UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MÉTODO PARA SELEÇÃO DE CASOS DE TESTE DE UNIDADE PARA ALTERAÇÕES CRÍTICAS

FÁBIO DE ALMEIDA FARZAT

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2011

MÉTODO PARA SELEÇÃO DE CASOS DE TESTE DE UNIDADE PARA ALTERAÇÕES CRÍTICAS

FÁBIO DE ALMEIDA FARZAT

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

_____
Márcio de Oliveira Barros, D.Sc. – UNIRIO

_____
Alexandre Albino Andreatta, D.Sc. – UNIRIO

_____
Guilherme Horta Travassos, D.Sc. – UFRJ

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 2011

A minha mãe.
Aos meus amigos.

Agradecimentos

A Deus, por todas as oportunidades, enriquecedoras ou não, que ajudaram a me guiar na direção desse objetivo.

A minha mãe Aleuda, pelo incentivo, confiança, amizade e amor. Esse apoio fundamental permitiu alcançar esse objetivo.

Ao meu orientador Prof. Márcio Barros, por acreditar no meu potencial, por todas as lições e direcionamento durante a realização deste trabalho.

Ao Prof. Alexandre Andreatta, pelo acompanhamento e ajuda no direcionamento deste trabalho.

Ao meu ex-gerente Antônio Gonçalves e à Prudential do Brasil, pela concordância e apoio para realizar o curso de mestrado.

Ao meu atual Superintendente Flávio Dias e à Mongeral Aegon, pela cessão dos dados de projeto para o experimento desse trabalho e apoio para concluir o curso de mestrado.

Resumo da Dissertação apresentada ao PPGI/UNIRIO como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MÉTODO PARA SELEÇÃO DE CASOS DE TESTE DE UNIDADE PARA ALTERAÇÕES CRÍTICAS

FÁBIO DE ALMEIDA FARZAT

AGOSTO/2011

Orientador: Márcio de Oliveira Barros

Testes de regressão são uma das várias técnicas que a Engenharia de Software propõe para minimizar erros e, consequentemente, aumentar a qualidade do produto. Porém, o custo de executar todos os testes a cada modificação de um software pode ser muito grande. Mesmo em empresas desenvolvedoras de software com um processo institucionalizado, a atividade de testes pode ser cortada ou reduzida devido a pressões do negócio. Esse comportamento é comum na indústria, onde muitos erros são encontrados diretamente em ambiente de produção e causam prejuízos à operação. Para minimizar esse risco, algum tipo de garantia precisa ser definida, como testar todo o código afetado pela alteração. Porém, separar manualmente os casos de teste relacionados com essa alteração pode consumir muito tempo. Visto que se trata de uma alteração crítica, pois foi realizada diretamente em ambiente de produção, tempo é um critério decisivo. Portanto, precisamos separar os casos de teste que possam executar dentro de um limite de tempo pré-estabelecido e que cubram partes críticas do software modificadas nas últimas alterações. O objetivo desse trabalho é propor um modelo, um procedimento de coleta de dados e um método heurístico (Algoritmos Genéticos) para encontrar soluções boas (próximas da ótima) para selecionar testes adequados para alterações críticas. Um estudo experimental foi conduzido para avaliar a proposta e conclui-se que uma busca heurística é necessária quando o tempo disponível não é suficiente para executar uma grande parte da suíte de testes.

Abstract of Dissertation presented to PPGI/UNIRIO as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

UNIT TEST CASE SELECTION METHOD TO SUPPORT CRITICAL CHANGES

FÁBIO DE ALMEIDA FARZAT

AUGUST/2011

Advisor: Márcio de Oliveira Barros

Regression tests are one of several techniques that the Software Engineering proposes to minimize errors and increase product quality. Due to their high cost, testing activities are often given less effort than required to fully evaluate the correctness of a software system. They may be almost eliminated from the development process for one or more releases in case of business pressure to deliver a new release as soon as possible. This behavior is even more common when errors are found in production, thus blocking the perfect execution of business transactions. In such situations, corrective activities are often executed in the production environment itself, thereby releasing the patched version of the software without proper tests, even though the risk of inserting new defects into the system may be worse than awaiting the time required for testing the release. Some sort of guarantee must be defined to minimize the risk of an incorrect patch, or one that introduces new defects. For instance, one may test all code referring to features changed in the production environment. However, manually separating test cases related to a specific set of changes can be time-consuming. The goal of this research was to propose a model, a data collect procedure and a Heuristic method (Genetic Algorithms) to find good-enough solutions to test case selection method for critical changes. An experimental study was conducted to evaluate the proposal and concluded that a heuristic search is necessary when the time available is not sufficient to run a large part of the test suite.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

INTRODUÇÃO E MOTIVAÇÃO

1.1. Motivação

A demanda por sistemas de informação por parte das organizações tem aumentado significativamente. Esse fato reflete-se no número de projetos de software que o mercado em geral tem produzido. Esses novos sistemas precisam atender a requisitos funcionais e não funcionais cada vez mais complexos, integrando-se com outros sistemas, prevendo adaptações e mudanças, em um espaço de tempo cada vez menor. Para tornar possível produzir software com qualidade dentro desse contexto a grande maioria das organizações usa práticas de Engenharia de Software.

Engenharia de Software pode ser definida como uma disciplina que aplica os princípios de engenharia com o objetivo de produzir software com alta qualidade a baixo custo (PRESSMAN 2005). Porém, mesmo utilizando as melhores práticas de Engenharia de Software, defeitos no produto podem ocorrer. Em Engenharia de Software existem práticas relacionadas à garantia da qualidade, que é um processo sistemático que focaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos (BARTIÉ 2002). Dentro do processo de garantia da qualidade existem práticas agrupadas em Validação, Verificação e Testes. Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades, como por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação (MALDONADO 1991).

As atividades de teste ocorrem durante o processo de desenvolvimento e segundo (MALDONADO 1991) organizam-se em três fases: testes de unidade, de integração e de sistema. Testes de unidade concentram esforços na procura de falhas na menor unidade lógica do projeto. Essa unidade pode ser um módulo e até

mesmo uma classe. Os testes de integração são atividades sistemáticas aplicadas durante a integração dos módulos do sistema, de forma a procurar falhas na interface entre esses módulos. Os testes de sistema tentam identificar falhas de funções e características de desempenho que não estejam de acordo com as especificações do software.

Considerada como sendo uma das mais onerosas atividades de todo o processo de desenvolvimento, chegando a custar 50% do projeto (HARROLD 1991), a atividade de testes tem o objetivo de revelar falhas, embora a maioria das equipes encare testes como a atividade de provar que determinada parte do software funciona corretamente (BARTIÉ 2002). Com essa abordagem os testes perdem bastante qualidade e muitos defeitos passam no produto sem ser notados (BARTIÉ 2002). Portanto fica claro que critérios econômicos e de negócio geram impacto diretamente no processo de testes e fazem com que menos casos de testes sejam planejados e executados.

Este comportamento é ainda mais comum quando são encontrados defeitos em ambiente de produção, defeitos estes que podem bloquear a correta execução de operações do negócio suportado pelo sistema. Em tais situações, as atividades de manutenção corretiva onde nenhum novo requisito é incluído, são freqüentemente executadas no ambiente de produção, lançando a versão corrigida do software em produção sem testes apropriados. Dada a necessidade de resolver o defeito, a equipe realiza a mudança e implanta imediatamente. Porém, o risco de inserir mais defeitos no sistema e causar danos colaterais para a operação pode ser ainda pior do que não testar uma mudança crítica.

