_j$ – Caso de teste que deve ser executado antes do caso de teste *j*, ou seja, seu precedente. Esta abordagem considera que um caso de teste possui no máximo um precedente.

- *coverage*$_j$ – Cobertura do caso de teste *j*. A cobertura de um caso de teste é o conjunto de requisitos que são testados por este caso de teste. Nesta abordagem, um requisito pode ser testado por um ou mais casos de teste, porém um caso de teste testa apenas um requisito.

4.2.3 Testadores

Testadores são as pessoas que testam a aplicação, ou seja, as pessoas que executam os casos de teste a fim de verificar se os requisitos foram implementados da forma correta.

Seja *T* o conjunto de todos os testadores alocados para executar os casos de teste selecionados. O conjunto *T* possui *P* testadores, ou seja, $T = \{t_w \mid w = 1, \dots, P\}$.

Cada testador possui o seguinte atributo:

- *experience*$_w$ – Experiência do testador *w*. Um valor entre 1 e 100 que representa o nível de experiência do testador. Valores maiores são atribuídos a testadores mais experientes.

4.2.4 Relacionamento entre testadores e casos de teste

As informações a seguir podem ser obtidas do relacionamento entre testadores e casos de teste:

- $preference_{w,j}$ – Preferência do testador *w* pelo caso de teste *j*. Um valor entre 1 e 100 que representa a preferência do testador pelo caso de teste referente. Cada testador atribui valores de preferência para cada caso de teste.

- $numberOfExecutions_{w,j}$ – Número de execuções do caso de teste *j* pelo testador *w*. Leva em consideração todas as execuções de teste desde o início do sistema. Para isso, considera-se que a informação de quem executou cada caso de teste foi armazenada. Esta informação é opcional para a abordagem, pois quando o caso de teste é novo, não há informações sobre a execução deste caso de teste.

4.2.5 Suítes de Teste

Conforme definido anteriormente, uma suíte de testes é um conjunto de casos de teste ordenados para a execução. O tamanho da suíte de testes é o número de casos de teste que a compõem. Nesta abordagem, a suíte de testes é representada de duas formas:

- Vetor binário *STC*

  O vetor binário *STC* tem tamanho *M*, onde *M* é o número de casos de teste do sistema. Cada posição no vetor *STC* representa um caso de teste. Se o valor para uma determinada posição em *STC* for 1, o caso de teste correspondente àquela posição está selecionado. Caso o valor seja 0, o caso de teste correspondente não está selecionado. O vetor *STC* representa a solução para a fase 1 da abordagem proposta (seleção de casos de teste).

A FIGURA 12 ilustra a representação do subconjunto de casos de teste representados por *STC*. De acordo com esse exemplo, os casos de teste selecionados seriam 2, 4, 6 e 7 (valor 1 nas posições 2, 4, 6 e 7).

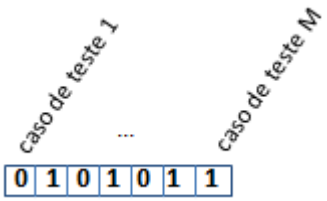

FIGURA 12 – Formato de STC

- Vetor de inteiros PTC

O tamanho do vetor *PTC* (*PTC.length*) é o número de casos de teste selecionados na fase 1 (seleção de casos de teste), e os números inteiros contidos em suas células representam o número dos casos de teste que serão executados, na respectiva ordem. O caso de teste contido na célula 1 do vetor *PTC* será executado primeiro, o caso de teste contido na célula 2 deve ser executado após o primeiro, e assim por diante. O vetor de inteiros *PTC* representa a solução da fase 2 da abordagem proposta (priorização de casos de teste). A FIGURA 13 representa o subconjunto de casos de teste *PTC*. De acordo este exemplo, a ordem de execução dos casos de teste seria: 6, 4, 2, 7.

FIGURA 13 – Formato de PTC

Cada suíte de teste possui as seguintes informações:

- $suiteCoverage_{STC}$ – Conjunto de requisitos cobertos pela suíte de testes representada pelo vetor binário *STC*. A cobertura é a união de todos os requisitos cobertos pelos casos de teste que estão selecionados em *STC*.

- $suiteImportance_{STC}$ – Importância da suíte de testes representada pelo vetor binário *STC*. A importância da suíte é a soma da importância de todos os requisitos cobertos pelos casos de teste que estão selecionados em *STC*.

- $suiteVolatility_{PTC}$ – Volatilidade da suíte de testes representada pelo vetor de inteiros *PTC*. A volatilidade da suíte é a soma ponderada da volatilidade de todos os requisitos cobertos pelos casos de teste que fazem parte de *PTC*, considerando a posição do caso de teste na suíte. Assim, uma maior volatilidade é atribuída à suíte que possui casos de teste que cobrem requisitos mais voláteis antes.

- $suiteComplexity_{PTC}$ – Complexidade da suíte de testes representada pelo vetor de inteiros *PTC*. A complexidade de uma suíte de testes é a soma da complexidade dos requisitos cobertos pelos casos de teste que fazem parte de *PTC*, considerando a posição do caso de teste na suíte. Assim, uma maior complexidade é atribuída à suíte que possui casos de teste que cobrem requisitos mais complexos antes.

- $testCasePosition_{j,PTC}$ – Posição do caso de teste *j* na suíte de testes representada por *PTC*. A posição é um número inteiro que representa a posição do caso de teste *j* na suíte de testes *PTC*. Este número pode variar entre 1 e o tamanho da suíte *PTC*.

4.2.6 Alocações

Na abordagem proposta, uma alocação é a atribuição de um caso de teste (tarefa) para um testador. Na terceira fase da abordagem, a solução é representada por um vetor de inteiros *ATC*, que é um conjunto de alocações.

Cada célula de *ATC* possui o número do testador que executará o caso de teste que está na posição correspondente no vetor *PTC*. A FIGURA 14 mostra os detalhes do conjunto de alocações *ATC*. Neste exemplo, o caso de teste da posição 1 (caso de teste 6 na suíte de testes representada por *PTC*, segundo o exemplo da FIGURA 13) deve ser executado pelo testador 3; o caso de teste da posição 2 (caso de teste 4 na suíte de testes representada por *PTC*) deve ser executado pelo testador 2, e assim por diante.

FIGURA 14 – Formato de ATC

O conjunto de alocações *ATC* possui as seguintes informações:

- $numberOfExecutions_{ATC}$ – Seja $numberOfExecutions_{w,j}$ o número de vezes que o testador *w* já testou o caso de teste *j*, como definido anteriormente. Somando-se esse valor para todos os testadores que foram alocados no projeto de testes, tem-se o valor de $numberOfExecutions_{ATC}$. Quanto menor $numberOfExecutions_{ATC}$, melhor, pois significa os testadores irão executar casos de teste que eles executaram poucas vezes ou nenhuma vez.

- $Preference_{ATC}$ – Seja $preference_{w,j}$ a preferência de um testador *w* por um caso de teste *j*, conforme definido anteriormente. Somando-se esse valor

para todos os testadores que foram alocados no projeto de testes, tem-se o valor de $preference_{ATC}$. Quanto maior $preference_{ATC}$, melhor, pois significa os testadores irão executar casos de teste que eles preferem mais.

- $Experience_{ATC}$ – Seja $experience_w$ a experiência de um testador *w* e $requiredExperience_j$ a experiência mínima requerida por um caso de teste (o testador deve possuir a experiência mínima), conforme definidos anteriormente. Somando-se esse valor para todos os testadores que foram alocados no projeto de testes, tem-se o valor de $experience_{ATC}$. Quanto maior $experience_{ATC}$, melhor, pois significa os testadores irão executar casos de teste contendo a experiência mínima requerida pelo caso de teste. Na abordagem proposta, é possível associar a um caso de teste um testador que não possui a experiência mínima requerida por ele, porém, nesses casos, uma penalidade deve ser aplicada, de modo a diminuir o valor de $experience_{ATC}$, já que não é a situação desejada. A função de penalidade é representada pela função $penalty_{w,PTC_a}$, onde *w* é o testador e PTC_a é o caso de teste *a* na suíte de testes representada por *PTC*. A função $penalty_{w,PTC_a}$ retorna um valor correspondente à diferença entre a experiência mínima exigida pelo caso de teste e a experiência do testador que foi alocado para executar este caso de teste, caso este não tenha a experiência mínima exigida. Caso tenha a experiência mínima exigida, a função retorna o valor zero, de modo a não diminuir o valor de $experience_{ATC.}$

4.3 Formulação Matemática para os Problemas

Esta seção mostra a formulação matemática integrada para os três problemas tratados nesta pesquisa.

4.3.1 Seleção de Casos de Teste

Deseja-se encontrar um vetor binário *STC* de tamanho *M*, onde *STC* representa um subconjunto de casos de testes, no sentido de atingir os objetivos OS1 e OS2 e respeitando as restrições RS1 e RS2, a seguir:

$(\mathbf{OS1})\ Max\ |suiteCoverage_{STC}|$, onde $suiteCoverage_{STC} = \{r_i \mid r_i \in coverage_{j,}\ STC_j = 1\}$

$(\mathbf{OS2})\ Max\ suiteImportance_{STC} = \sum(importance_i), \forall\ r_i \in suiteCoverage_{STC}$

Sujeito a:

$(\boldsymbol{RS1})\ \sum_{j=1}^{M} executionTime_j * STC_j \leq\ maximumTimeAllowed$

$(\boldsymbol{RS2})\ \forall\ t_{j1}\ e\ t_{j2} \in C,\ \left(\left(precedence_{t_{j2}} = \mathrm{t_{j1}}\right) \mathrm{e}\ (\mathrm{X_{j2}} = 1)\right) \rightarrow\ (\mathrm{STC_{j1}} = 1)$

Onde,

A equação (**OS1**) deseja maximizar a quantidade de requisitos cobertos pelo subconjunto de casos de testes representado por *STC*, ou seja, a cardinalidade deste subconjunto. A cobertura pode ser calculada da seguinte forma:

$$suiteCoverage_{STC} = \{r_i \mid r_i \in coverage_{j,}\ STC_j = 1\}$$

A cobertura do subconjunto de casos de teste *STC* é a união dos requisitos cobertos pelos seus casos de teste. O termo STC_j representa se um caso de teste está selecionado ou não.

A equação (**OS2**) expressa a importância de *STC*, cujo valor é a soma da importância dos requisitos cobertos por *STC*. Quanto mais requisitos cobertos por *STC* que possuem valor alto para a importância, maior será a importância de *STC*.

A equação (**RS1**) é uma restrição e não permite que a soma dos tempos dos casos de teste selecionados seja maior que o tempo máximo permitido. A constante *maximumTimeAllowed* representa o tempo máximo para os testes.

A equação (**RS2**) é a segunda restrição do problema de seleção de casos de teste, e determina que, caso um caso de teste selecionado possua um caso de teste precedente (que deve ser executado antes), o caso de teste precedente também deve estar presente na suíte de testes representada por *STC*.

4.3.2 Priorização de Casos de Teste

Dado o vetor binário *STC* originado da seleção, deseja-se encontrar um vetor de inteiros *PTC*, onde *PTC* representa um subconjunto ordenado de casos de teste que contém os casos de teste selecionados na fase anterior, no sentido de atingir os seguintes objetivos OP1 e OP2 e respeitando a restrição RP1, a seguir:

$$(\boldsymbol{OP1})\ Max\ \ suiteVolatility_{PTC} = \sum\ (volatility_i * coverageWeight_i)\ , \forall\ r_i\ \in suiteCoverage_{STC})$$

$$(\boldsymbol{OP2})\ Max\ \ suiteComplexity_{PTC} = \sum\ (complexity_i * coverageWeight_i)\ , \forall\ i\ \in suiteCoverage_{STC})$$

Sujeito a:

$$(\boldsymbol{RP1})\ \forall\ t_{j1},\ t_{j2} \in C,\ \left(precedence_{t_{j2}} =\ t_{j1}\right) \rightarrow (testCasePosition_{t_{j1},PTC} < testCasePosition_{t_{j2},PTC})$$

A equação (**OP1**) deseja maximizar a volatilidade da suíte de testes *PTC*, que pode ser calculada da seguinte forma:

$$suiteVolatility_{PTC} = \sum (volatility_i * coverageWeight_i)\ , \forall\ i \in suiteCoverage_{STC})$$

Para cada requisito *i* pertencente ao conjunto $suiteCoverage_X$, ou seja, para cada requisito coberto pela suíte de testes representada por *STC*, seu valor de volatilidade é acumulado para representar o valor de volatilidade da suíte de teste, ponderado pela posição do primeiro caso de teste na suíte representada por *PTC* que cobre este requisito ($fistTestCasePosition_{PTC,i}$), como pode ser visto abaixo.

$$coverageWeight_i = Y.lenght - firstTestCasePosition_{PTC,i} + 1$$

Quando o primeiro caso de teste que cobre o requisito *i* está no início da suíte de testes *Y*, o valor para *coverageWeight*$_i$ é maior, tornando maior, nesse caso, a ponderação para sua volatilidade (equação OP1) e sua complexidade (equação OP2). Isso faz com que as suítes que cobrem os casos de teste mais voláteis mais cedo tenham maior valor para sua volatilidade. A mesma afirmação vale para a complexidade, constante na equação OP2.

A equação (**OP2**) expressa a complexidade da suíte de testes *Y*, que pode ser calculada da seguinte forma:

$$suiteComplexity_{PTC} = \sum (complexity_i * coverageWeight_i), \forall\, i \in suiteCoverage_{STC})$$

A restrição (**RP1**) reforça a precedência dos casos de teste: caso um caso de teste t_{j1} seja precedente do caso de teste t_{j2}, t_{j1} deve ser executado antes de t_{j2}, ou seja, deve estar antes de t_{j2} na suíte de testes.

4.3.3 Alocação de Casos de Teste

Dado o vetor de inteiros originado da priorização, deseja-se encontrar um vetor de inteiros *ATC*, que representa os testadores alocados para cada caso de teste e a ordem de execução desses casos de teste, no sentido de se atingir os objetivos OA1, OA2 e OA3 e respeitando a restrição RA1, a seguir:

$$(\boldsymbol{OA1})\ Min\ numberOfExecutions_{ATC} = \sum_{w=1}^{P}\sum_{a=1}^{Y.lenght} numberOfExecutions_{w,PTC_a} * isAllocated_{w,PTC_a,ATC}$$

$$(\boldsymbol{OA2})\ Max\ preference_{ATC} = \sum_{w=1}^{P}\sum_{a=1}^{PTC.lenght} preferences_{PTC_a,w} * isAllocated_{w,PTC_a,ATC}$$

$$(\boldsymbol{OA3})\ Max\ experience_{ATC} = \sum_{w=1}^{P}\sum_{a=1}^{PTC.lenght}\left(experience_w - penalty_{w,PTC_a}\right) * isAllocated_{w,PTC_a,ATC}$$

Sujeito a:

$$(\boldsymbol{RA1})\ \forall\ t_{j1},\ t_{j2} \in C,$$
$$\left(\left(precedence_{t_{j2}} = t_{j1}\right)\ and\ (isAllocated_{w,t_{j2},ATC} = 1)\right) \rightarrow$$
$$(isAllocated_{w,t_{j1},ATC} = 1)$$

A equação (**OA1**) deseja atribuir para cada testador os casos de teste que ele executou menos vezes. A abordagem proposta considera casos de teste que estão sendo testados pela primeira vez, além de casos de teste antigos, já executados (teste de regressão).

A função $isAllocated_{w,PTC_a,ATC}$ informa se um caso de teste PTC_a está alocado para o testador *w* na solução gerada *ATC*. PTC_a representa o caso de teste que será executado na ordem *a*, contido em *PTC*. Esta função retorna 1 quando o testador *w* está alocado para o caso de teste PTC_a ou 0, caso contrário.

Então, para cada alocação dada pela função *isAllocated*, o número de vezes que o testador executou o caso de teste correspondente é computado na função *suiteNumberOfExecutions*.

A equação (**OA2**) deseja atribuir aos testadores os casos de teste que eles preferem executar. Para cada alocação dada pela função *isAllocated*, o valor de preferência do testador pelo caso de teste ao qual está alocado é computado na função *suitePreference*.

A equação (**OA3**) deseja alocar os casos de teste a testadores que tem no mínimo a experiência exigida pelo caso de teste. Cada caso de teste possui um atributo chamado "experiência mínima", conforme definido anteriormente, que informa qual a experiência mínima do testador que deve ser alocado a este caso de teste. Deve ser possível alocar testadores com experiência menor que a requerida pelo caso de teste, porém quando isto acontecer uma penalidade deve ser aplicada à função que representa este objetivo, para torná-la menos competitiva.

Para cada alocação dada pela função *isAllocated*, o valor de experiência do testador é computado na função suiteExperience, e diminuído da penalidade se for o caso.

A equação (**RA1**) é uma restrição do problema e garantirá que, para dois casos de teste t_{j1} e t_{j2}, se t_{j1} for precedente de t_{j2}, este deve ser executado pelo mesmo testador que executará t_{j2} (na equação, o testador w).

4.4 Considerações Finais

Este capítulo apresentou matematicamente a abordagem proposta nesta dissertação. As funções foram implementadas em Java e os resultados dos experimentos serão mostrados no Capítulo 5, a seguir.

5 AVALIAÇÃO

5.1 Visão Geral dos Experimentos

Este capítulo detalha os experimentos realizados para avaliar o desempenho de algumas metaheurísticas, comumente utilizadas na literatura, na resolução dos problemas modelados nesta dissertação. Mais especificamente, os experimentos foram planejados de modo a responder as questões abaixo :

Q1) Qual metaheurística é mais eficiente para a abordagem proposta nesta dissertação, dentre as três selecionadas para os experimentos (NSGA-II, MOCell e SPEA2)?

Q2) Qual a influência dos fatores de entrada da abordagem proposta na busca de soluções para os problemas de seleção, priorização e alocação de casos de teste?

Q3) O uso de técnicas de otimização, conforme a abordagem integrada, interativa e multi-objetiva, é competitiva em relação à respostas de gerentes de teste?

Para responder as questões acima, três experimentos foram projetados e executados:

Experimento 1) Aplicação de Técnicas de Otimização

Neste experimento, três algoritmos multi-objetivos serão aplicados aos problemas de seleção, priorização e alocação de casos de teste, e serão comparados mediante o uso de indicadores, descritos na subseção 5.3. A comparação também será feita com um algoritmo randômico. Este experimento responderá a questão Q1.

Experimento 2) Análise de Sensibilidade

Para realizar a análise de sensibilidade, criou-se instâncias que variam uma determinada entrada para a abordagem proposta. Por exemplo, para estudar a influência da variação no número de casos de teste, três instâncias que contém número de casos de teste diferentes (pequeno, médio e grande) são executadas e seus resultados comparados. Este experimento responderá a questão Q2.

Experimento 3) Competitividade Humana

Neste experimento, as respostas dos algoritmos multi-objetivos são comparadas com as respostas de possíveis usuários da abordagem proposta, ou seja, gerentes de teste. Para cada problema, elaborou-se um questionário contendo a descrição do problema e os dados da instância. Além disso, o tempo de resposta dos usuários foi registrado, por problema (seleção, priorização e alocação de casos de teste). Este experimento responderá a questão Q3.

5.2 Algoritmos

Os algoritmos utilizados nos experimentos são: NSGA-II, MOCell e SPEA2, todos baseados em algoritmos genéticos, porém multi-objetivos. Esses algoritmos foram escolhidos devido ao seu bom desempenho em relação aos demais algoritmos multi-objetivos, como pode ser visto em (NEBRO et al., 2008), (ZITZLER et al., 2000) e (ZITZLER et al.,m 2001).

Inicialmente, foi realizado um estudo das melhores taxas de recombinação e mutação a serem utilizadas para cada algoritmo. Nesse processo, foram criadas combinações de taxas, e cada uma foi testada para cada algoritmo. Foram realizadas cinco execuções por algoritmo por combinação, utilizando a instância BASIC_2, que será explicada na subseção 5.3. A TABELA 2 descreve as taxas utilizadas e a TABELA 3 descreve a combinação entre as mesmas.

TABELA 2 – Taxas de Recombinação e Mutação Utilizadas

Taxa de Recombinação			Taxa de Mutação		
R_S	R_M	R_L	M_S	M_M	M_L
0,7	0,8	0,9	0,005	0,007	0,1

Na TABELA 2, as colunas S, M e L representam, respectivamente, taxas baixa (S), média (M) e alta (L). Utilizaremos a notação R_a e M_a para denotar as taxas de recombinação e mutação, sendo a = S, M ou L.

TABELA 3 – Combinação de Taxas de Recombinação e Mutação

	M_S	M_M	M_L
R_S	R_SM_S	R_SM_M	R_SM_L
R_M	R_MM_S	R_MM_M	R_MM_L
R_L	R_LM_S	R_LM_M	R_LM_L

Após a execução dos algoritmos, o *hypervolume*, que será explicado na subseção 5.4, foi calculado para cada algoritmo para descobrir qual a mellhor combinação de taxas para cada um deles. O resultado foi o seguinte: para o NSGA-II, combinação R_LM_L, para o MOCell, combinaçãoR_MM_M, e, finalmente, para o SPEA2, combinação R_LM_L, conforme as combinações da TABELA 3.

Além das taxas de recombinação e mutação, foi necessário definir o tipo de recombinação e mutação (como serão aplicados esses operadores). Neste caso, a escolha foi feita por problema (seleção, priorização ou alocação de casos de teste).

Para o problema de seleção de casos de teste, um operador de recombinação com apenas um ponto de corte foi utilizado, o mesmo definido na Fundamentação Teórica desta dissertação. Como operador de mutação, um algoritmo que troca um *bit* é utilizado (se o *bit* for 0, o algoritmo troca por 1; se o *bit* for 1, o algoritmo troca por 0).

Para o problema de priorização de casos de teste, o algoritmo de recombinação utilizado é baseado em dois pontos de corte (dois números escolhidos aleatoriamente), ficando cada indivíduo dividido em três partes: o meio e duas

extremidades. Um terceiro número, entre 0 e 1, é gerado randomicamente. Se este terceiro número for zero, as extremidades do indivíduo são mantidos no herdeiro, e apenas a parte do meio, delimitada pelos dois pontos de corte, será trocada com o outro indivíduo pai. Se o terceiro número for 1, a parte do meio é mantida e as extremidades são trocadas. Durante a operação de troca, os elementos são inseridos na ordem em que estão listados no outro pai.

No problema de alocação de casos de teste, o mesmo algoritmo de recombinação foi implementado, com apenas uma diferença: para o problema de alocação, é permitido repetir os genes do indivíduo, que neste caso serão os testadores, enquanto no problema de priorização não é permitido repetir os genes, visto que um caso de teste não pode estar em mais de uma posição na suíte de testes.

Tanto para o problema de priorização como para o problema de alocação de casos de teste, o operador utilizado para mutação é um algoritmo que escolhe dois genes do indivíduo e os troca de posição.

Como operador de seleção foi utilizado o torneio binário, para os três problemas.

5.3 Instâncias

Para a execução dos experimentos, foram criadas 29 instâncias, de modo aleatório. A TABELA 4 detalha as instâncias criadas.

TABELA 4 – Instâncias Geradas

INSTÂNCIA	Número de Requisitos	Número de Casos de Teste	Número de Testadores	% de Precedência dos Casos de Teste	% do Tempo de Execução (Tempo Máximo)
BASIC_1	20	40	2	20	50
BASIC_2	100	140	2	20	50
SEL_TC_S	100	100	2	20	50
SEL_TC_M	100	200	2	20	50
SEL_TC_L	100	300	2	20	50
SEL_REQ_S	50	140	2	20	50
SEL_REQ_M	80	140	2	20	50
SEL_REQ_L	140	140	2	20	50

SEL_TIME_S	100	140	2	20	30
SEL_TIME_M	100	140	2	20	70
SEL_TIME_L	100	140	2	20	90
SEL_PREC_S	100	140	2	10	50
SEL_PREC_M	100	140	2	50	50
SEL_PREC_L	100	140	2	80	50
PRI_TC_S	100	100	2	20	50
PRI_TC_M	100	200	2	20	50
PRI_TC_L	100	300	2	20	50
PRI_PREC_S	100	140	2	10	50
PRI_PREC_M	100	140	2	50	50
PRI_PREC_L	100	140	2	80	50
ALO_TC_S	100	100	2	20	50
ALO_TC_M	100	200	2	20	50
ALO_TC_L	100	300	2	20	50
ALO_TES_S	100	140	3	20	50
ALO_TES_M	100	140	8	20	50
ALO_TES_L	100	140	15	20	50
ALO_PREC_S	100	140	2	20	30
ALO_PREC_M	100	140	2	20	70
ALO_PREC_L	100	140	2	20	90

As instâncias com o mesmo prefixo são utilizadas para uma mesma análise de sensibilidade (experimento 2). Por exemplo, as instâncias SEL_TC_S, SEL_TC_M e SEL_TC_L são executadas e comparadas para avaliar a influência da variação do número de casos de teste (TC) para o problema de seleção de casos de teste (SEL), onde o número de casos de teste pode ser pequeno (S), médio (M) ou grande (L). As demais notações utilizadas para análise de sensbilidade são: REQ (varia a quantidade de requisitos), TIME (varia o tempo máximo para os testes), PREC (varia o percentual de precedência dos casos de teste) e TES (varia a quantidade de testadores).

5.4 Indicadores Utilizados

Nos experimentos, utilizaremos três indicadores para a comparação: o *hypervolume* (DEB, 2008), o *spread* (DEB, 2008) e o tempo de execução.

O *hypervolume*, que avalia convergência e diversidade, é um indicador que calcula o volume coberto por um conjunto de soluções, usando um ponto de referência. Para cada solução i pertencente ao conjunto de soluções gerado, um hipercubo é construído com esta solução e um ponto de referência W (DEB, 2008).

Esse ponto de referência pode ser formado pelos piores valores das funções objetivo, e pode ser visualizado na FIGURA 15.

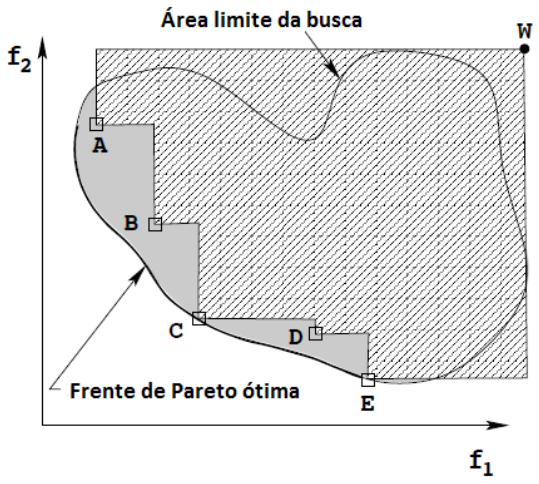

FIGURA 15 – Hypervolume – adaptado de (DEB, 2008)

A seguinte fórmula calcula o *hypervolume* para um conjunto de soluções:

$$HV = volume(\bigcup_{i=1}^{|Q|} v_i)$$

Como pode ser visto na fórmula e visualizando na FIGURA 15, o hypervolume representa o volume da união de hipercubos de todas as soluções pertencentes a *Q*, que representa um conjunto de soluções, ou seja, são |*Q*| hipercubos. O volume de um hipercubo é representado por v_i na fórmula acima.

Chamaremos de *hypervolume* neste trabalho o *hypervolume* normalizado, que é a razão entre o *hypervolume* do conjunto de soluções gerado pelos algoritmos sobre o *hypervolume* da Frente de Pareto ótima, que nesta abordagem é formada pelas melhores soluções dos três algoritmos multi-objetivos. São desejados valores próximos de 1 para este indicador. A equação para o *hypervolume* normalizado é:

$$HVR = \frac{HV(Q)}{HV(P^*)}$$

onde *Q* é um conjunto de soluções geradas por um algoritmo multi-objetivo e P^* é a Frente de Pareto ótima.

O *spread* é um indicador utilizado para determinar o quão bem distribuídas estão as soluções geradas ao longo da Frente de Pareto. O *spread* é dado pela seguinte fórmula:

$$\Delta = \frac{\sum_{m=1}^{M} d_m^e + \sum_{i=1}^{|Q|} |d_i - \bar{d}|}{\sum_{m=1}^{M} d_m^e + |Q|\bar{d}}$$

Onde Q é o conjunto de soluções gerado. O termo d_i é a distância entre duas soluções consecutivas de Q, $\bar{d}$ é o valor médio dessas distâncias e d_m^e é a distância entre as extremidades de Q e a Frente de Pareto ótima, gerada pelas melhores soluções encontradas pelos algoritmos. Valores próximos de zero indicam proximidade à Frente de Pareto real.No exemplo da FIGURA 16, seja Q um conjunto formado pelas soluções geradas por um algoritmo multi-objetivo.

A solução da extremidade da direita de Q, coincide com a solução de extremidade direita da Frente de Pareto (solução 8), então a distância entre as extremidades da direita (d_2^e) é igual a 0. Porém, a solução da extremidade da esquerda de Q não coincide com a solução de extremidade esquerda da Frente de Pareto ótima (solução 1), logo a distância entre as mesmas (d_1^e) deve ser calculada. Além disso, a distância entre cada uma das soluções pertencentes a Q será calculada, de acordo com a fórmula do *spread*.

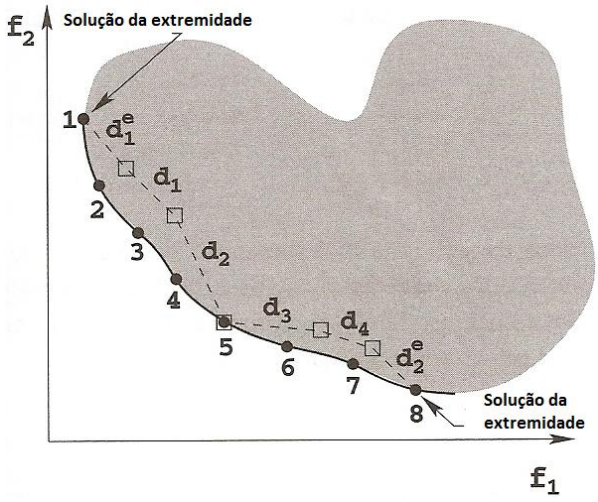

FIGURA 16 – spread – adaptado de (DEB, 2008)

5.5 Experimento 1: Aplicação das Técnicas de Otimização

As metaheurísticas NSGA-II, MOCell e SPEA2, além de um algoritmo randômico, foram aplicados a todas as instâncias descritas anteriormente. Os resultados desses experimentos são apresentados nas tabelas a seguir. Os resultados apresentados são por algoritmo, por problema e por instância, ou seja, são descritas 348 execuções no total (3 problemas x 4 algoritmos x 29 instâncias).