Para minimizar o risco de um *patch* incorreto ou ainda de incluir mais defeitos, algum tipo de verificação deve ser aplicada sobre o software mesmo em uma situação crítica de atualização. Por exemplo, pode-se testar todo o código escrito ou alterado como parte da alteração crítica. Para tal assume-se a premissa de que o sistema já possui testes que cubram essas alterações. É comum que sistemas de grande porte possuam uma boa suíte de testes unitários, os quais são escritos na fase de desenvolvimento do sistema. No entanto, separar manualmente os casos de teste para um conjunto específico de mudanças também pode ser uma tarefa demorada. Uma vez que estamos tratando de uma alteração crítica, o tempo consumido é um critério decisivo. Outra desvantagem é que considerar apenas o código novo ou alterado pode não garantir que outras partes essenciais do sistema continuem consistentes após a alteração.

Assim, é preciso selecionar um conjunto de casos de teste unitários que possa ser executado em um determinado período de tempo e que cubra as funcionalidades mais importantes que foram afetadas pela mudança crítica, de acordo com a prioridade de cada funcionalidade. Selecionar um conjunto apropriado desses testes é um problema de otimização e, em sistemas com um conjunto de testes de grande porte, talvez não exista um método exato capaz de encontrar a solução ótima em tempo razoável.

A motivação deste trabalho está fortemente inserida no contexto das grandes empresas do mercado financeiro brasileiro, produtoras de seus próprios softwares operacionais (Bancos, Seguradoras, Entidades Previdenciárias, etc.). O mercado financeiro brasileiro sofre uma intensa regulação governamental e esse fato exige rápidas alterações nos sistemas para adequação às mudanças na legislação. Para essas empresas os prazos para adequação são críticos e por vezes algumas etapas importantes do processo de desenvolvimento de software são abandonadas visando apenas o prazo. Esse comportamento permite que vários novos defeitos sejam inseridos e permaneçam nos sistemas e causem problemas na operação dessas empresas, gerando a insatisfação das áreas de negócio (clientes das equipes de TI) e danos à imagem da empresa.

1.2. Objetivo da dissertação

Este trabalho aborda um problema recorrente na maioria das organizações que desenvolvem ou mantém sistemas de software. Nesse contexto o cenário exposto na seção anterior caracteriza um problema que nesse trabalho de pesquisa foi nomeado como problema da seleção de casos de teste unitários para alterações críticas. Este trabalho apresenta um modelo formal para este problema, um procedimento de coleta de dados para obter suas informações de forma sistemática e um método heurístico, baseado em um algoritmo genético, para propor boas soluções para o problema.

O trabalho apresenta uma proposta de solução que consiste de um método de otimização baseado no modelo formal do problema e no procedimento de coleta de dados. Esses dados são transformados em instâncias do problema, juntamente com algumas configurações sob as quais as instâncias podem ser analisadas. As instâncias e suas configurações são submetidas a uma técnica de otimização a fim de buscar uma suíte de teste que possa cobrir de maneira próxima da ótima as alterações realizadas no software sob testes.

Com a aplicação da técnica proposta espera-se diminuir o risco de defeitos que passam para o ambiente de produção do sistema, aumentando sua confiabilidade e diminuindo os prejuízos operacionais. No entanto essa conseqüência não será avaliada – o estudo experimental apresentando como parte desse trabalho restringe-se a avaliar se a busca heurística proposta é efetivamente necessária para encontrar boas soluções para o problema. Neste sentido o estudo avaliará se o problema pode ser resolvido por uma técnica de busca mais simples como a busca local ou até mesmo aleatória.

1.3 Estrutura do Texto

Este trabalho está organizado em seis capítulos. O primeiro capítulo compreende esta introdução. O segundo capítulo, Testes de Regressão e Técnicas de seleção de casos de teste, faz um resumo sobre testes de regressão, detalha os tipos de teste existentes e os tipos principais de técnicas para seleção de casos de teste: minimização de suítes, priorização de suítes e seleção específicas de casos de teste.

O terceiro capítulo, Métodos Heurísticos aplicados a Teste de Software apresenta os vários de tipos de busca heurística aplicados em testes de software e apresenta vários trabalhos relacionados. O quarto capítulo, Procedimento de Seleção de Casos de teste Críticos, apresenta o modelo formal do problema, discutindo cada entidade que foi incluída com suas relações. Descreve o procedimento de coleta de dados e como as informações coletadas serão usadas. Discute a proposta de solução baseada em técnicas de experimentação com busca heurística.

O quinto capítulo, Avaliação do Método Proposto, descreve os experimentos conduzidos, quais as questões que nortearam a pesquisa, as instâncias utilizadas nos experimentos, quais algoritmos foram escolhidos para tratar o problema e faz uma análise dos resultados. Ao fim, discute as ameaças a validade do experimento como um todo e tira algumas conclusões.

Por fim, o sexto capítulo contém a conclusão do trabalho e considerações sobre os futuros trabalhos relacionados.

CAPÍTULO 2

TÉCNICAS DE SELEÇÃO DE CASOS DE TESTE

2.1. Introdução

O propósito dos testes de regressão é aumentar a garantia de que um conjunto de alterações previamente executadas no código do sistema sob testes, não afete o comportamento esperado das partes não alteradas desse sistema (BARTIÉ, 2002). O esforço de criação e manutenção desses testes, na situação ideal em que eles são desenvolvidos e atualizados para as novas alterações, cresce em proporção compatível com o próprio software. Portanto, com todo esse crescimento, executar a suíte de teste inteira aumenta significativamente o custo total do desenvolvimento, visto que a prática de execução dos testes é feita a cada conjunto de alterações. Essa limitação motivou a geração de várias técnicas, visando reduzir o conjunto total de testes necessários de várias formas.

Das várias técnicas que estudam o problema dos testes de regressão, três das mais citadas na literatura são a *minimização*, *seleção* e *priorização de casos de teste* (YOO e HARMAN, 2009). A técnica de minimização da suíte de testes visa identificar e eliminar casos de testes obsoletos ou redundantes. A técnica de seleção de casos de teste procura um subconjunto adequado dos testes que será usado para testar as partes modificadas do sistema. Já a técnica de priorização de casos de teste visa encontrar a ordem ideal de execução dos casos de teste para maximizar alguma propriedade desejada, como por exemplo, aumentar o percentual de detecção de falhas. Conceitos sobre testes não serão abordados nesse trabalho de pesquisa, mas podem ser encontrados no trabalho de (LEUNG E WHITE, 1989), de onde foi retirada a classificação de casos de teste utilizada nesse trabalho.

Este capítulo está dividido em seis seções. A Seção 2.2 apresenta os principais conceitos para compreensão das técnicas abordadas. A Seção 2.3 apresenta a técnica de minimização. A seção 2.4 apresenta a técnica de seleção. A Seção 2.5 apresenta a técnica de priorização. A Seção 2.6 faz uma análise consolidada das principais questões abordadas neste capítulo e conclusões.

2.2 Classificação de casos de teste

Este capítulo aborda o tema de testes e técnicas de seleção de casos de teste. Porém, todas as técnicas abordadas possuem semelhanças, o que torna necessário detalhar claramente os conceitos utilizados, principalmente em relação à classificação dos testes.

O primeiro trabalho que apresenta uma classificação para testes de regressão é o de LEUNG E WHITE (1989). Neste trabalho, casos de testes podem ser progressivos ou corretivos. Testes progressivos são testes escritos para testar as alterações recentes feitas no sistema sob testes. Já os testes corretivos não procuram cobrir as alterações recentes, e sim testar as partes do código que não foram alteradas. Portanto, os testes corretivos são utilizados sem alterações.

No total de cinco categorias duas são usadas para classificar os testes progressivos (*Estrutural e Especificação* e três para classificar os testes corretivos (*Reutilizáveis*, *Retestáveis* e *Obsoletos)*. Os testes *Estruturais* visam cobrir a estrutural geral do código, de forma a executar ao menos uma vez cada instrução de um determinado método ou bloco de código. Os testes de *Especificação* são escritos a partir da especificação do sistema, com o objetivo de testar as novas partes do código. Os testes reutilizáveis são aqueles que testam apenas o código não alterado entre duas versões do sistema. Portanto, são desnecessários para garantir a cobertura das porções alteradas do sistema sob testes. Porém, são utilizados nos testes de regressão como forma de garantir que as partes não alteradas do código estão conformes com os antigos requisitos e que as novas alterações não afetaram o comportamento dessas partes. Testes classificados como *Retestáveis* são aqueles que cobrem as alterações recentes do sistema sob testes, de forma a garantir que essas alterações estão conformes com os novos requisitos. Por último, testes classificados como *Obsoletos* não podem mais contribuir para garantir a cobertura do sistema sob testes, devido a três fatores: (i) a relação de entrada e saída do teste não está mais correta, face às novas alterações do sistema sob testes; (ii) o teste não garante mais o que ele foi escrito para cobrir/garantir no sistema; e (iii) um teste que tem por objetivo cobrir algum ponto da estrutura/projeto do sistema já não faz mais sentido face às ultimas mudanças.

Os testes podem ainda ser classificados como positivos ou negativos. NYMAN (2010) conceitua testes como segue:

- Testes positivos: testes que visam demonstrar que um módulo de código-fonte faz o que deve fazer, de acordo com sua especificação. Por exemplo, um teste positivo pode verificar se uma rotina gera certo resultado quando um conjunto de valores em particular é lançado como parâmetro de entrada.
- Testes negativos: testes que visam demonstrar que um módulo de código-fonte não faz nada que não se deve fazer, de acordo com as restrições especificadas nos documentos de projeto. Estes testes são desenvolvidos com a intenção de "quebrar" o sistema. Um exemplo de comportamento inesperado é quando o sistema exibe uma mensagem quando não deveria ou retorna um valor que não era esperado.

2.3 Problema de Minimização de Suítes de Teste

As técnicas de minimização de suítes de teste, também conhecidas como técnicas de redução de suítes, visam reduzir o tamanho total da suíte eliminando casos de testes redundantes ou obsoletos. As técnicas possuem duas abordagens quanto ao subconjunto escolhido. Casos de teste considerados redundantes ou obsoletos podem ser excluídos definitivamente da suíte ou apenas não constarem no subconjunto final proposto pela técnica. ROTHERMEL et al. (2002) definem formalmente o problema de minimização como:

Dado: Uma suíte de testes ***T***, um conjunto de requisitos de testes $R \{r_1, ..., r_n\}$, que precisam ser satisfeitos por um teste adequado e subconjuntos de $T \{ t_1, t_2, ..., t_n\}$, cada um associado a um r_i, de forma que qualquer t_j associado a T_i pode satisfazer o requisito r_i.

Problema: Encontrar um subconjunto representativo ***T'*** que satisfaça todo conjunto de requisitos ***R***.

O problema de minimização pode ser modelado como o problema da cobertura mínima de vértices, clássico problema de otimização do tipo NP - difícil. Porém, para que a comparação seja verdadeira, uma condição precisa ser adicionada a definição do problema: para cada $\mathbf{r_i}$, de ***R***, ao menos um caso de teste deve satisfazê-lo. Na prática essa condição pode não ser satisfeita, tornando o problema um pouco mais complexo.

Segundo SHIN e HARMAN (2009) a maior parte das técnicas de minimização de suíte de testes propostas utilizam heurísticas comumente utilizadas no problema da cobertura mínima de vértices. As abordagens atuais seguem o conceito do refatoração de código (FOWLER 1999), porém aplicado ao código dos testes. O código dos próprios testes sofre revisões visando diminuir a quantidade e a complexidade dos testes. Avalia-se se um novo teste substitui um ou mais testes antigos para descartá-los. Um problema dessa técnica é o fato de que a suíte de testes pode perder o percentual de cobertura ou a capacidade de descoberta de falhas, se os testes removidos não forem totalmente cobertos pelos novos.

2.4 Problema de Seleção de Casos de Teste

As técnicas de seleção de casos de teste são similares às técnicas de minimização descritas anteriormente. O objetivo de ambas é encontrar um subconjunto do conjunto de casos de teste original que será usado para garantir um *release* do sistema sob testes. A diferença entre elas está no fato de que, na técnica de seleção, o critério para escolha dos casos de teste que irão compor o subconjunto final é apoiado nas últimas alterações no código. Portanto, a técnica visa descobrir o subconjunto mínimo de testes ligados às modificações feitas no sistema sob testes.

Para formalizar o problema, ROTHERMEL e HARROLD (1996) introduziram o conceito de modificação. Uma modificação é uma diferença entre duas versões de um sistema, onde ***P*** é o sistema original e ***P'*** é o sistema com as modificações. Rothermel e Harrold definem uma modificação como um caso de teste que cobre uma alteração, desde que ***P(t) ≠ P'(t)***. A definição formal do problema de seleção de casos de testes, segundo ROTHERMEL e HARROLD (1996) é:

Dado: Um programa ***P***, a versão modificada de ***P***, ***P'***, e uma suíte de testes ***T***.

Problema: Encontrar um subconjunto de ***T***, ***T'***, que teste ***P'***.

Com a definição do problema fica claro que basta selecionar um subconjunto de casos de teste que cubra as modificações que geraram ***P'***. Porém, em seu trabalho (ROTHERMEL e HARROLD, 1996) consideram essa uma abordagem fraca, pois nesse subconjunto que cobre todas as modificações teríamos ainda dois tipos de casos de teste para analisar:

Corretos: Casos de teste que cobrem modificações e geram uma saída adequada, correta.

Obsoletos: Casos de teste que apesar de cobrirem alguma modificação, não geram mais saídas adequadas para garantir ***P'***. Portanto, estes casos de teste precisam ser corrigidos ou descartados.

Com as definições de casos de testes corretos e obsoletos temos uma noção mais clara de que, no caso da aplicação da técnica de seleção de casos de teste, não se trata apenas de encontrar todos os casos de teste que cobrem ***P'***, e sim de determinar dentre eles quais de fato contribuem para aumentar a detecção de falhas. Podemos então concluir que os casos de teste a serem identificados devem, além de cobrir as modificações, aumentar o percentual total da suíte ***T'*** em detecção de falhas.