As TABELAS 5 a 7 mostram o valor do *hypervolume* para cada instância e algoritmo, para os problemas de seleção, priorização e alocação de casos de teste, respectivamente. Os melhores valores foram destacados em negrito.

De uma maneira geral, o SPEA2 obteve melhor *hypervolume*. Para o problema de seleção de casos de teste, o SPEA2 obteve os melhores valores para esta métrica em 68,97% das instâncias, seguido pelo NSGA-II, com 27,59%. O bom desempenho do SPEA2 se repete para os problemas de priorização de casos de teste, onde obteve melhor *hypervolume* para 72,41% das instâncias, e alocação de casos de teste, obtendo melhor *hypervolume* para 44,82% das instâncias.

Como esperado, os piores valores foram obtidos pelo algoritmo randômico, para os três problemas (seleção, priorização e alocação).

Como pode ser visto na TABELA 5, em diversas instâncias o algoritmo MOCell obteve valor 0 (zero) para o *hypervolume*. Para todas as vezes que isso ocorreu, notou-se que este algoritmo gerou apenas uma solução para a instância, e que esta solução não faz parte da Frente de Pareto da instância e que, ainda, essa solução é dominada por todas as outras pertencentes à Frente de Pareto ótima, criada a partir da composição dos resultados de todos os algoritmos. O valor zero para *hypervolume* ocorreu por conta da maneira de se calcular o indicador para um conjunto de soluções, neste caso, geradas pelo MOCell. Conforme definido anteriormente, um ponto de referência (uma solução gerada a partir dos piores valores para cada um dos objetivos considerando todas as soluções) é calculado, e então o volume coberto pelo conjunto de soluções é calculado, tomando como base este ponto de referência. Como, nesse

caso, existe apenas uma solução e ela é dominada por todas as outras pertencentes à Frente de Pareto ótima, ela mesma é escolhida como ponto de referência para cálculo do *hypervolume*. Sendo assim, como o conjunto de soluções do algoritmo é composto apenas por esta solução, o volume desta solução em relação ao ponto de referência, que também é esta solução, possui valor zero. Ao calcular o *hypervolume* normalizado, que é o utilizado nestes experimentos, o valor final será zero, visto que o numerador da equação que representa este valor é zero.

TABELA 5 – *Hypervolume* para o Problema de Seleção de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	**0,9754**	0,6777	0,8599	0,2889
BASIC_2	**0,9592**	0,0000	0,9560	0,3923
SEL_TC_S	0,0000	0,0000	0,0000	**1,0000**
SEL_TC_M	0,9687	0,6808	**0,9974**	0,3510
SEL_TC_L	0,9171	0,7304	**0,9900**	0,4384
SEL_REQ_S	0,9901	0,7800	**0,9993**	0,3331
SEL_REQ_M	0,9815	0,6296	**0,9985**	0,3255
SEL_REQ_L	0,0000	0,0000	**1,0000**	0,3369
SEL_TIME_S	**0,9771**	0,4995	0,9637	0,4033
SEL_TIME_M	0,9759	0,0000	**0,9992**	0,2015
SEL_TIME_L	0,9547	0,0000	**0,9996**	0,1440
SEL_PREC_S	0,9825	0,0000	**0,9996**	0,3578
SEL_PREC_M	0,9041	0,5806	**0,9987**	0,1368
SEL_PREC_L	0,9436	0,0000	**0,9853**	0,0713
PRI_TC_S	**1,0000**	0,0000	0,9506	0,3668
PRI_TC_M	0,9467	0,5670	**0,9978**	0,2762
PRI_TC_L	0,9579	0,6603	**0,9990**	0,2435
PRI_PREC_S	0,9852	0,6239	**0,9974**	0,6869
PRI_PREC_M	**0,9978**	0,6750	0,9929	0,1427
PRI_PREC_L	0,8893	0,0000	**0,9981**	0,0467
ALO_TC_S	**1,0000**	0,0000	0,0000	0,5578
ALO_TC_M	0,9366	0,5743	**0,9964**	0,2671
ALO_TC_L	**0,9869**	0,0000	0,9748	0,2561
ALO_TES_S	0,9537	0,6133	**0,9567**	0,2977
ALO_TES_M	0,9458	0,0000	**0,9517**	0,2977
ALO_TES_L	0,9217	0,6308	**0,9532**	0,3620
ALO_PREC_S	0,9582	0,0000	**0,9590**	0,3288
ALO_PREC_M	**0,9968**	0,5381	0,9812	0,3049
ALO_PREC_L	0,9278	0,6000	**0,9982**	0,3380

TABELA 6 – *Hypervolume* para o Problema de Priorização de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	0,9900	0,9837	**0,9908**	0,8504
BASIC_2	**0,9974**	0,9081	0,9857	0,7076
SEL_TC_S	0,9886	0,9490	**0,9976**	0,7478
SEL_TC_M	**0,9992**	0,9219	0,9843	0,5141

SEL_TC_L	0,9680	0,9279	**0,9997**	0,7139
SEL_REQ_S	0,9792	0,9330	**0,9995**	0,5636
SEL_REQ_M	**0,9960**	0,9011	0,9889	0,5929
SEL_REQ_L	**0,9970**	0,9484	0,9758	0,7301
SEL_TIME_S	0,9970	0,9796	**0,9998**	0,7699
SEL_TIME_M	**0,9988**	0,9526	0,9900	0,6589
SEL_TIME_L	0,9754	0,9416	**0,9903**	0,6520
SEL_PREC_S	0,9692	0,9459	**0,9991**	0,6813
SEL_PREC_M	0,9829	0,9421	**0,9923**	0,6970
SEL_PREC_L	0,9852	0,9438	**0,9859**	0,7315
PRI_TC_S	0,9494	0,9347	**0,9977**	0,7278
PRI_TC_M	0,9473	0,9436	**0,9998**	0,6053
PRI_TC_L	**0,9932**	0,9525	0,9680	0,5401
PRI_PREC_S	**0,9923**	0,9297	0,9732	0,6516
PRI_PREC_M	0,9742	0,9797	**0,9997**	0,7235
PRI_PREC_L	0,9809	0,9413	**0,9912**	0,7507
ALO_TC_S	**0,9939**	0,9651	0,9887	0,7195
ALO_TC_M	0,9433	0,9177	**0,9994**	0,6385
ALO_TC_L	0,9676	0,8917	**0,9999**	0,5338
ALO_TES_S	0,9670	0,9395	**0,9707**	0,6793
ALO_TES_M	0,9564	0,9261	**0,9999**	0,6310
ALO_TES_L	0,8981	0,8211	**0,9022**	0,6694
ALO_PREC_S	0,9960	0,9046	**0,9984**	0,6670
ALO_PREC_M	0,9914	0,9066	**0,9960**	0,6566
ALO_PREC_L	0,9904	0,9429	**0,9978**	0,7269

TABELA 7 – *Hypervolume* para o Problema de Alocação de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	0,000787	**0,000796**	0,000813	0,000433
BASIC_2	**0,000049**	0,000045	0,000045	0,000015
SEL_TC_S	0,000395	0,000408	**0,000460**	0,000140
SEL_TC_M	0,000126	0,000161	0,000202	**0,013550**
SEL_TC_L	0,000110	0,000099	**0,000142**	0,000041
SEL_REQ_S	**0,000385**	0,000272	0,000257	0,000058
SEL_REQ_M	0,000188	**0,000255**	0,000125	0,000019
SEL_REQ_L	0,000031	0,000024	**0,000063**	0,000001
SEL_TIME_S	0,000328	0,000337	**0,000358**	0,000020
SEL_TIME_M	**0,000112**	0,000065	0,000093	0,000000
SEL_TIME_L	0,000096	0,000089	**0,000113**	0,000007
SEL_PREC_S	**0,000293**	0,000200	0,000286	0,000067
SEL_PREC_M	**0,000152**	0,000121	0,000141	0,000069
SEL_PREC_L	0,000064	0,000055	**0,000085**	0,000050
PRI_TC_S	**0,000653**	0,000477	0,000506	0,000155
PRI_TC_M	0,000126	0,000129	**0,000157**	0,000012
PRI_TC_L	0,000174	0,000172	**0,000178**	0,000056
PRI_PREC_S	0,000780	0,000565	0,000692	**0,003700**
PRI_PREC_M	0,000085	0,000102	**0,000123**	0,000044
PRI_PREC_L	**0,000179**	0,000122	0,000116	0,000116
ALO_TC_S	0,000274	0,000248	**0,000306**	0,000000
ALO_TC_M	0,000136	0,000140	**0,000147**	0,000048
ALO_TC_L	0,000120	**0,000145**	0,000090	0,000034
ALO_TES_S	0,000168	0,000155	0,000182	**0,000207**
ALO_TES_M	0,000450	0,000375	0,000521	**0,001500**

ALO_TES_L	0,047770	0,037500	0,046300	**0,060800**
ALO_PREC_S	0,000285	0,000283	0,000300	**0,000911**
ALO_PREC_M	0,000198	0,000166	**0,000303**	0,000006
ALO_PREC_L	0,000054	0,000138	**0,000269**	0,000003

As TABELAS 8, 9 e 10 apresentam os valores de tempo de execução dos algoritmos. O algoritmo randômico obteve os melhores tempos de execução, porém gerou soluções bem piores que os demais algoritmos. Não considerando o algoritmo randômico, o NSGA-II obteve menores valores para o tempo de execução para os problemas de seleção, em 93,10% das instâncias, e alocação de casos de teste, em 86,21%. Já para os problemas de priorização de casos de teste, os dois melhores foram o SPEA2, em 51,72%, e o MOCell, em 41,38%.

Pode-se notar, porém, que o algoritmo SPEA2 é mais lento que os demais algoritmos. O SPEA2 chegou a ser 36,14 vezes mais lento que o NSGA-II, na instância BASIC_1, para o problema de seleção de casos de teste. Mas, no contexto geral, o SPEA2 foi 2,1 vezes mais lento que o NSGA-II e 1,32 vezes mais lento que o MOCell. Este tempo é medido em milisegundos.

TABELA 8 – Tempo de Execução dos Algoritmos para o Problema da Seleção de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	**578**	2142	20891	31
BASIC_2	**2406**	4535	5094	78
SEL_TC_S	**1672**	4398	7500	47
SEL_TC_M	**3828**	6604	9609	125
SEL_TC_L	**8282**	9901	13719	141
SEL_REQ_S	**2390**	3822	7922	62
SEL_REQ_M	**2734**	3997	8109	63
SEL_REQ_L	**3281**	5180	6672	63
SEL_TIME_S	**3469**	5027	7578	2046
SEL_TIME_M	**2438**	4506	7109	47
SEL_TIME_L	**2953**	4507	7094	78
SEL_PREC_S	**3328**	5422	6515	78
SEL_PREC_M	**2859**	5931	6188	78
SEL_PREC_L	**3375**	5506	8328	94
PRI_TC_S	**1500**	3726	7595	47
PRI_TC_M	**4125**	6525	8938	93
PRI_TC_L	8984	**8756**	13938	125
PRI_PREC_S	**2438**	4434	5156	78
PRI_PREC_M	**2625**	4665	6969	78
PRI_PREC_L	**2703**	5209	10328	78
ALO_TC_S	**1562**	4392	12438	62

ALO_TC_M	**5171**	6029	9187	94
ALO_TC_L	8531	**7841**	14438	125
ALO_TES_S	**2938**	5644	9344	63
ALO_TES_M	**3563**	5897	8312	94
ALO_TES_L	**2781**	4896	7328	79
ALO_PREC_S	**2843**	4601	6594	94
ALO_PREC_M	**2672**	5045	8453	109
ALO_PREC_L	**2422**	4255	6718	62

TABELA 9 – Tempo de Execução dos Algoritmos para o Problema da Priorização de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	**3394**	7217	10516	140
BASIC_2	57385	60182	**56593**	719
SEL_TC_S	43391	47206	**42640**	532
SEL_TC_M	98141	**88010**	**88010**	953
SEL_TC_L	177328	**150837**	157801	2266
SEL_REQ_S	34297	33786	**32407**	547
SEL_REQ_M	**45672**	47870	48625	562
SEL_REQ_L	94797	89856	**83892**	953
SEL_TIME_S	**39406**	42203	41125	484
SEL_TIME_M	84453	77183	**71125**	906
SEL_TIME_L	108031	96532	**89469**	1125
SEL_PREC_S	63907	66155	**63437**	656
SEL_PREC_M	77343	71390	**70953**	656
SEL_PREC_L	84484	73527	**68078**	718
PRI_TC_S	36563	40420	**35485**	437
PRI_TC_M	109592	**94321**	96141	1000
PRI_TC_L	187687	**138310**	156704	2031
PRI_PREC_S	58312	61839	**56172**	656
PRI_PREC_M	76891	**69687**	73812	703
PRI_PREC_L	99091	77404	**69144**	766
ALO_TC_S	41563	**36370**	37909	485
ALO_TC_M	118609	**92025**	107766	1000
ALO_TC_L	180188	**148543**	174328	1562
ALO_TES_S	89453	78792	**65234**	640
ALO_TES_M	77609	**67012**	69688	719
ALO_TES_L	71438	**67754**	74610	922
ALO_PREC_S	63281	65021	**59625**	625
ALO_PREC_M	79375	**63377**	74605	672
ALO_PREC_L	89547	**71322**	83687	781

TABELA 10 – Tempo de Execução dos Algoritmos para o Problema da Alocação de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	**16048**	21056	27766	141
BASIC_2	**1494160**	1944050	2254896	4828
SEL_TC_S	**174102**	205678	224283	1328
SEL_TC_M	**483142**	627135	583538	5250
SEL_TC_L	**1660313**	2233863	2252546	15735
SEL_REQ_S	**362875**	437923	526906	2297
SEL_REQ_M	**276063**	317276	356098	2656

SEL_REQ_L	**513658**	910719	821391	2688
SEL_TIME_S	**83938**	86797	84767	1984
SEL_TIME_M	**550560**	798633	712565	3781
SEL_TIME_L	**835907**	1252056	1267570	5234
SEL_PREC_S	**388312**	432166	527416	2532
SEL_PREC_M	**395141**	616682	622672	2047
SEL_PREC_L	**647593**	1104769	1269484	2047
PRI_TC_S	142571	**138103**	210608	1109
PRI_TC_M	**749610**	890964	873641	4735
PRI_TC_L	**1220654**	1366504	1366813	12657
PRI_PREC_S	258791	**251301**	273330	2250
PRI_PREC_M	**584000**	778376	792697	1969
PRI_PREC_L	**501416**	719044	790176	1406
ALO_TC_S	71313	**68596**	72049	906
ALO_TC_M	**748563**	974425	1153547	4546
ALO_TC_L	**2271602**	4032921	4333860	14672
ALO_TES_S	**348685**	452072	563966	2922
ALO_TES_M	389605	**387368**	419750	2531
ALO_TES_L	**320622**	366912	374454	3000
ALO_PREC_S	**282848**	357539	365451	2671
ALO_PREC_M	**393657**	468044	460319	2640
ALO_PREC_L	**312219**	332267	334859	2860

As TABELAS 11, 12 e 13 apresentam os valores para o indicador *spread*. Os melhores valores de spread foram obtidos pelo algoritmo randômico. É importante notar que, sempre que o hypervolume tem valor 0 (algoritmo gerou apenas uma solução), o *spread* tem valor 1. Isso se deve ao fato de que, como dito anteriormente, o *spread* mede o grau de distribuição das soluções, ou seja, a diversidade das soluções. Quando existe apenas uma solução gerada, não há como medir sua distribuição, por isso um valor 1 é associado ao indicador *spread*. Como só existe uma solução (um ponto) no conjunto gerado, na equação do *spread* os valores para d_i e $\bar{d}$ são 0 (zero). Sendo assim, a equação fica na forma $\sum_{m=1}^{M} d_m^e / \sum_{m=1}^{M} d_m^e$, gerando como resultado o valor 1. Valores maiores que 1 são atribuídos para este indicador quando há uma distribuição não uniforme das soluções.