Baseadas nessas formulações, várias pesquisas foram desenvolvidas onde a principal diferença está na forma como se determina uma modificação e como se classificam os testes. Algumas técnicas se baseiam em análise de fluxo de dados, análise de fluxo de controle, gráficos de dependência do programa, gráficos de dependência sistêmicos, cortes de programas e execução simbólica, diferença textual, entre outras. As próximas subseções tratarão algumas dessas técnicas expondo seus pontos fortes e fracos.

2.4.1 Análise do Fluxo de Dados

Técnicas de seleção baseadas em análise do fluxo de dados visam identificar dados novos, modificados ou excluídos em ***P'***, e selecionar os casos de teste que estão ligados a esses dados (TAHA AB et. al. 1989).

Segundo YOO e HARMAN (2009), existem técnicas nessa linha que combinam minimização e priorização com o objetivo de diminuir o custo da técnica de seleção de casos de teste, visto que analisar todo o fluxo de dados é muito trabalhoso.

Uma das desvantagens comum às técnicas que se apóiam no conceito da análise do fluxo de dados é o fato de não conseguirem detectar mudanças que não são perceptíveis no fluxo. Um exemplo é a inclusão de um novo método que não possui parâmetros de entrada e não chama nenhum método ou algum dos métodos modificados. Como aparentemente não há mudança no fluxo de dados, por não afetar nenhuma saída de nenhum método, testes desse novo método não serão selecionados.

2.4.2 Análise de grafo sistêmico

ROTHERMEL e HARROLD (1993) criaram uma técnica baseada na análise de grafos sistêmicos, grafos de dependências (Control - CDG, Program - PDG, System - SDG) e o grafo de fluxo de controle do sistema. Na figura 1 dois grafos de controle sistêmico de diferentes versões de um sistema são apresentados como exemplo. São gerados os grafos de controle de dependências (CDG) de ***P*** e ***P'*** e, para cada nó lexicamente diferente, os casos de testes que executam todos os nós predecessores são selecionados.

Essa técnica foi proposta inicialmente para cobrir modificações entre funções e não as modificações internas sofridas por estas funções, pois segundo os autores seria computacionalmente custoso tratar todas as modificações internas nas funções alteradas. Essa característica é uma limitação da técnica, que posteriormente evoluiu para cobrir modificações internas dos métodos (ROTHERMEL e HARROLD, 1994), inclusive estendendo sua análise a software orientado a objeto.

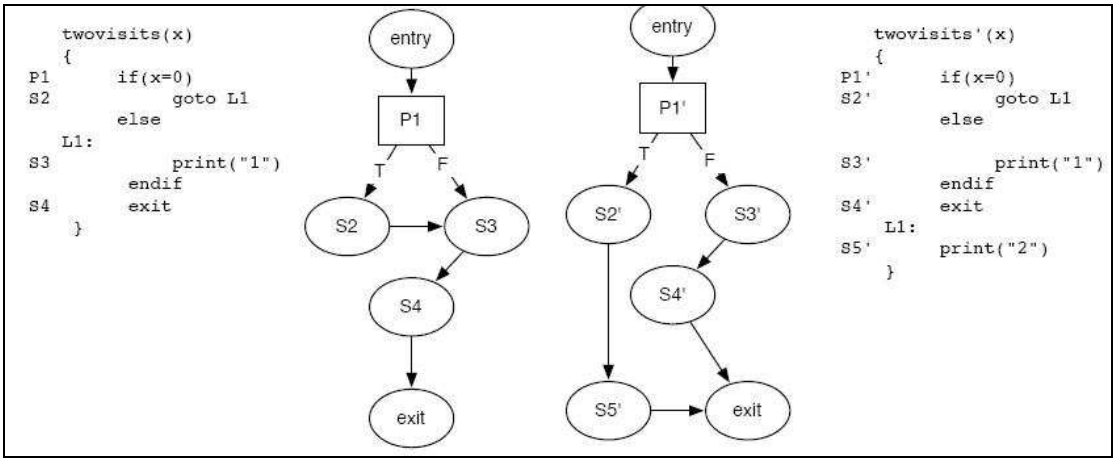

Figura 1. Gráfico de fluxo de controle **(ROTHERMEL e HARROLD, 1994)**

2.4.3 Diferença de Texto

Essa técnica utiliza o código do sistema sob testes para detectar modificações. A comparação dos arquivos de código-fonte que compõem a solução de **P** e ***P'*** é feita por alguma ferramenta de diferença de texto, que mostra o que foi alterado em cada arquivo, conforme a Figura 2. Para cada arquivo em ***P'*** marcado como modificado, qualquer caso de teste que o cubra é selecionado. Apesar de trabalhar com uma representação física do sistema sob testes, ela é muito similar a técnica análise de grafo sistêmico descrita anteriormente.

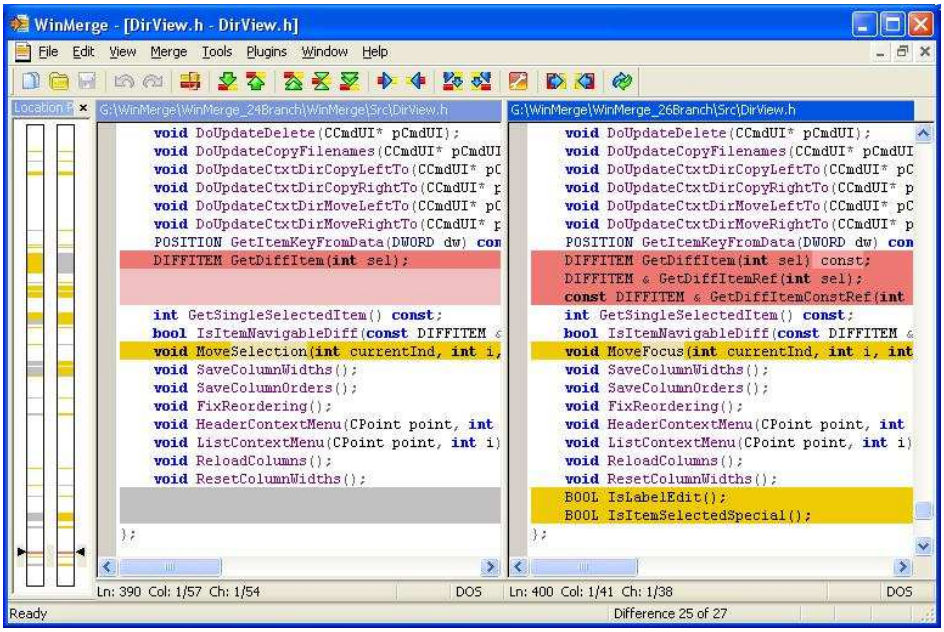

Figura 2. Tela do aplicativo *Winmerge*.

2.4.4 Seleção de Casos de Teste baseada em Modificações

CHEN et. al. (1994) divide o sistema sob testes ***P'*** em entidades, executando os casos de teste com o propósito de estabelecer uma ligação entre eles e as entidades que eles executam. A técnica também cria as entidades em ***P*** para confrontar com as de ***P'***, de forma a encontrar as entidades que foram alteradas. Ao final, temos uma lista de todas as entidades alteradas. Todos os casos de teste ligados a uma entidade alterada serão re-executados. Essa técnica transforma em entidade funções, métodos, tipos e variáveis. Dessa forma, mesmo sendo similar a análise de grafo sistêmico ela é mais abrangente pois considera também as alterações internas a essas classes.