TABELA 11 – *Spread* para o Problema de Seleção de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	1,9111	**0,2285**	1,8063	0,9684
BASIC_2	1,5913	1,0000	1,7417	**0,8836**
SEL_TC_S	1,0000	1,0000	1,0000	**0,9100**
SEL_TC_M	1,5043	**0,1350**	1,8000	0,9577
SEL_TC_L	1,3283	**0,1029**	1,7552	1,0086
SEL_REQ_S	1,6271	0,9703	1,9393	**0,9086**
SEL_REQ_M	1,7415	0,9532	1,6625	**0,9308**

SEL_REQ_L	1,0000	**0,0000**	1,6512	0,9134
SEL_TIME_S	1,8326	**0,2600**	1,6993	0,9100
SEL_TIME_M	1,5725	1,0000	1,8887	**0,9523**
SEL_TIME_L	1,7625	1,0000	1,9416	**0,9854**
SEL_PREC_S	1,6964	1,0000	1,8160	**0,9139**
SEL_PREC_M	1,6831	**0,2267**	1,8553	0,9778
SEL_PREC_L	**0,0312**	1,0000	1,8621	0,9780
PRI_TC_S	1,9797	1,0000	1,9535	**0,9347**
PRI_TC_M	1,3312	**0,2136**	1,6491	0,9375
PRI_TC_L	1,5814	0,9682	1,5849	**0,9271**
PRI_PREC_S	1,7425	**1,2001**	1,7466	1,4159
PRI_PREC_M	1,4948	**0,1506**	1,7487	0,9855
PRI_PREC_L	1,7059	**1,0000**	1,8990	1,0046
ALO_TC_S	1,9797	1,0000	1,0000	**0,9339**
ALO_TC_M	1,6245	0,9483	1,7346	**0,9008**
ALO_TC_L	1,6338	1,0000	1,6917	**0,9005**
ALO_TES_S	1,6446	0,9563	1,8155	**0,8820**
ALO_TES_M	1,6221	1,0000	1,8788	**0,8869**
ALO_TES_L	0,9123	**0,1454**	1,8556	0,9395
ALO_PREC_S	1,5527	1,0000	1,6342	**0,9868**
ALO_PREC_M	1,5655	**0,2410**	1,4270	0,9577
ALO_PREC_L	1,7623	**0,2045**	1,6290	0,8602

TABELA 12 – Spread para o Problema de Priorização de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	1,7281	1,6563	1,7306	**0,5447**
BASIC_2	1,7739	1,5692	1,8308	**0,7439**
SEL_TC_S	1,7658	1,5619	1,7720	**0,7558**
SEL_TC_M	1,8469	1,6800	1,8415	**0,7483**
SEL_TC_L	1,7016	1,5889	1,8963	**0,7971**
SEL_REQ_S	1,8120	1,5213	1,9037	**0,7587**
SEL_REQ_M	1,8381	1,5951	1,8454	**0,7261**
SEL_REQ_L	1,7784	1,6834	1,8603	**0,6667**
SEL_TIME_S	1,7069	1,5827	1,7405	**0,6986**
SEL_TIME_M	1,8202	1,6921	1,8796	**0,7786**
SEL_TIME_L	1,8266	1,6527	1,9138	**0,8245**
SEL_PREC_S	1,8095	1,6358	1,8513	**0,7748**
SEL_PREC_M	1,7955	1,6249	1,8679	**0,6964**
SEL_PREC_L	1,7879	1,6706	1,8673	**0,6203**
PRI_TC_S	1,8163	1,6161	1,8094	**0,7004**
PRI_TC_M	1,7594	1,7380	1,9097	**0,7471**
PRI_TC_L	1,8623	1,5666	1,5912	**0,7371**
PRI_PREC_S	1,7868	1,5658	1,9129	**0,7001**
PRI_PREC_M	1,8353	1,6762	1,9025	**0,7179**
PRI_PREC_L	1,7963	1,6707	1,8141	**0,7518**
ALO_TC_S	1,6511	1,4797	1,7370	**0,7541**
ALO_TC_M	1,8010	1,6698	1,8930	**0,7488**
ALO_TC_L	1,6613	1,5948	1,9111	**0,7588**
ALO_TES_S	1,7605	1,6047	1,8300	**0,7599**
ALO_TES_M	1,7773	1,6361	1,8426	**0,8118**
ALO_TES_L	1,7853	1,6160	1,8845	**0,7662**

ALO_PREC_S	1,7382	1,6375	1,8340	**0,6847**
ALO_PREC_M	1,8332	1,6696	1,8657	**0,7422**
ALO_PREC_L	1,8517	1,6863	1,8855	**0,7139**

TABELA 13 – Spread para o Problema de Alocação de Casos de Teste

INSTÂNCIA	NSGA-II	MOCell	SPEA2	Randômico
BASIC_1	1,0810	**0,6802**	1,1095	0,9779
BASIC_2	1,9388	**0,4725**	0,4725	1,9385
SEL_TC_S	1,0784	0,9132	1,2126	**0,8025**
SEL_TC_M	1,2197	1,1277	1,3033	**0,8747**
SEL_TC_L	1,1814	1,0198	0,9671	**0,8615**
SEL_REQ_S	1,1798	1,5544	1,2966	**0,8138**
SEL_REQ_M	1,5002	0,9847	1,1183	**0,8341**
SEL_REQ_L	1,7012	1,5579	1,8546	**0,8198**
SEL_TIME_S	1,5178	1,1323	0,8856	**0,7987**
SEL_TIME_M	1,7626	1,2332	1,4318	**0,9054**
SEL_TIME_L	1,2243	1,3474	1,4289	**0,8933**
SEL_PREC_S	1,2737	1,0416	1,4751	**0,8407**
SEL_PREC_M	1,6419	1,1966	1,5088	**0,8555**
SEL_PREC_L	1,6106	1,4727	1,7320	**0,8819**
PRI_TC_S	0,9140	**0,8042**	1,1675	0,8399
PRI_TC_M	1,0694	0,9661	1,3698	**0,9127**
PRI_TC_L	0,9342	1,3632	0,9795	**0,8764**
PRI_PREC_S	0,9739	0,9263	0,8218	**0,7857**
PRI_PREC_M	1,7182	1,3765	1,7708	**0,7660**
PRI_PREC_L	**1,5020**	1,5149	1,6923	1,6923
ALO_TC_S	1,1384	1,4240	**0,8097**	0,8435
ALO_TC_M	1,6271	1,2481	1,6376	**0,8052**
ALO_TC_L	1,6262	1,1950	1,5364	**0,8997**
ALO_TES_S	1,1375	1,1421	1,0393	**0,7520**
ALO_TES_M	1,0962	0,9829	1,0922	**0,7583**
ALO_TES_L	0,7536	0,9137	0,9111	**0,7451**
ALO_PREC_S	1,2697	1,1736	0,9804	**0,9250**
ALO_PREC_M	1,2894	1,0697	1,1555	**0,7854**
ALO_PREC_L	1,0858	0,9852	0,9361	**0,8812**

Para ilustrar graficamente alguns desses resultados, foram gerados dois gráficos, podendo ser vistos nas FIGURAS 17 e 18. Percebe-se que as soluções do algoritmo randômico são bem piores que as soluções encontradas pelas técnicas de otimização multi-objetivo.

As FIGURAS 19 e 20 mostram os mesmos resultados que as FIGURAS 17 e 18, porém sem o algoritmo randômico. Nota-se que muitas das soluções da Frente de Pareto são obtidas do algoritmo SPEA2.

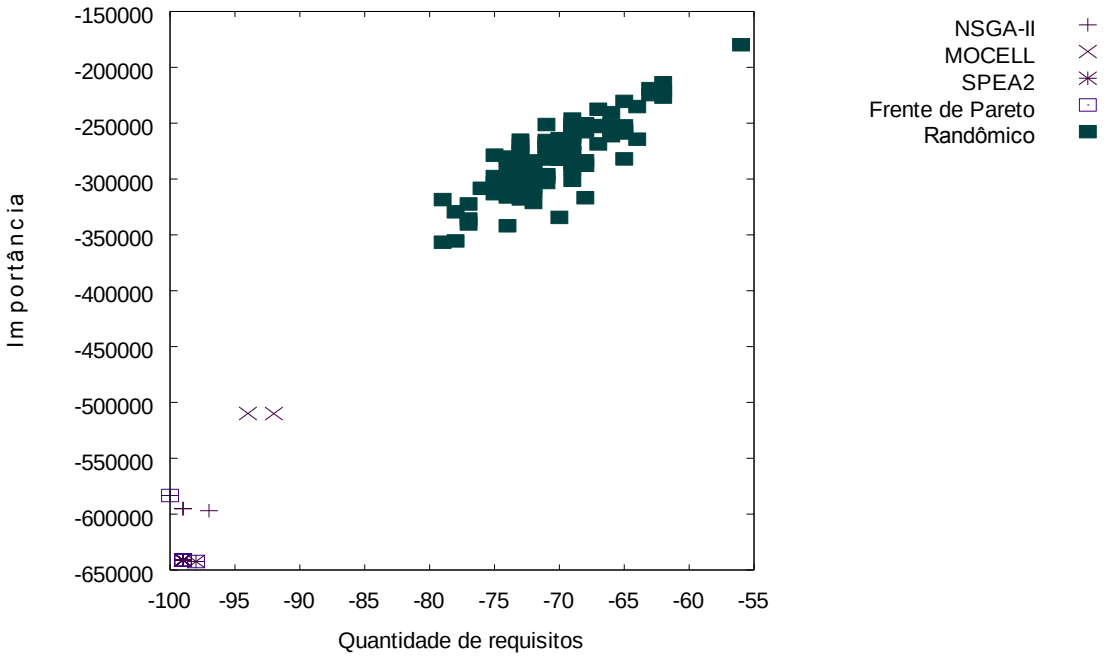

FIGURA 17 – Problema de seleção de casos de teste – Instância SEL_TC_L – Com Randômico

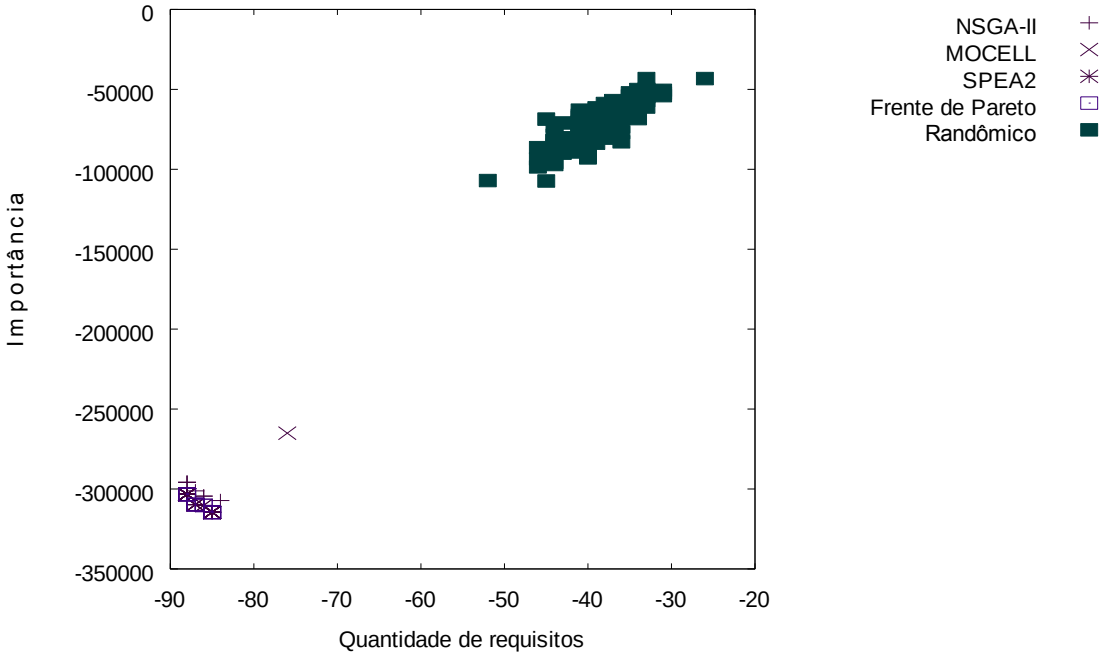

FIGURA 18 – Problema de seleção de casos de teste – Instância SEL_TIME_M – Com Randômico

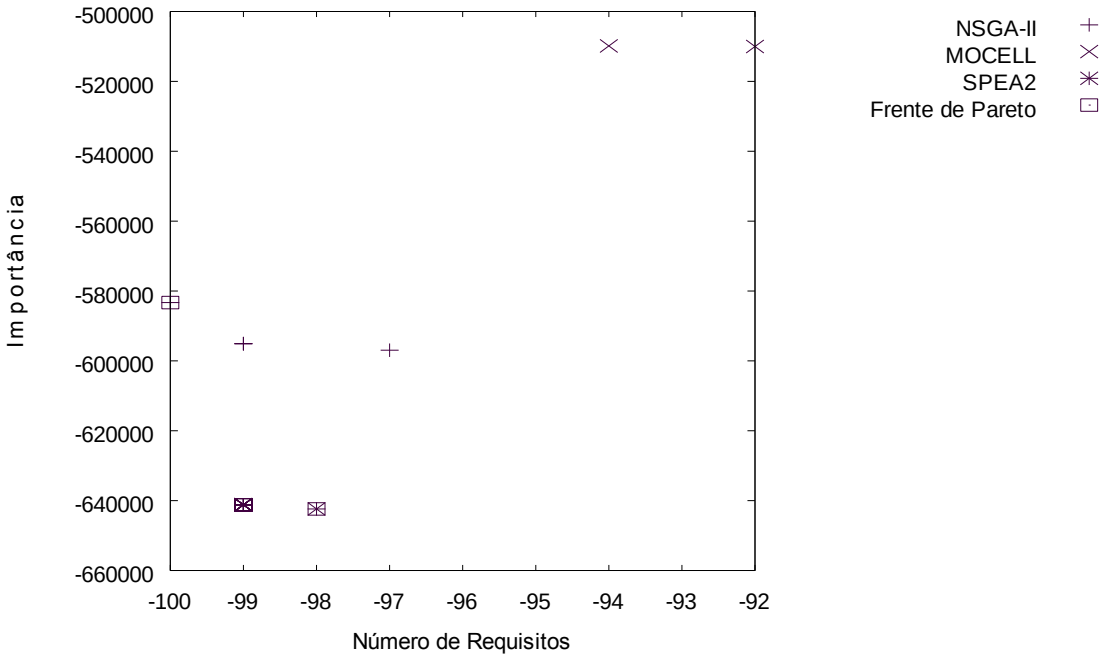

FIGURA 19 – Problema de seleção de casos de teste – Instância SEL_TIME_M – Sem Randômico