2.4.5 Firewall

Essa técnica (WHITE L et. al. 2008) desenha um perímetro em volta dos módulos da aplicação, de forma que tudo dentro desse perímetro esteja ligado. Caso algo dentro do perímetro seja alterado, todo ele precisa ser testado novamente. A técnica utiliza uma categorização dos módulos para determinar se entre eles há uma ligação forte o suficiente para exigir o re-teste.

Sem alteração: módulo A não foi alterado, ***NoCh(A)***;

Só código alterado: módulo A possui a mesma especificação, mas seu código foi alterado, ***CodeCh(A)***;

Alteração de especificação: módulo A possui alterações na especificação, ***SpecCh(A)***.

Essa categorização visa definir quais tipos de casos de teste ligados ao módulo alterado serão executados. Testes unitários serão selecionados para alteração de apenas código, enquanto testes de integração serão selecionados em caso de modificação de especificação. Essa técnica é considerada conservadora, visto que o número de casos de teste selecionados é geralmente alto.

2.4.6 Identificação de agrupamentos (*Cluster*)

LASKI J. et al. (1992) utilizam o Grafo de Fluxo de Controle (CFG) para determinar *clusters* de alterações, conforme a Figura 3. O grafo é gerado para ***P*** e ***P'*** e para toda modificação que é agrupada, um único nó de modificação (MOD) é incluído do grafo de ***P'*** para representar o grupo de modificações. Portanto, cada caso de teste que executa o caminho de um agrupamento é selecionado para ser re-executado. Um dos pontos fortes da técnica é que ela trata qualquer alteração, desde inclusão, modificação até a exclusão de instruções. Porém, como ponto fraco, um cluster pode englobar mais do que apenas as alterações. Isso implica em ter casos de teste selecionados que não cobrem de fato uma modificação.

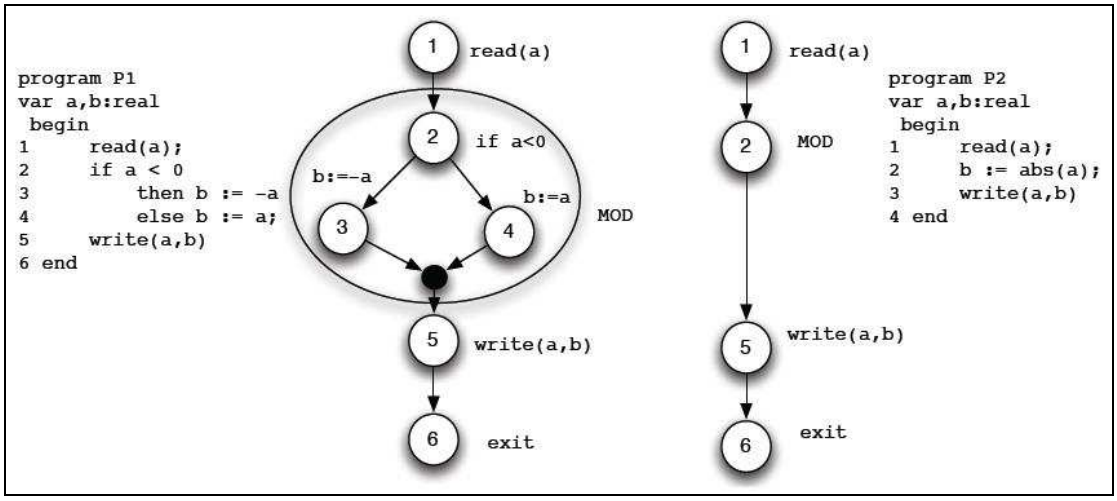

Figura 3. Análise de cluster. (LASKI J et al.,1992)

2.4.7 Seleção Baseada em Modelo

Técnicas baseadas em modelos utilizam UML para determinar as modificações e propor a seleção dos casos de teste MANSOUR e NASHAT (2009). Normalmente utilizam a classificação feita nos casos de teste (*Obsoletos*, *Retestáveis* e *Reutilizáveis*) para, seguindo as modificações encontradas, propor a seleção final dos casos de teste.

Portanto, a técnica de seleção baseada em modelo parte das alterações feitas na documentação do projeto para determinar as modificações internas do sistema e selecionar os testes ligados a essas alterações. Essa técnica utiliza a documentação técnica do sistema, além de contar com uma matriz de rastreabilidade entre essa documentação e os artefatos do sistema e ainda entre os artefatos e seus testes.

2.5 Problema de Priorização de Casos de Teste

As técnicas de priorização visam maximizar os benefícios da execução dos casos de teste por meio da descoberta da ordem ideal de execução, mesmo que essa execução seja interrompida em algum momento. Essas técnicas não envolvem seleção de casos de testes pois a aplicação é sempre para toda a suíte. As técnicas também não levam em consideração as últimas modificações no código. Formalmente, o problema da priorização dos casos de teste pode ser definido como:

> **Dado**: Uma suíte de testes ***T***, o subconjunto de permutações de ***T***, ***PT***, e uma função de ***PT*** para os números reais, $f(PT) \rightarrow \Re$.
>
> **Problema**: Encontrar um subconjunto de ***T***, ***T'***, onde $T' \in PT$ sujeito a: $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Normalmente, a ordem ideal de execução dos casos de teste procura aumentar a taxa de detecção de falhas, ou seja, ordenar decrescentemente os casos de teste por quantidade de defeitos que cada um descobre. Porém, é impossível medir a taxa de detecção de falhas de uma determinada ordenação antes de sua real execução. Nesse cenário, várias das técnicas de priorização utilizam outra variável, que tenha correlação com a taxa de detecção de falhas. Alguns exemplos serão explicados nas subseções seguintes.

2.5.1 Priorização Baseada em Cobertura

Cobertura de código é uma das métricas que possui uma correlação positiva com a taxa de detecção de falhas e é muito utilizada como critério para priorização (MALISHEVSKY et. al. 2002). A principal idéia desse tipo de priorização é que quanto mais uma ordenação cobre previamente o código, maiores são as chances de descobrir falhas rapidamente (CHU et. al. 2001).

O percentual de cobertura de um determinado teste é medido pela quantidade de linhas de código que são executadas por esse teste, dividido pelo número total de linhas de código do sistema sob testes. Apesar de a cobertura de um caso de teste

sofrer a mesma limitação da taxa de detecção de falhas (só é medida após a execução do teste), obrigatoriamente todo teste é executado após sua implementação, como forma de garantir sua completude. Portanto, nesse momento o seu percentual de cobertura pode ser armazenado para que a técnica de priorização possa ser aplicada, conforme mostra o exemplo da Figura 4 com a ferramenta de execução de testes *NCover*[1]. Essa ferramenta executa os casos de teste e coleta informações sobre cobertura e as partes do código onde cada teste passou, ou seja, que linhas de código-fonte foram visitadas por quais testes. Isso permite uma análise voltada para a seleção de teste por cobertura, além de apoiar a equipe na avaliação da completude de um determinado teste.