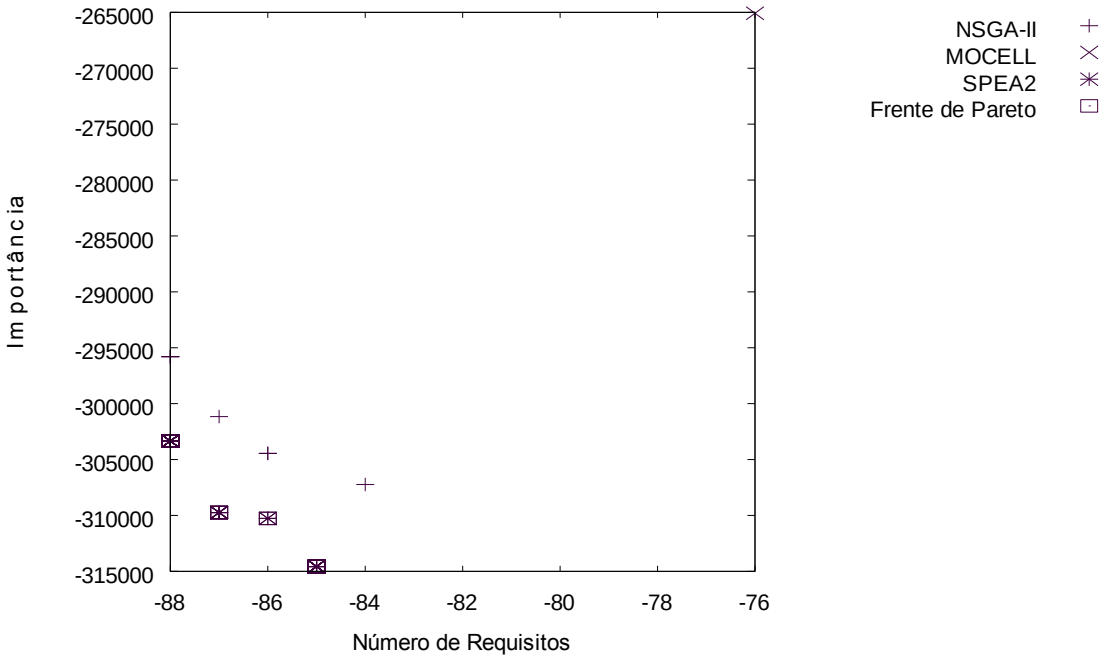

FIGURA 20 – Problema de seleção de casos de teste – Instância SEL_TIME_M – Sem Randômico

5.6 Experimento 2: Análise de Sensibilidade

A análise de sensibilidade realizada neste experimento consiste em manter fixo o valor para cada parâmetro de entrada desta abordagem, com exceção de um dos parâmetros, que deve ter seu valor variado, com o objetivo de estudar o impacto deste parâmetro para a abordagem.

Neste experimento, foram realizadas análises de sensibilidade para os três problemas: seleção, priorização e alocação de casos de teste. Para o problema da seleção de casos de teste, a sensibilidade dos seguintes parâmetros foi estudada: número de casos de teste, número de requisitos, limite de tempo para testes (um percentual sobre a soma do tempo de execução de todos os casos de teste) e o percentual de casos de teste com precedência. Para o problema de priorização de casos de teste, os parâmetros número de casos de teste e percentual de casos de teste com precedência foram estudados. Finalmente, para o problema de alocação de casos de teste, estudou-se os parâmetros número de casos de teste, número de testadores e percentual de casos de teste com precedência. A escolha dos parâmetros a serem variados foi realizada de acordo com a formulação de cada problema.

5.6.1 Variação do Número de Casos de Teste

A análise de sensibilidade referente à quantidade de casos de teste foi aplicada nos três problemas: seleção, priorização e alocação de casos de teste. O número de requisitos e os demais parâmetros se mantém, alterando-se apenas a quantidade de casos de teste. As instâncias são representadas pelos nomes SEL_TC_S, PRI_TC_S e ALO_TC_S, que representa um número de casos de teste menor, SEL_TC_M, PRI_TC_M e ALO_TC_M, que representa um número de casos de teste médio, e SEL_TC_L, PRI_TC_L e ALO_TC_L, que representa uma quantidade maior de casos de teste.

À medida que o número de casos de teste aumenta e o número de requisitos é mantido, entende-se que mais casos de teste cobrem o mesmo requisito. Sendo assim, por exemplo, pode-se cobrir um requisito importante com um caso de teste $t_{1,}$ que tem um tempo de execução baixo, e também com um caso de teste t_2, que possui um tempo de execução maior. Para a abordagem, não há diferença entre selecionar t_1 e

t_2, pois ambos cobrem o mesmo requisito. Porém, como existe a restrição de tempo de execução máximo, o algoritmo multi-objetivo tenderá a selecionar os casos de teste como tempo de execução menor para cobrir o requisito.

As FIGURAS 21 a 23 mostram o resultado da variação do número de casos de teste, sendo a FIGURA 21 representando o algoritmo SPEA2 para o problema da seleção de casos de teste, a FIGURA 22 representando o algoritmo MOCell para o problema de priorização de casos de teste, e a FIGURA 23 representanto o algoritmo NSGA-II para o problema de alocação de casos de teste. Nem todos os gráficos foram inseridos nesta dissertação, devido à grande quantidade dos mesmos, porém os gráficos são semelhante para todos os algoritmos.

Para o problema da seleção de casos de teste, observa-se que à medida que aumenta o número de casos de teste, o número de requisitos cobertos pela solução é aumentado, e consequentemente o valor de importância associado ao conjunto de casos de teste selecionados, visto que reflete a importância dos requisitos selecionados.

Como a quantidade de requisitos selecionados é maior, aumenta-se também o valor para volatilidade e complexidade associados à suite de testes, funções utilizadas no problema de priorização de casos de teste. Além disso, e pelo mesmo motivo, observou-se um aumento no valor de preferência e experiência acumulado das alocações. Na FIGURA 23, o primeiro grupo de soluções, do lado esquerdo, representa a instância ALO_TC_S, com valores de experiência e número de execuções menor. O segundo grupo de soluções representa a instância ALO_TC_M, enquanto o terceiro grupo (à direita) representa a instância ALO_TC_L.

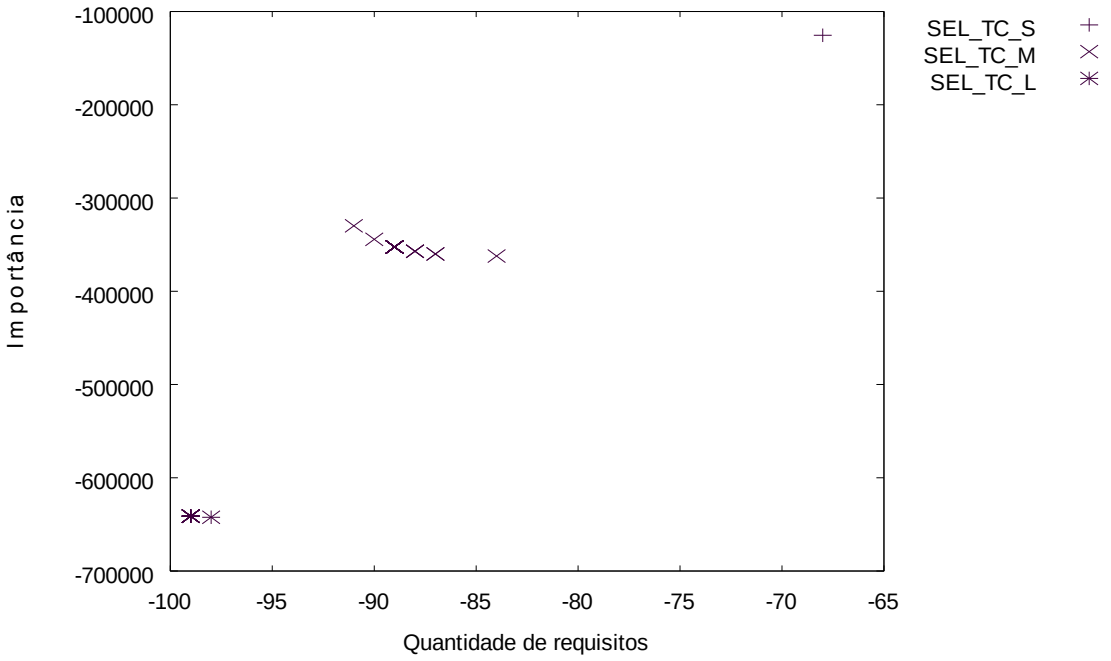

FIGURA 21 – Análise de Sensibilidade – Seleção – Número de Casos de Teste – SPEA2

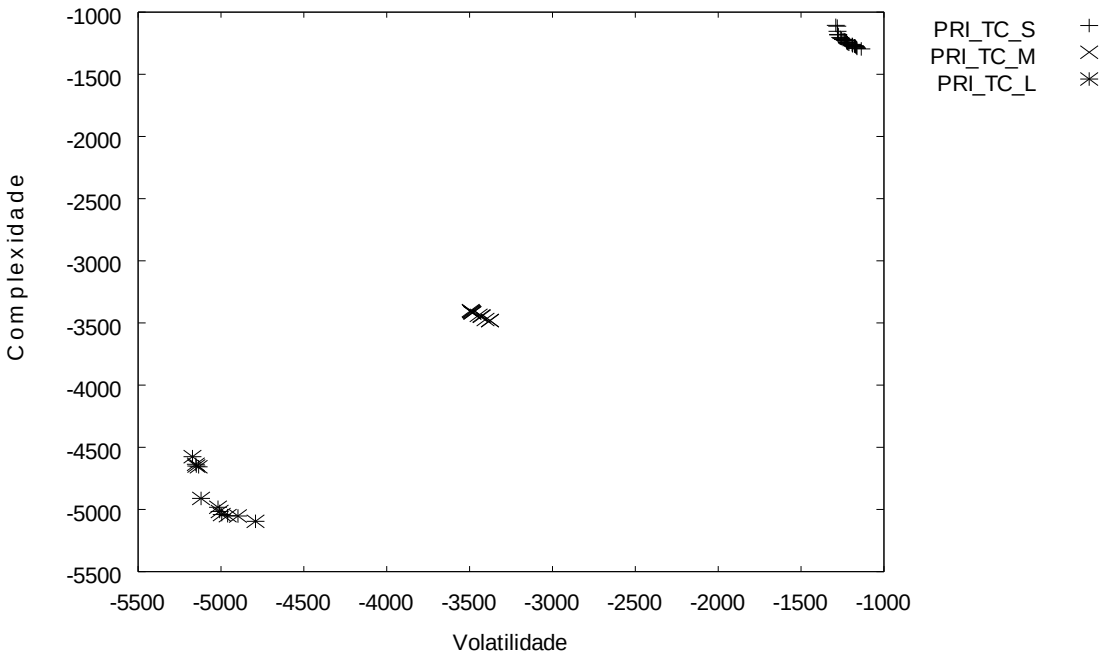

FIGURA 22 – Análise de Sensibilidade – Priorização – Número de Casos de Teste – MOCell

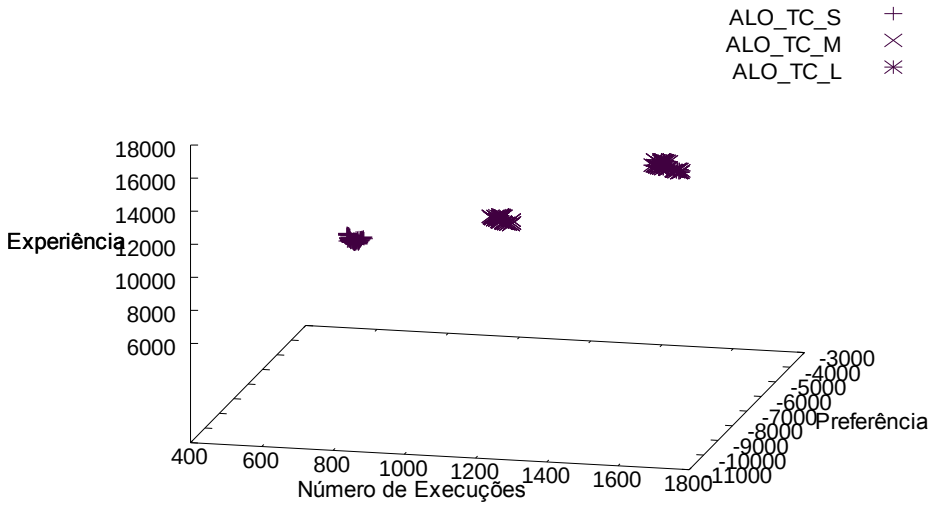

FIGURA 23 – Análise de Sensibilidade – Alocação – Número de Casos de Teste – NSGA-II

5.6.2 Variação do Número de Requisitos

A análise de sensibilidade referente à quantidade de requisitos foi aplicada apenas no problema de seleção de casos de teste. O número de casos de teste e os demais parâmetros se mantém, alterando-se apenas a quantidade de requisitos. As instâncias são representadas pelos nomes SEL_REQ_S, que representa um número de requisitos menor, SEL_REQ_M, que representa um número de requisitos médio, e SEL_REQ_L, que representa uma quantidade maior de requisitos.

À medida que o número de requisitos aumenta e o número de casos de teste é mantido, entende-se que mais casos de teste cobrem o mesmo requisito, de modo semelhante à análise anterior. Como todos os requisitos devem ser cobertos por pelo menos um caso de teste, e o número de casos de teste se mantém, então estes casos de teste deverão cobrir mais requisitos. O resultado é o mesmo obtido na variação do número de casos de teste: o algoritmo multi-objetivo tenderá a selecionar os casos de teste como tempo de execução menor para cobrir o requisito.

A FIGURA 24 mostra o resultado da variação do número de requisitos para o algoritmo NSGA-II. Observa-se que à medida que aumenta o número de requisitos,

o número de requisitos cobertos pela solução é aumentado, visto que os casos de teste cobrem mais requisitos.

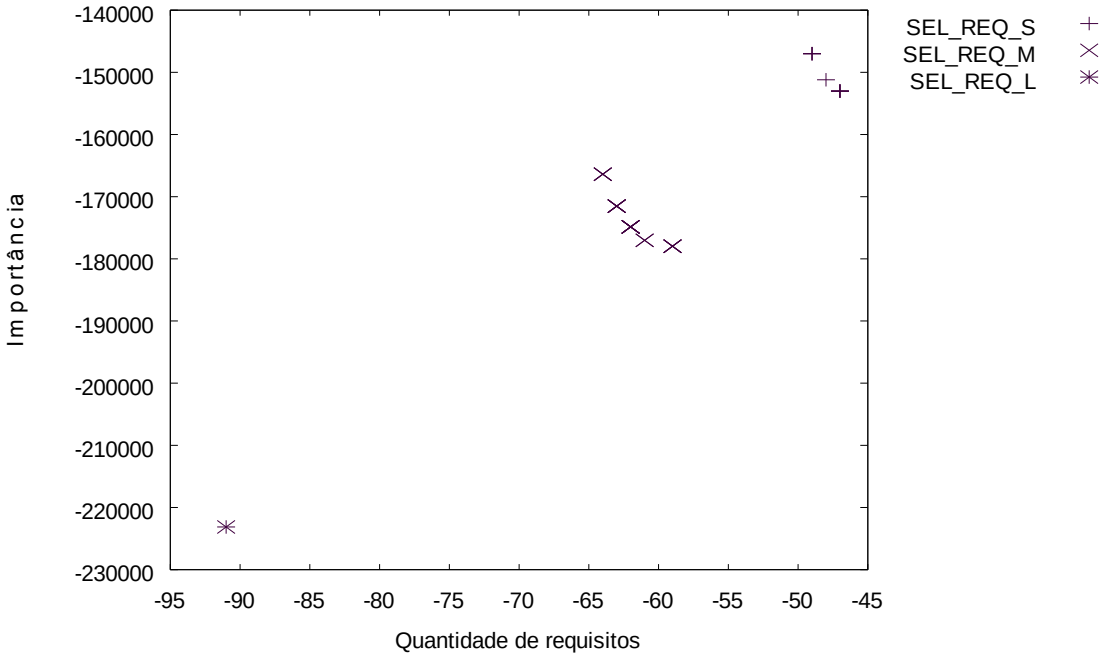

FIGURA 24 – Análise de Sensibilidade – Seleção – Número de Requisitos – NSGA-II

5.6.3 Variação do Tempo Máximo para Testes

A análise de sensibilidade reerente ao tempo disponível para testes foi aplicada apenas no problema de seleção de casos de teste. As instâncias são representadas pelos nomes SEL_TIME_S, que representa um tempo menor para a execução dos testes, SEL_TIME_M, que representa um tempo considerado médio para os testes, e SEL_TIME_L, que representa um tempo maior para a execução dos testes.

À medida que o tempo limitado para testes aumenta, entende-se que mais casos de teste podem ser selecionados. Selecionando mais casos de teste, mais requisitos poderão ser cobertos pelo subconjunto de casos de teste selecionados. O resultado é uma quantidade maior de casos de teste selecionados, e consequentemente um valor de importância maior associado às soluções geradas (conjunto de casos de teste selecionados).

A FIGURA 25 mostra o resultado da variação do tempo para testes, para o algoritmo SPEA2.

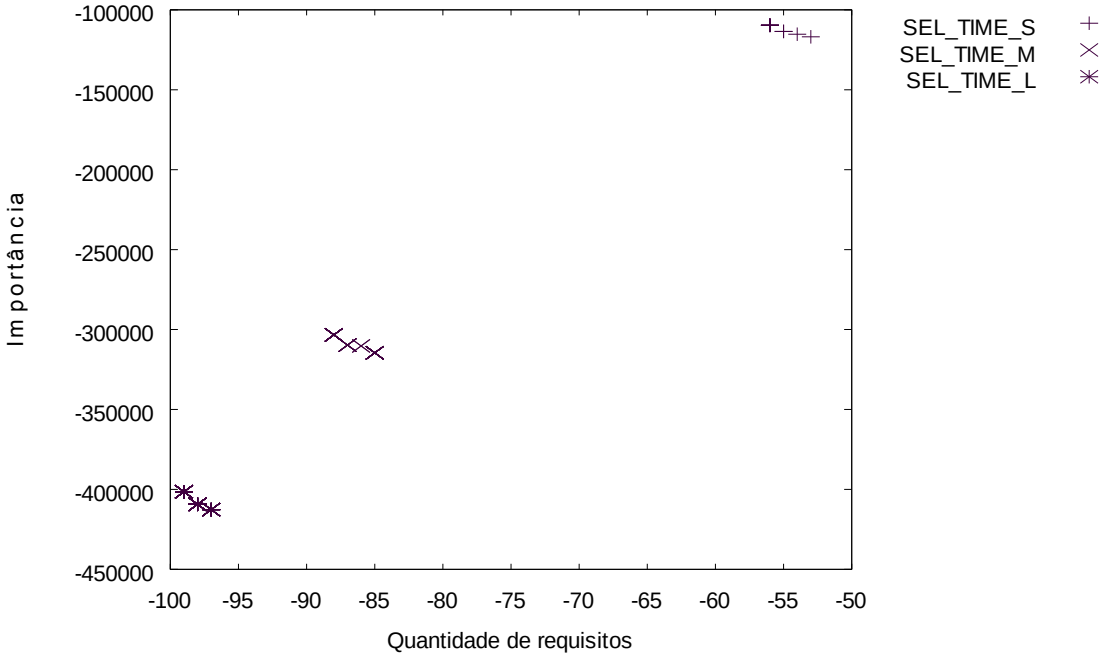

FIGURA 25 – Análise de Sensibilidade – Seleção – Tempo para Testes –SPEA2

5.6.4 Variação do Percentual de Precedência dos Casos de Teste

A análise de sensibilidade referente ao percentual de casos de teste que possuem precedente foi aplicada nos três problemas: seleção, priorização e alocação de casos de teste. As instâncias são representadas pelos nomes SEL_PREC_S, PRI_PREC_S e ALO_PREC_S, que representa um número menor de casos de teste que possuem precedente, SEL_PREC_M, PRI_PREC_M e ALO_PREC_M, que representa um número de casos de teste médio com precedentes, e SEL_PREC_L, PRI_PREC_L e ALO_PREC_L, que representa uma quantidade maior de casos de teste que possui precedentes.

À medida que o número de casos de teste com precedentes aumenta, pode ser necessário selecionar casos de teste que não cubram requisitos mais importantes para o cliente. A seleção destes casos de teste ocorre somente porque são precedentes de outros casos de teste selecionados. Sendo assim, pode ser que o valor total de importância da suíte de testes diminua, como pode ser visualizado na FIGURA 26. Um

outro cenário que pode ocorrer é selecionar um precedente que cubra o mesmo requisito, diminuindo assim a quantidade de requisitos selecionados. À medida que o número de requisitos diminui, o valor acumulado para a volatilidade e para a complexidade dos casos de teste priorizados também diminui, como pode ser visto na FIGURA 27.

Para o problema da alocação de casos de teste, observou-se na FIGURA 28 que o valor associado à experiência do testador diminiu nas alocações. De acordo com a definição do problema apresentada anteriormente, se um caso de teste t_1 tiver um precedente t_2, estes dois casos de teste devem ser executados pelo mesmo testador. Pode ser que o testador não tenha a experiência mínima requerida pelo caso de teste precedente, então, neste cenário, a função de penalidade é aplicada, fazendo com que o valor da experiência seja diminuído.

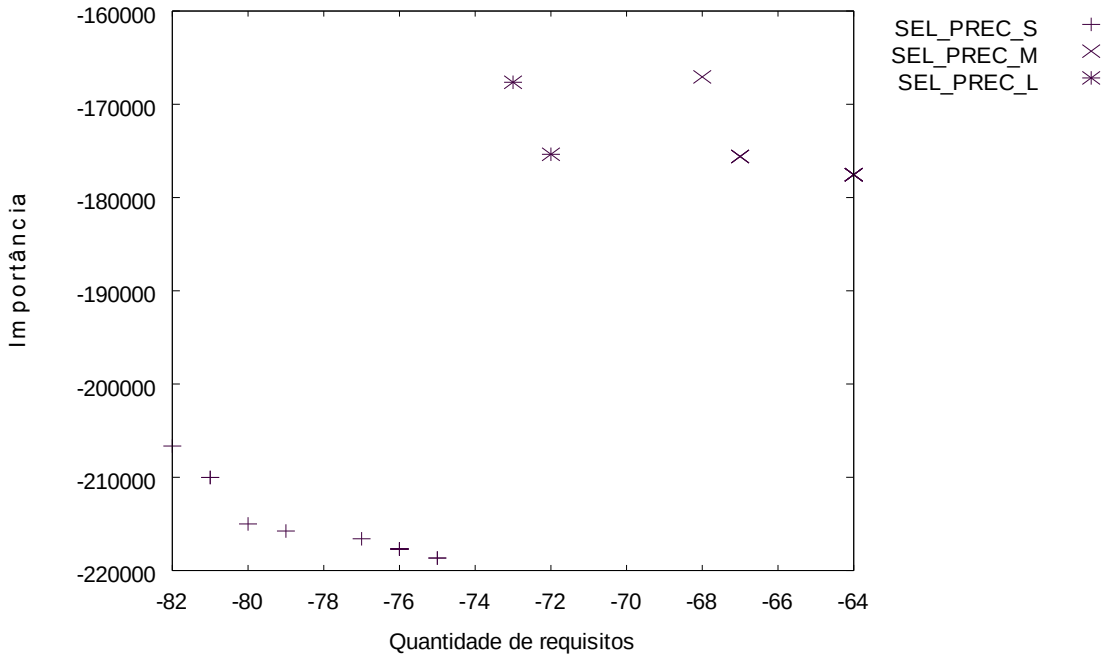

FIGURA 26 – Análise de Sensibilidade – Seleção – % de Precedência –NSGA-II

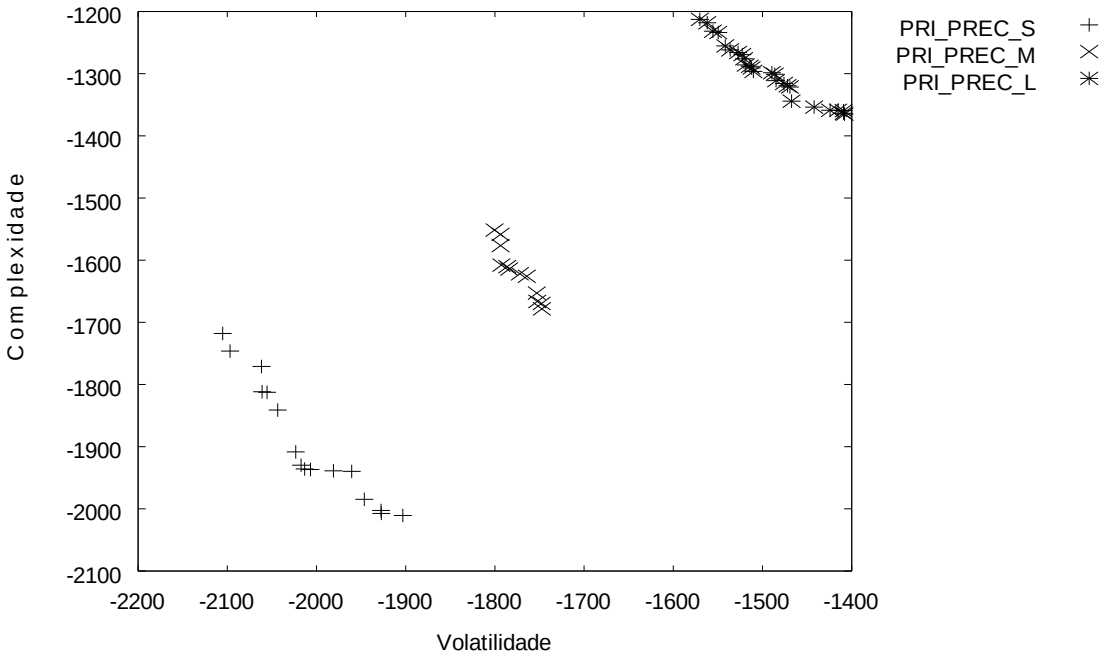

FIGURA 27 – Análise de Sensibilidade – Priorização – % de Precedência – MOCell

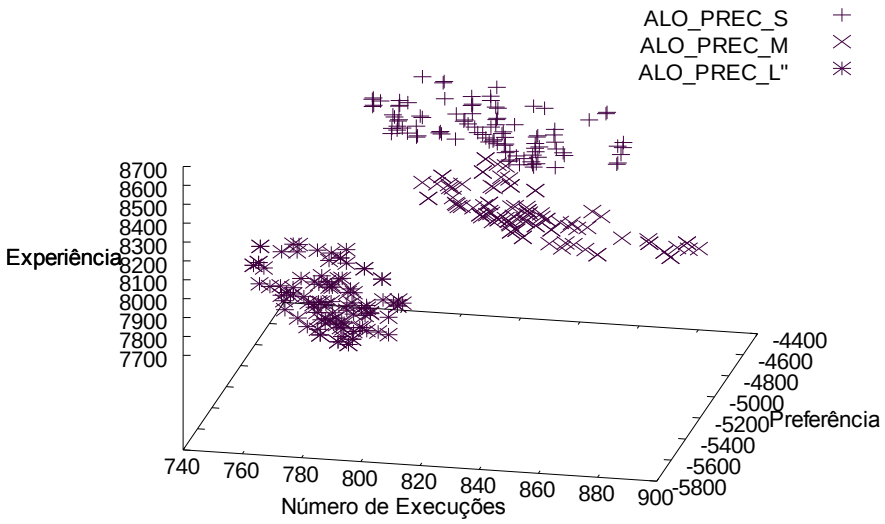

FIGURA 28 – Análise de Sensibilidade – Alocação – % de Precedência – SPEA2

Na FIGURA 28, o primeiro grupo de soluções, do lado esquerdo, representa a instância ALO_PREC_L, com valores de experiência e número de execuções menor. O segundo grupo de soluções representa a instância ALO_PREC_M, enquanto o terceiro grupo (à direita) representa a instância ALO_PREC_L.

5.6.5 Variação do Número de Testadores

A análise de sensibilidade referente ao número de testadores foi aplicada apenas ao problema de alocação de casos de teste. As instâncias são representadas pelos nomes ALO_TES_S, que representa um número menor de testadores, ALO_TES_M, que representa um número de testadores médio, e ALO_TES_L, que representa uma quantidade maior de testadores.

Na FIGURA 29, observa-se que à medida que o número de testadores aumenta, o valor acumulado para a experiência diminui nas alocações. Um dos motivos para que ocorra é a inserção de novos testadores com experiência baixa, fazendo com que os casos de teste sejam alocados a esses testadores inexperientes, e seja aplicada uma penalidade por isso. Essa penalidade faz com que o valor da experiência acumulada para o conjunto de casos de teste diminua.