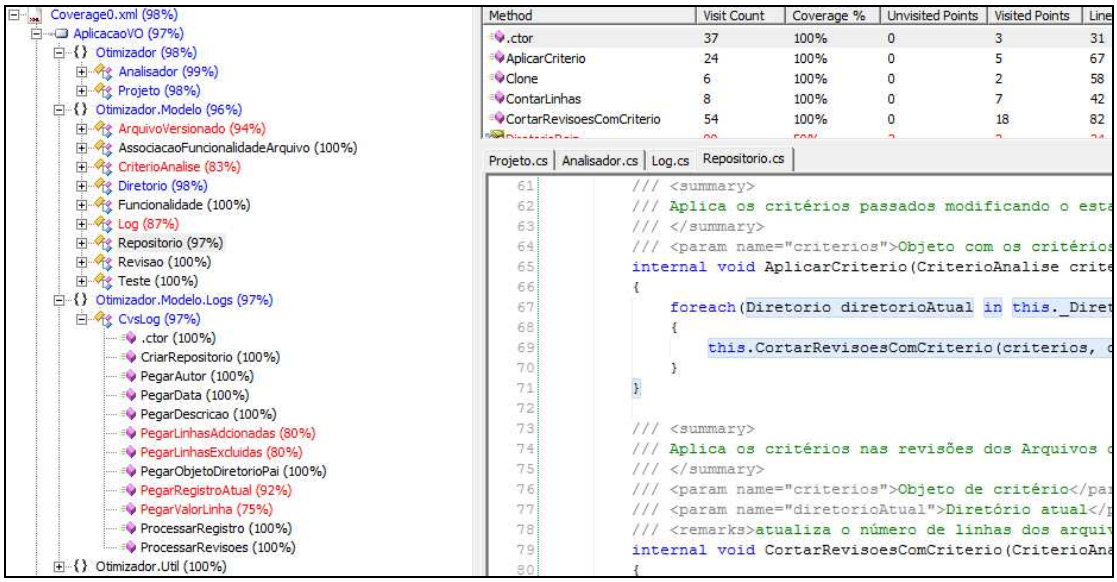

Figura 4. Tela da ferramenta *NCover*.

2.5.2 Priorização baseada em Requisitos

SRIKANTH et. al. (2005) apresentam uma técnica de priorização baseada em requisitos. Os casos de teste são cruzados com os requisitos para os quais foram desenvolvidos. A partir de propriedades dos requisitos (complexidade, prioridade junto ao cliente) os casos de teste são priorizados. Essa técnica possui uma fraqueza na avaliação das propriedades dos requisitos, que por vezes são subjetivas e de difícil avaliação.

2.5.3 Priorização baseada em Modelos

Na técnica baseada em modelos, os casos de teste são separados em dois grupos: alta prioridade e baixa prioridade. Essencialmente, casos de teste que tem ligação

[1] http://www.ncover.com/

com as últimas modificações do sistema sob testes são classificados como alta prioridade. O restante é classificado como baixa prioridade. A partir dessa classificação, uma ordenação aleatória é feita separadamente e depois os dois grupos são unidos, propondo então a ordem final de execução dos casos de teste. KOREL et. al. (2008) desenvolveram uma heurística mais sofisticada baseada em análise da dependência entre modelos aplicando uma construção gulosa ao processo de ordenação original e avaliando a taxa de detecção de falhas das suítes geradas.

2.5.4 Priorização baseada em Custo

WALCOTT et. al. (2006) propõem uma técnica de priorização com um limite de custo para sua execução. Nessa abordagem, o tempo é utilizado como limite para execução da priorização. A principal idéia por trás do conceito de limitar a execução é o de selecionar os melhores casos de testes já descobertos pela priorização e executar somente esses.

KRISHNAMOORTHI et. al. (2009) propõem um algoritmo genético que constrói a priorização pela cobertura do código e utiliza uma restrição de tempo máximo de execução para decidir a suíte que executa mais rápido e com maior cobertura do código, confirme a Figura 5.

A restrição de tempo é o principal critério de seleção de testes para compor a suíte final. A cobertura dos grupos de testes é a forma de avaliar o quanto um grupo é superior ao outro. Essa técnica, porém não leva em consideração as últimas alterações realizadas no sistema sob testes.

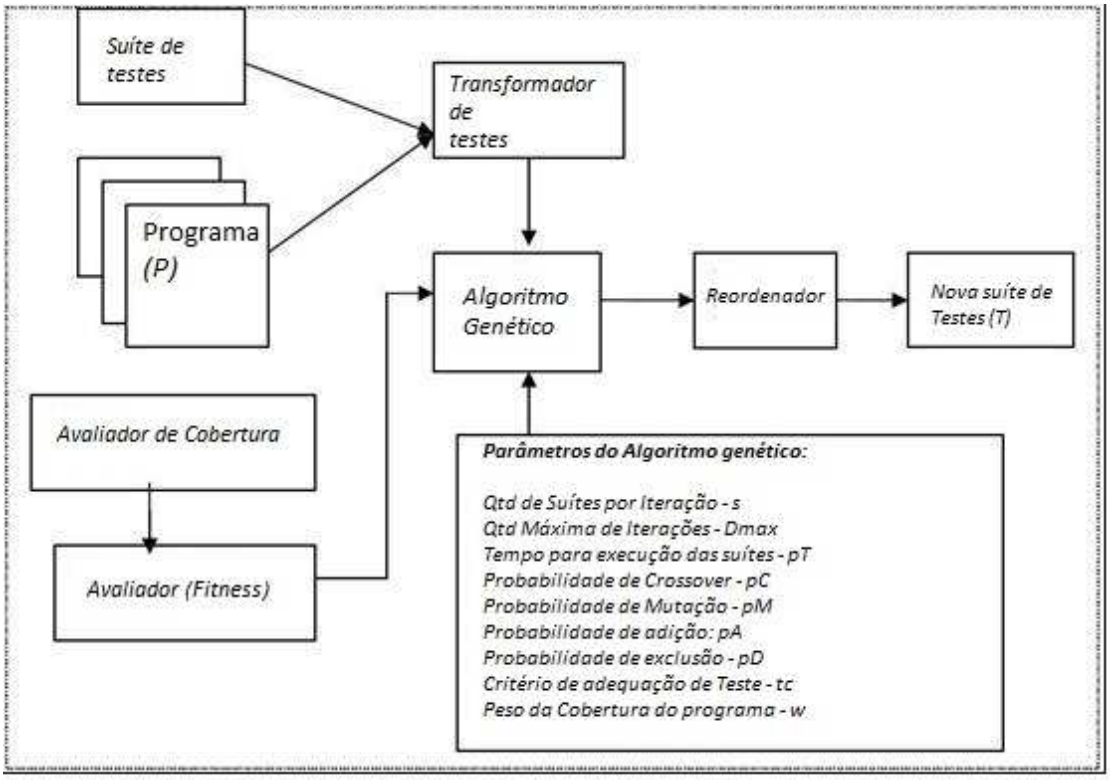

Figura 5. Algoritmo genético com restrição de tempo (KRISHNAMOORTHI et al., 2009)

2.6 Conclusão

Conforme abordado nesse capítulo, existem três tipos de técnicas principais para preparação e execução dos testes de regressão. Na técnica de minimização o objetivo é diminuir a suíte de testes. Na técnica de seleção o objetivo é encontrar um subconjunto dos testes que possuem relação com as últimas modificações. Por último, a técnica de priorização visa encontrar a ordem ideal de execução dos testes para toda a suíte. Em todas as abordagens foi constatado o uso de algoritmos baseados em heurísticas na tentativa de encontrar uma boa solução para o respectivo problema.

No próximo capítulo, essas heurísticas serão apresentadas e aplicadas em pesquisas relacionadas com testes de software, sendo discutidas com o objetivo de entender seu uso em cada técnica proposta e analisar os resultados obtidos a partir de sua aplicação.