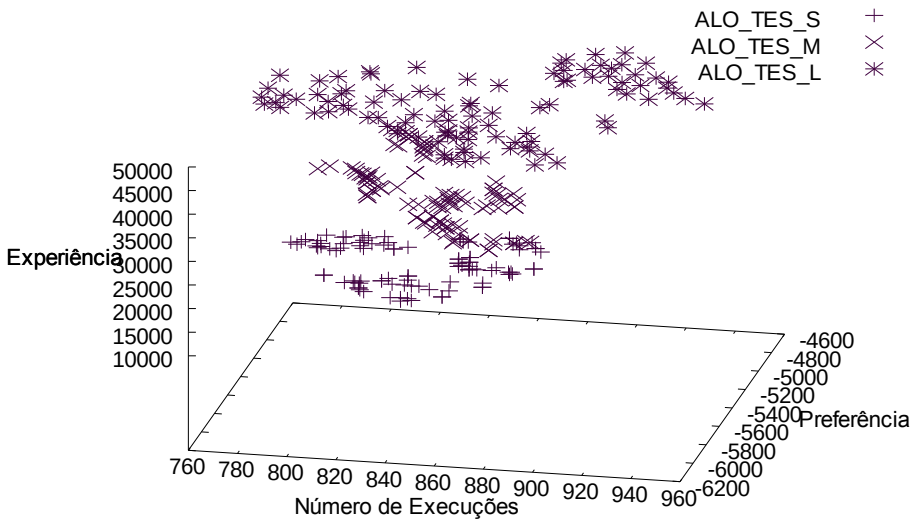

FIGURA 29 – Análise de Sensibilidade – Alocação – Número de Testadores – SPEA2

Na FIGURA 29, o grupo de soluções que está mais em baixo representa a instância ALO_TES_S, com valores de experiência menor. O segundo grupo de soluções, no meio, representa a instância ALO_TES_M, enquanto o terceiro grupo (em cima) representa a instância ALO_TES_L.

5.7 Experimento 3: Competitividade Humana

Como dito anteriormente, o objetivo deste experimento é comparar o desempenho dos algoritmos multi-objetivos com respostas fornecidas por possíveis usuários desta abordagem, ou seja, gerentes de teste.

Alguns possíveis usuários foram selecionados e elaboraram uma resposta para os problemas de seleção, priorização e alocação de casos de teste. Esses usuários são pessoas formadas em Ciência da Computação, engenheiros de software, sendo que uma parte (cerca de 60%) tem experiência em testes no mercado de trabalho e uma parte tem experiência apenas acadêmica. Para o problema de seleção de casos de teste há 10 respostas, para o problema de priorização há 7 respostas, e para o problema de alocação de casos de teste há 5 respostas.

As FIGURAS 30, 31 e 32 mostram a comparação das respostas humanas com as respostas obtidas pelos algoritmos multi-objetivos para os problemas de seleção, priorização e alocação de casos de teste, respectivamente.

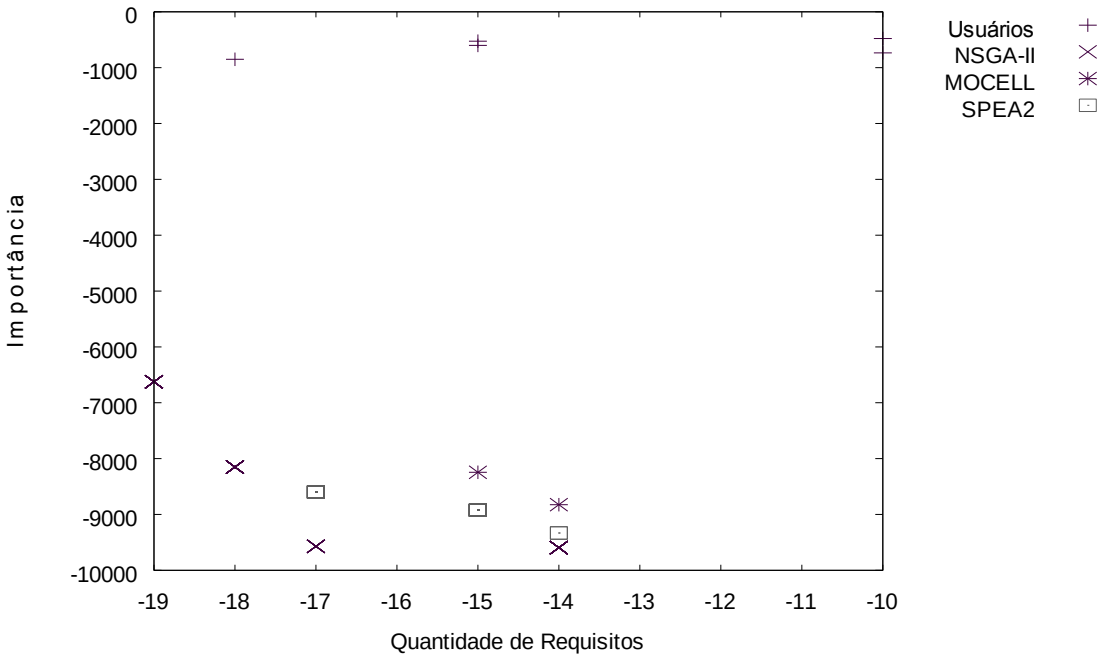

FIGURA 30 – Competitividade Humana – Seleção de Casos de Teste

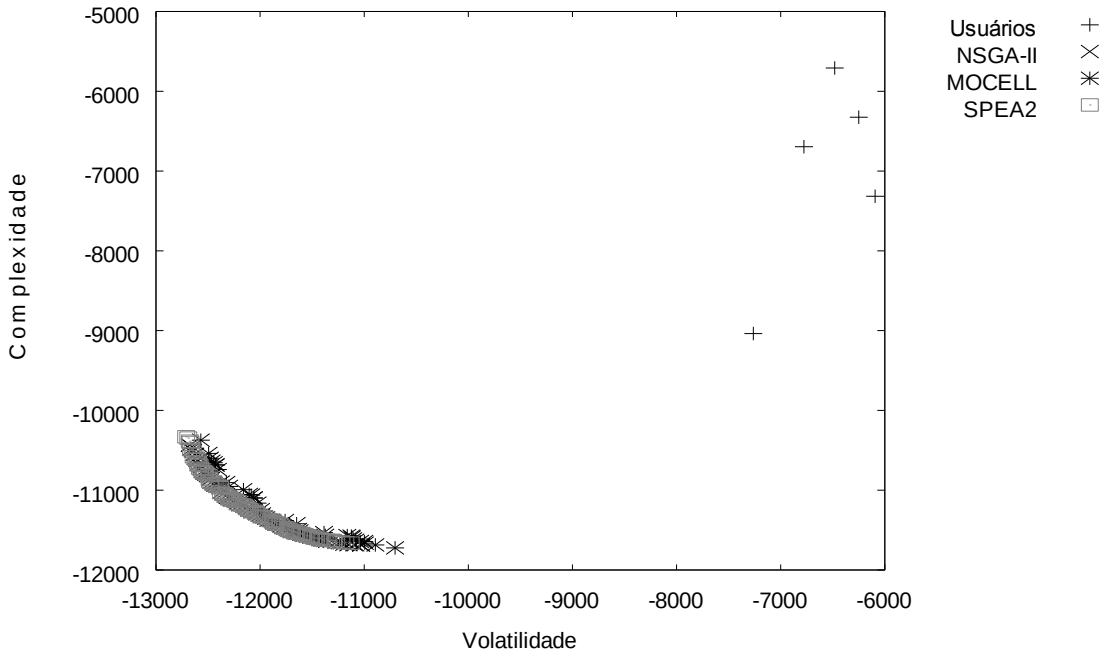

FIGURA 31 – Competitividade Humana – Priorização de Casos de Teste

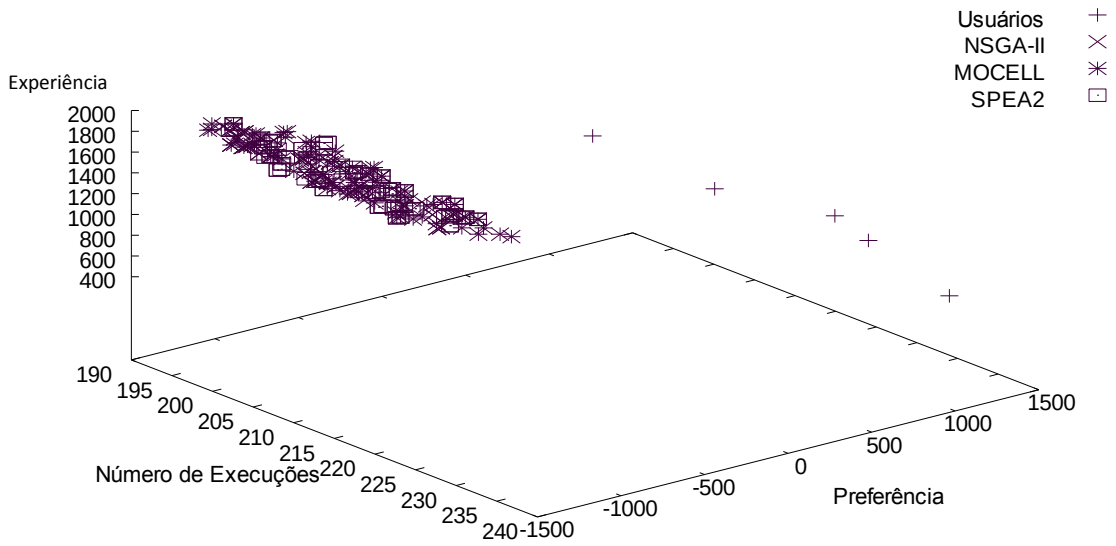

FIGURA 32 – Competitividade Humana – Alocação de Casos de Teste

Ao observar na FIGURA 30, que representa o conjunto de soluções para o problema de seleção de casos de teste, nota-se que houve uma tendência dos usuários

(humanos) de escolher uma quantidade maior de requisitos, não dando muita prioridade para a importância dos requisitos.

Para o problema da priorização de casos de teste, pode-se notar que as soluções encontradas pelos algoritmos ficaram bem próximas umas das outras, e também próximo à Frente de Pareto ótima, que é a união das melhores soluções dos três algoritmos, pois a Frente de Pareto real não é conhecida. As soluções dos usuários têm um valor bem menor para as funções objetivo, como pode ser notado na Figura 31. A mesma observação pode ser feita para o problema de alocação de casos de teste, cujo conjunto de soluções está representado na FIGURA 30. Neste caso, curiosamente, a maior diferença foi para o objetivo "preferência", o qual os usuários humanos não priorizaram com tanta ênfase.

Para comparar as soluções, foi calculado o *hypervolume* das soluções geradas pelos usuários, bem como o tempo médio para a resolução dos problemas de seleção, priorização e alocação de casos de teste. Os resultados podem ser vistos nas TABELAS 14 e 15. A média do valor do hypervolume e tempo dos algoritmos é em relação à instância BASIC_1, a mesma utilizada pelos usuários.

TABELA 14 – Comparação do Hypervolume – Análise de Competitividade

	Seleção de Casos de Teste	**Priorização de Casos de Teste**	**Alocação de Casos de Teste**
Resposta humana	0,0905	0,3299	0,00003
NSGA-II	0,8943	0,9781	0,00188
MOCell	0,3469	0,9336	0,00150
SPEA2	0,9122	0,9884	0,00184

TABELA 15 – Comparação do Tempo de Resposta (em segundos) – Análise de Competitividade

	Seleção de Casos de Teste	**Priorização de Casos de Teste**	**Alocação de Casos de Teste**
Resposta humana	2490,00	1885,8	1656,00
NSGA-II	3,39	8,25	568,21
MOCell	5,29	7,32	778,39
SPEA2	8,90	7,45	824,74

Como pode ser visto na TABELA 14, as soluções geradas pelos algoritmos multi-objetivos gerou soluções mais próximas à Frente de Pareto, devido ao valor bem maior do *hypervolume*.

Para validar os algoritmos como ferramenta viável para a busca de soluções dos problemas de seleção, priorização e alocação de casos de teste, o tempo de resposta também foi comparado ao tempo de resposta humana. O tempo de resposta humano foi, em média, 279,78 vezes maior que o tempo que o algoritmo mais lento (SPEA2) usou para encontrar um conjunto de soluções, para o problema de seleção de casos de teste. Para o problema da priorização de casos de teste, o tempo de resposta humano foi 228,58 vezes mais lento que o algoritmo NSGA-II, e 2,01 vezes mais lento que o SPEA2 no problema de alocação de casos de teste. Para este último problema, o tempo de execução dos algoritmos foi bem maior por conta da complexidade da geração da população inicial dos mesmos. O tempo de geração desta população chegou a ocupar mais de 90% do tempo da execução do algoritmo.

Finalmente, é válido destacar que algumas soluções geradas pelos usuários (humanos) foram inválidas, não respeitando as restrições dos problemas. Para o problema de seleção de casos de teste, a metade das soluções (5 soluções) foram inválidas, e para o problema de priorização de casos de teste, 2 das 7 soluções foram inválidas. O cálculo do *hypervolume* e do tempo levou em consideração apenas as soluções válidas geradas pelos usuários.

5.8 Considerações Finais

Este capítulo apresentou os resultados dos experimentos, de modo a validar a abordagem proposta. Os três experimentos projetos e implementados foram detalhados. O capítulo seguinte discutirá as conclusões e apresentará idéias para trabalhos futuros com a abordagem proposta nesta dissertação.

6 CONSIDERAÇÕES FINAIS

Como dito anteriormente, para que o processo de testes realmente seja um diferencial no desenvolvimento do software, contribuindo para a boa qualidade do produto final, é necessário um bom planejamento dos testes, inclusive da execução dos casos de teste. Esta dissertação propõe uma abordagem integrada, interativa e multi-objetiva dos problemas de seleção, priorização e alocação de casos de teste.

As principais contribuições desta dissertação são:

1. Formulação de maneira integrada, interativa e multi-objetiva dos problemas de seleção, priorização e alocação de casos de teste.

2. Comparação de três algoritmos multi-objetivos para a abordagem proposta.

3. Análise de sensibilidade da abordagem proposta, em relação aos seus parâmetros.

4. Análise de competitividade humana dos algoritmos multi-objetivos utilizados nos experimentos, para a abordagem proposta.

Foram projetados e implementados três experimentos, e cada um dos experimentos responde a uma das questões constantes na subseção 5.1. A seguir, as conclusões serão apresentadas para cada uma dessas questões.

6.1 Conclusões para Q1

A questão Q1 foi respondida pelo experimento 1. O objetivo era saber qual das três técnicas escolhidas para implementação é a mais adequada à abordagem proposta. De uma maneira geral, os resultados do experimento mostrou que o algorimo SPEA2 obteve melhor desempenho, em relação ao *hypervolume*. A maioria das soluções geradas por este algoritmo compôs a Frente de Pareto ótima, formada pelas melhores soluções dos três algortimos. O *hypervolume* do SPEA2 foi melhor em

68,97% das instâncias no problema de seleção de casos de teste, 72, 41% no problema de priorização de casos de teste, e 58,62% no problema de alocação de casos de teste. A desvantagem deste algoritmo foi seu tempo de execução, que é, em média, 1,32 vezes mais lento que o MOCell e 2,1 vezes mais lento que o NSGA-II.

O algoritmo MOCell gerou poucas soluções para os três problemas, muitas vezes chegando a gerar apenas uma solução. Além disso, na maioria das vezes, poucas soluções geradas por este algoritmo compuseram a Frente de Pareto ótima. Uma vantagem deste algoritmo é que ele gerou um valor de *spread* melhor que os outros dois para o problema de seleção de casos de teste, sendo superado apenas pelo algoritmo randômico.