CAPÍTULO 3

MÉTODOS HEURÍSTICOS APLICADOS A TESTE DE SOFTWARE

3.1. Introdução

Heurísticas são técnicas que têm por finalidade a busca de soluções boas, próximas da solução ótima, para problemas combinatórios (REEVES, 1995). Esse tipo de técnica visa baixar o custo computacional de criar soluções de qualidade, ainda que nem sempre seja encontrada a solução ótima. Um ponto fraco desse tipo de abordagem é que não existem garantias da qualidade da solução gerada, nem mesmo sobre quão próxima ela está da solução ótima. Porém, em problemas cujo espaço de busca cresce rapidamente para instâncias grandes e onde não se pode presumir a distribuição das soluções no espaço de busca, os métodos heurísticos tornam-se muito utilizados para encontrar soluções próximas ao ótimo.

O problema da seleção dos testes críticos é um problema complexo de otimização combinatória. Problemas dessa natureza consistem na procura de grupos, ordenações ou atribuições de um conjunto discreto e finito de objetos, naturalmente representados por como variáveis de decisão, que atendam um conjunto de restrições impostas, com a finalidade de minimizar ou maximizar o valor de uma determinada função objetivo (HOLGER e STÜZLE, 2005). Contudo, existem muitas variações quanto aos critérios utilizados para definir uma suíte de teste ideal. Vários trabalhos similares propõem uma solução específica, de acordo com seu contexto e métodos, para encontrar soluções para o problema. Esse capítulo apresentará alguns desses trabalhos.

Este capítulo apresenta trabalhos de pesquisa que tratam do problema de priorização e seleção de casos de testes e está dividido em sete seções. A Seção 3.2 apresenta trabalhos que aplicam algoritmos gulosos na seleção de casos de teste. Na seção 3.3 é apresentado o trabalho de (ZHENG e HARMAN, 2007), onde um algoritmo de Hill Climbing é aplicado para selecionar casos de teste que avaliam o código afetado por uma alteração. Na seção 3.4 é apresentado o trabalho de

(KRISHNAMOORTHI et. al. 2009), que utiliza a heurística genética na busca por um grupo de casos de teste de cobertura máxima impondo uma restrição de tempo. A Seção 3.5 apresenta o trabalho LOIOLA et al. (2009), onde foi implementado um algoritmo baseado no GRASP Reativo para priorização de casos de teste. A Seção 3.6 apresenta o trabalho de (BAUDRY et al. 2005), que desenvolveram uma técnica de otimização de casos de teste utilizando a heurística da adaptação bacteriológica com o objetivo de encontrar um grupo ótimo que satisfaz algumas restrições. A Seção 3.7 faz uma análise consolidada das técnicas apresentadas neste capítulo e apresenta suas conclusões.

3.2 Seleção de Casos de Teste com Algoritmos Gulosos

Um algoritmo guloso (REEVES, 1995) é uma implementação da filosofia de busca pela "melhor opção". Ele trabalha o princípio de que o elemento com o valor máximo é a melhor opção, seguido do elemento com o segundo maior valor e assim por diante, até uma solução completa. Para que um algoritmo guloso encontre uma solução, o problema tem que permitir que a solução seja dividida em candidatos que quando somados representam uma solução ótima (REEVES, 1995). Assim, a cada iteração o algoritmo vai escolhendo o que lhe parece ser o melhor candidato e vai compondo a solução. Um algoritmo guloso é um método computacionalmente simples e, em situações em que é possível aplicá-lo, é um método atraente porque é barato tanto no tempo de execução quanto de implementação.

SRIVASTAVA e THIAGARAJAN (2002) combinaram a abordagem baseada na priorização gulosa com a seleção de casos de teste. Na técnica resultante, eles inicialmente identificam os blocos de código modificados na nova versão do sistema sob testes, comparando o seu código binário com a versão anterior. Uma vez que os blocos modificados são identificados, a priorização de casos de teste é realizada usando um algoritmo guloso que considera somente a cobertura dos blocos modificados, descartando casos de teste que cobrem outras partes do software.

A técnica gulosa proposta é dividida em três etapas. Na primeira etapa é utilizada uma ferramenta de BMAT (*Binary Matching Tool*) (HARROLD e ROTHERMEL, 2001) para casar os blocos binários da versão anterior do sistema com os blocos binários da nova versão. O objetivo é descobrir quais blocos binários sofreram alteração. Ao fim da primeira etapa, o conjunto de blocos com qualquer tipo de alteração é denominado como o conjunto de blocos alterados.

Na segunda etapa, a técnica determina que blocos alterados da nova versão são cobertos por pelo menos um caso de teste existente, conforme mostra a Figura 6. Após as duas etapas iniciais, a técnica determina o conjunto de blocos alterados (blocos modificados novos e antigos) que são cobertos pelo conjunto de testes.

```
While (qualquer t em ListaTestes cobrir qualquer bloco em Blocosimpactados)
{
        Blocosatuais = Blocosimpactados
        Inicie uma nova sequencia Seq
        While ( qualquer t em ListaTestes cobrir qualquer bloco em Blocosatuais)
        {
                para  cada Teste em ListaTestes faça
                {
                        peso(Teste)=contar[Blocosatuais ∩ Cobertura(Teste)]
                }
                Selecione teste t em ListaTestes com o peso máximo
                Adicionar t a sequencia atual seq
                Remover t da ListaTestes
                CurrBlkset = CurrBlkset – Coverage(p)
        }
}
Put all remaing tests in Testlist in a new Sequence seq
```

Figura 6. Algoritmo Guloso para priorização de casos de teste (HARROLD e ROTHERMEL, 2001)

Na terceira etapa, a técnica prioriza o conjunto de testes. Conforme mostrado na Figura 6, a técnica usa a cobertura do caso de teste em um determinado bloco para propor um peso para a relação entre o bloco e o caso de teste. Os casos de teste são priorizados de acordo com estes pesos, que são calculados como o total de linhas cobertas pelo teste dividido pelo total de linhas alteradas no bloco. A técnica utiliza um algoritmo iterativo guloso para encontrar uma pequena lista de casos de teste que cubram o máximo possível de cada bloco, de acordo com os pesos calculados.

O caso de teste com o peso máximo é o primeiro selecionado. Em caso de empate, os autores sugerem pegar o caso de teste com a maior cobertura global. O teste selecionado é removido da lista de casos de teste avaliados e os blocos cobertos por ele são removidos da lista de blocos alterados (Blocos atuais). O peso de cada caso de teste é recalculado com base no conjunto de blocos alterados

atualizado. Ao fazer isso, o algoritmo tenta escolher um dos testes restantes que cobrirá o maior número de blocos restantes e afetados pelas alterações.

Este processo é repetido enquanto existir pelo menos um caso de teste que possa cobrir quaisquer blocos alterados. Os casos de teste selecionados formam a primeira seqüência, que irá fornecer o máximo de cobertura dos blocos alterados. Este processo é repetido para gerar a próxima seqüência, enquanto existir ao menos um teste que cubra qualquer um dos blocos alterados (Blocos impactados). Testes restantes são adicionados a uma seqüência separada em ordem da sua cobertura global. Como mantemos uma lista de testes ordenada pelo peso, encerramos a busca por um teste quando peso calculado for maior do que o peso original do próximo teste.