O NSGA-II foi o segundo melhor, considerando-se o *hypervolume*. Além disso, obteve o melhor tempo de execução para os problemas de seleção e alocação de casos de teste, com 93,10% e 86,21% das instâncias. Em relação ao número de soluções deste algoritmo que fizeram parte da Frente de Pareto ótimo, foram menos soluções que o SPEA2 e mais que o MOCell.

Os três algoritmos superaram o algoritmo randômico em termos de qualidade das soluções, como pode ser visualizado nas figuras apresentadas na Seção 5. As soluções geradas pelo algoritmo randômico estão bem distantes da Frente de Pareto real, formada pelo conjunto das melhores soluções obtidas pelos três algoritmos.

6.2 Conclusões para Q2

O segundo experimento realiza uma análise de sensibilidade, variando os parâmetros de entrada e analisando as soluções geradas. Os parâmetros a serem variados foram selecionados a partir da formulação matemática dos três problemas. Os resultados confirmaram os resultados esperados.

Como exemplo, foi mostrado que, ao aumentar o número de casos de teste e manter o número de requisitos, mais casos de teste cobrem um mesmo requisito,

gerando um aumento no número de requisitos cobertos e consequentemente o valor de importância da suíte de testes, no problema de seleção de casos de teste. Além deste, outros resultados esperados foram confirmados neste experimento.

6.3 Conclusões para Q3

No experimento 3, os resultados obtidos pelos algoritmos multi-objetivos foram comparados com respostas de gerentes de teste.

Como pode ser visto nas figuras da Seção 5 referentes a este experimento, as soluções geradas pelos usuários estão distantes da Frente de Pareto ótima. Além disso, o hypervolume e o tempo de execução foram bem piores que os valores para os algoritmos multi-objetivos. O tempo de resposta humano foi aproximadamente 279,78 vezes maior que o do algoritmo mais lento (SPEA2) no problema de seleção de casos de teste, 228,58 vezes mais lento no problema de priorização de casos de teste, e 2,01 vezes mais lento no problema de alocação de casos de teste.

É importante destacar que algumas soluções geradas pelos usuários foram inválidas. Para o problema de seleção de casos de teste, este número chegou a 50% das soluções humanas.

6.4 Limitações

Pode-se citar quatro limitações para a abordagem proposta. A primeira limitação é a ausência de uma interface para a entrada de dados. São muitos dados de entrada: informações sobre requisitos, casos de teste, testadores e histórico de execução. Sem uma interface gráfica, uma pessoa ou ferramenta deve gerar arquivos de texto no formato lido pela ferramenta desenvolvida para a abordagem proposta.

A segunda limitação é o tempo para levantar os valores de cada parâmetro de entrada da abordagem proposta. Como descrito no parágrafo anterior, são muitos dados, e à medida que aumenta o número de requisitos e casos de teste, aumenta-se

muito a quantidade de dados a serem informados. Caso não exista uma base de dados ou arquivo que armazene estas informações, o tempo para realizar o levantamento dos mesmos é alto.

A terceira limitação são os objetivos levados em consideração nesta abordagem. Muitos objetivos utilizados em outras abordagens de outros trabalhos não estão contemplados nesta abordagem.

Por último, a quarta limitação se refere ao uso de dados gerados randomicamente na abordagem. Os resultados seriam mais realistas se os dados utilizados fossem reais.

6.5 Ameaças à Validade

Pode-se citar duas ameaças à validade dos experimentos. A primeira ameaça é a configuração dos algoritmos. Poderiam ser testados mais combinações de taxas de mutação e recombinação, bem como outros métodos de recombinação e mutação.

A segunda ameça é o uso de dados gerados, visto que o algoritmo que gerou os dados randomicamente pode tê-lo feito de maneira viciada ou incondizente com a realidade das empresas.

6.6 Trabalhos Futuros

Esta subseção descreve alguns trabalhos futuros que podem ser desenvolvidos a partir desta dissertação.

Um primeiro possível trabalho futuro é comparar os algoritmos utilizados nesta dissertação com outros algoritmos multi-objetivos que não sejam algoritmos evolucionários, como um algoritmo de colônia de formigas multi-objetivo ou têmpera simulada.

Outro trabalho que poderia ser desenvolvido pode levar em consideração alterações na formulação matemáticas dos problemas, como por exemplo, considerar

vários precedentes para um caso de teste, ou limitar a quantidade de casos de teste alocados por testador, ou considerar a taxa de detecção de falhas dos casos de teste, ou ainda considerar o custo do teste como objetivo.

Um terceiro trabalho seria o aprofundamento na análise de competitividade, para avaliar se o nível experiência do usuário da abordagem influencia na qualidade das suas respostas.

Um outro trabalho futuro possível seria coletar dados reais de um projeto em uma empresa de *software* e executar os mesmos experimentos para estes dados.

Por último, poderia-se desenvolver uma interface gráfica para a abordagem proposta, possibilitando mais agilidade na informação dos parâmetros de entrada.

REFERÊNCIAS BIBLIOGRÁFICAS

AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W. **Incremental Regression Testing**. In: Proceedings of the Conference on Software Maintenance, p. 348-357, 1993.

ALANDER, J. T., MANTERE, T., TURUNEN, P. **Genetic Algorithm Based Software Testing**. In: Proceedings of International Conference of Artificial Neural Nets and Genetic Algorithms, p. 325-328, 1997.

ALBRECHT, A. J. **Measuring Application Development Productivity**. In: Proceedings of IBM Applications Development Svmp, IBM Corp., Monterey, CA, p. 83, 1979.

ANTONIOL, G., DI PENTA, M., HARMAN, M. **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, p. 240-249, 2005.

ANTONIOL, G., KPODJEDO, S., RICCA, F., GALINIER, P. **Evolution and Search Based Metrics to Improve Defects Prediction**. In: Proceedings of the 1st International Symposium on Search Based Software Engineering, p. 23-32, 2009.

BAGNALL, A. J., RAYWARD-SMITH, V. J., WHITTLEY, I. M. **The Next Release Problem.** Information and Software Technology, Elsevier, v. 43, n. 14, p. 883-890, 2001.

BARRETO, A., BARROS, M. O., WERNER, C. M. L. **Staffing a software project: A constraint satisfaction and optimization-based approach**. Computers & Operations Research, v.35, n. 10, p. 3073-3089, 2008.

BASTOS, A., CRISTALLI, R., MOREIRA, T., RIOS, E. **Base de Conhecimento em Teste de Software**. Martins Fontes, segunda edição, 2007.

BITNER, J. R., REINGOLD, E. M. **Backtrack Programming Techniques**. Communications, v. 18, n. 11, p. 651-655, 1975.

BRIAND, L. C., LABICHE, Y., SHOUSHA, M. **Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems**. Genetic Programming and Evolvable Machines, v. 7, n. 2, p. 145–170, 2006.

COELLO, C. A. C., LAMONT, G. B., VAN VELDHUIZEN, D. A. **Evolutionary Algorithms for Solving Multi-Objective Problems**. segunda edição, Springer, 2006.

COLARES, F., SOUZA, J., CARMO, R., PÁDUA, C., MATEUS, G. R. **A New Approach to the Software Release Planning**. XXIII Brazilian Symposium on Software Engineering, p. 207-215, 2009.

COLLETTE, Y., SIARRY, P. **Multi-Objective Optimization: Principles and Case Studies**. Springer, 2003.

DARWIN, C. **On the Origin of Species**. 1859.

DAVIES, E. **The Use of Genetic Algorithms for Flight Test and Evaluation of Artificial Intelligence and Complex Software System**s. Technical Report AD-A284824, Naval Air Warfare Center, Patuxent River, 1994.

DEB, K., AGRAWAL, S., PRATAP, A., MEYARIVAN, T. **A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II**. In: Proceedings of the Parallel Problem Solving from Nature VI Conference, Springer, p. 849-858, 2000.

DEB, K. **Multi-Objective Optimization using Evolutionary Algorithms**. John Wiley & Sons, 1st Edition, 2008.

DI PENTA, M., HARMAN, M., ANTONIOL, G., QURESHI, F. **The effect of communication overhead on software maintenance project staffing: a search-based approach**. IEEE International Conference on Software Maintenance, p. 315-324, 2007.

DI PENTA, M., HARMAN, M., ANTONIOL, G. **The use of Search-Based Optimization Techniques to Schedule and Staff Software Projects: an Approach and an Empirical Study**. Software – Practice and Experience, 2009.

DUGGAN, J., BYRNE, J., LYONS, G. J. **A task allocation optimizer for software construction**. IEEE Software, V. 21, N. 3, P. 76-82, 2004.

EIBEN, A. E., SMITH, J. E. **Introduction to Evolutionary Computing**. Springer, 2003.

ELBAUM, S., ROTHERMEL, G., KANDURI, S., MALISHEVSKY, A. G. **Selecting a Cost-Effective Test Case Prioritization Technique**. Software Quality, v. 12, n. 3, P. 185-210, 2004.

GLOVER, F., LAGUNA, M. **Tabu Search**. Citeseer, 1993.

GLOVER, F., KOCHENBERGER, G. A. **Handbook of Metaheuristics**. Springer, 1st edition, 2003.

GUPTA, R., HARROLD, M. J., SOFFA, M. L. **Program Slicing-Based Regression Testing Techniques**. Software Testing, Verification and Reliability, v. 6, n. 2, p. 83-111, 1996.

HARALICK, R. M., ELLIOTT, G. L. **Increasing tree search efficiency for constraint satisfaction problems**. Artificial Intelligence, v. 14, n. 3, p. 263-313, 1980.

HARMAN, M. **The Current State and Future of Search Based Software Engineering**. Proceedings of the Future of Software Engineering, 2007.

HARMAN, M., JONES, B. F.**Search-based software engineering**. Information and Software Technology , v. 43, n. 14, p. 833-839, 2001.

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

HARROLD, M. J., GUPTA, R., SOFFA, M. L. **A Methodology for Controlling the Size of a Test Suite**. ACM Transactions on Software Engineering and Methodology, V. 2, N. 3, p. 270-285, 1993.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. The University of Michigan Press, 1975.

KIRKPATRICK, S., GELLAT, D. C., VECCHI, M. P. **Optimization by Simulated Annealing**. Science, v. 220, n. 4598, pp. 671-680, Citeseer, 1983.

KUPERBERG, M., OMRI, F. **Using Heuristics to Automate Parameter Generation for Benchmarking of Java Methods**. Electronic Notes in Theoretical Computer Science, v. 253, n. 1, p. 57-75, 2009.

LAWLER, E. L., WOOD, D. E. **Branch-and-Bound Methods: A Survey**. Operations Research, v. 14, n. 4, p. 699-719, 1966.

LI, R., CHAUDRON, M. R. V., LADAN, R. C. **Towards Automated Software Architectures Design using Model Transformations and Evolutionary Algorithms**. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10), p. 1333, 2010.

LI, Z., HARMAN, M., HIERONS, R. M. **Search Algorithms for Regression Test Case Prioritization**. IEEE Transactions on Software Engineering, v. 33, n. 4, P. 225-237, 2007.

MAIA, C. L. B., CARMO, R. A. F., CAMPOS, G. A. L., SOUZA, J. T. **A Reactive GRASP Approach for Regression Test Case Prioritization**. In: Proceedings of XL Simpósio Brasileiro de Pesquisa Operacional (SBPO 2008), 2008.

MAIA, C. L. B., CARMO, R. A. F., FREITAS, F. G., CAMPOS, G. A. L., SOUZA, J . T. **A Multi-Objective Approach for the Regression Test Case Selection Problem**. In: Proceedings of XLI Simpósio Brasileiro de Pesquisa Operacional (SBPO 2009), p. 1824-1835, 2009.

MAIA, C. L. B., CARMO, R. A. F., FREITAS, F. G., CAMPOS, G. A. L., SOUZA, J. T. **Automated Test Case Prioritization with Reactive GRASP**. Advances in Software Engineering, 2010.

MANSOUR, N., BAHSOON, R., BARADHI, G. **Empirical Comparison of Regression Test Selection Algorithms**. Journal of Systems and Software, v. 57, n. 1, p. 79-90, Elsevier, 2001.

MATHUR, A. P. **Foundations of Software Testing**. Pearson, 2008.

MILLER, W., SPOONER, D. L. **Automatic generation of floating-point test data**. IEEE Transactions on Software Engineering, IEEE, p. 223-226, 1976.

MYERS, G. **The Art of Software Testing**. John Wiley & Sons, Inc, 2nd Edition, 2004.

NEBRO, A., DURILLO, J., COELLO, C. C., LUNA, F, ALBA, E. **A Study of Convergence Speed in Multi-Objective Metaheuristics**. Parallel Problem Solving from Nature, Springer, p. 763-772, 2008.

NEBRO, A. J., DURILLO, J. J., LUNA, F., DORRONSORO, B., ALBA, E. **MOCell: A cellular genetic algorithm for multiobjective optimization.** International Journal of Intelligent Systems, v. 24, n. 7, p. 726-746, 2009.

PRESSMAN, R. S. **Engenharia de Software**. Mc Graw Hill, quinta edição, 2001.

REEVES, C. R. **Modern Heuristic Techniques for Combinatorial Problems**. John Wiley & Sons, 1993.

ROTHERMEL, G., UNTCH, R. H., CHU, C. e HARROLD, M. J. **Prioritizing Test Cases for Regression Test**. IEEE Transactions on Software Engineering, v. 27, n. 10, p. 929-948, 2001.

SOUZA, J. T., MAIA, C. L. B., FREITAS, F. G., COUTINHO, D. P. **The Human Competitiveness of Search Based Software Engineering**. In: Proceedings of Second International Symposium on Search based Software Engineering (SSBSE 2010), p. 143-152, 2010.

TAKAGI, H. **Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation**. In: Proceedings of IEEE, v. 89, n. 9, p. 1275-1296, 2001.

TALLAM, S., GUPTA, N. **A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization**. In: Proceedings of 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, p. 35-42, 2006.

WALCOTT, K. R., SOFFA, M. L., KAPFHAMMER, G. M., ROOS, R. S. **Time-Aware Test Suite Prioritization**. In: Proceedings of the International Symposium on Software Testing and Analysis, p. 1-12, 2006.

YOO, S., HARMAN, M. **Pareto Efficient Multi-Objective Test Case Selection**. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA'07), p. 140-150, 2007.

YOO, S., HARMAN, M. **Using Hybrid Algorithm for Pareto Efficient Multi-Objective Test Suite Minimisation**. Journal of Systems and Software, v. 83, n. 4, p. 689-701, 2010.

ZITZLER, E., THIELE, L. **Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach**. IEEE Transactions on Evolutionary Computation v. 3, n. 4, p. 257–271, 1999.

ZITZLER, E., DEB, K, THIELE, L. **Comparison of Multiobjective Evolutionary Algorithms: Empirical Results**. Evolutionary Computation, v. 8, n. 2, p. 173-195, 2000.

ZITZLER, E., LAUMANNS, M., THIELE, **SPEA2: Improving the Strength Pareto Evolutionary Algorithm**, 2001.