ZHENG e HARMAN (2007) produziram um estudo empírico no qual fizeram uma comparação entre duas variações do algoritmo guloso para construção de uma solução: o Algoritmo Guloso Adicional e o Algoritmo Guloso Ótimo.

O Algoritmo Guloso Adicional é uma variante de algoritmo guloso que combina o resultado das escolhas anteriores antes de decidir qual será a próxima escolha. A cada iteração, o algoritmo seleciona o elemento de peso máximo que não tenha sido selecionado antes e recalcula os pesos dos elementos remanescentes. Essencialmente, a diferença entre a abordagem gulosa padrão e a gulosa adicional é que o segundo utiliza dados das seleções anteriores e as combina de forma que selecione a próxima parte que melhor completar o problema, de acordo com o que foi selecionado previamente.

O algoritmo Guloso Ótimo é uma implementação do algoritmo K-Ótimo (S LIN, 1965). A abordagem K-Ótima seleciona os próximos K elementos que juntos resolvem a maior parte do problema. No caso do K-Ótimo Guloso, é a maior parte remanescente do problema que será selecionada, ou seja, o algoritmo escolhe o conjunto de testes que juntos resolvem a parte do problema que ainda não tem solução. Portanto, ele também precisa atualizar os dados de cobertura para os casos de teste não selecionados após a escolha de cada K casos de teste.

O estudo realizado por (ZHENG e HARMAN, 2007) utilizou sistemas variando de 374 até 11.148 linhas de código. O objetivo do experimento era maximizar a cobertura dos testes sobre os sistemas. Para isso, três critérios de cobertura foram usados: cobertura de bloco, cobertura de ramificação e cobertura total. Com isso, dependendo do critério utilizado no experimento uma das seguintes métricas foi utilizada:

- APBC (*Average Percentage Block Coverage*). Essa métrica permite classificar os testes de acordo com a sua cobertura por blocos de instrução;
- APDC (*Average Percentage Decision Coverage*). Essa métrica classifica os testes de acordo com a quantidade de ramificações (caminhos de decisões diferentes) que eles cobrem;
- APSC (*Average Percentage Statement Coverage*). Essa métrica classifica os testes de acordo com a quantidade de funções ou classes que eles cobrem;

Levou-se em consideração também o tamanho das suítes de teste selecionadas, classificando-as em pequenas (de 8 até 155 casos de teste) e em grandes (de 228 até 4350 casos de teste).

Os resultados analisados indicaram que o algoritmo guloso padrão tem desempenho pior do que o algoritmo Guloso Adicional, pior do que o Guloso Ótimo e do que algoritmos genéticos em geral. Finalmente, os estudos indicam ainda que não há diferença significativa entre o desempenho do Guloso Ótimo e do Guloso Adicional em termos de eficácia.

3.3 Hill Climbing

Hill Climbing é um algoritmo de busca local guloso que possui duas variações principais: a subida mais íngreme e subida mais próxima. Essa técnica também é detalhada no estudo empírico realizado por (ZHENG e HARMAN, 2007). Nesse trabalho, a implementação escolhida foi a subida mais íngreme, seguindo os seguintes passos:

1. Escolher aleatoriamente uma solução e colocá-la como solução atual;
2. Avaliar toda a vizinhança do estado atual e escolher a solução vizinha com o melhor valor;
3. Mover o estado atual para o vizinho escolhido. Se não há vizinho com um valor melhor do que o do estado atual, então nenhum movimento é feito.
4. Repetir os passos dois e três até o que não exista mais movimento a ser feito;
5. Defina o estado atual como a solução.

Nesse estudo, uma solução é uma ordenação de um conjunto de testes e a vizinhança é definida como qualquer nova ordem de um conjunto de testes que possa ser obtido pela troca de posição do primeiro caso de teste por algum outro.

Com essa definição de vizinhança, qualquer conjunto de testes tem N-1 vizinhos que devem ser avaliados para cada iteração do algoritmo. Existem muitas opções de vizinhos válidos nessa abordagem. Não há nada de especial no primeiro caso de teste do grupo, trata-se apenas de uma escolha aleatória. Apesar da simplicidade e baixo custo computacional do Hill Climbing, geralmente não encontra o ótimo global, pois fica preso em eventuais ótimos locais.

HARMAN e MCMINN (2010) também utilizaram um algoritmo de Hill Climbing em seu estudo para geração de dados de teste de estrutura. Teste de estrutura consiste em testar um sistema com a maior cobertura possível de sua estrutura, ou seja, cobrir o código de maneira ampla e passando pelas parte mais abstratas do que as concretas. Nesse estudo um Algoritmo Genético (detalhado na seção 3.3) foi comparado com um Hill Climbing no estudo de cobertura de alguns sistemas open source em linguagem C. Os resultados mostraram que, embora o algoritmo genético encontre soluções melhores que o Hill Climbing, a distância entre essas soluções não é grande. Isso demonstra que, para esse problema em particular, é provável que o custo da execução do algoritmo genético não justifique o seu uso (HARMAN e MCMINN, 2010).

3.4 Algoritmos Genéticos

Algoritmos Genéticos (GA) são heurísticas para problemas de otimização combinatória inspirados nos mecanismos da evolução de populações de seres vivos. Segundo (GOLDBERG, 1989), algoritmos genéticos utilizam regras de transição probabilísticas e não-determinísticas, o que lhes permite analisar simultaneamente uma grande área do espaço de busca e trabalhar com um grande número de parâmetros complexos. Essas características permitem que eles escapem dos mínimos locais, tornando-os excelentes estratégias heurísticas para obter soluções boas para problemas de otimização combinatória.

Krishnamoorthi e Sahaaya (2009) propõem um método de priorização de casos de teste impondo uma restrição de tempo para execução dos mesmos. O objetivo é encontrar a ordem de execução dos testes que maximiza a descoberta de erros em uma situação onde os testes podem ser interrompidos no meio da execução ou planejados para executar dentro de um limite de tempo. Para implementação do método foi desenvolvido um procedimento por meio da adaptação de um GA.

O percentual de cobertura é usado como parte do critério para decisão do conjunto de casos de teste e é dividido em duas partes: cobertura de método e

bloco. Na cobertura de método, é quantificado o número de métodos cobertos por determinado teste, não interessando se a cobertura sobre o código do método é de 100%. Já quando o critério é também por bloco, a unidade passa a ser a linha de código. A partir daí, cada instrução conta para determinar o percentual de cobertura. O modelo leva ainda em consideração outro critério: o tempo máximo de execução da suíte escolhida. Esse valor é escolhido e fornecido ao algoritmo como uma constante.

O algoritmo monta a primeira geração de forma aleatória. Exemplo: TS_m = {T1,T2,T3} e TS_n = {T3,T4,T5,T6}. A avaliação dos indivíduos se dá em duas partes. Na primeira parte, um valor de cobertura é definido multiplicando-se o percentual de cobertura por um peso. Já na segunda parte temos a avaliação do tempo em que um caso de teste executa, mas é preciso maximizar o valor para casos de teste com maior cobertura e menor tempo. Portanto há uma razão entre o tempo atual da suíte e seu tempo máximo. Para finalizar a avaliação de um indivíduo, basta somar as partes.

Se uma suíte viola a restrição de tempo então o seu valor para a função de fitness é atribuído em -1. Assim, essa suíte não estará presente na próxima geração. Após a avaliação, as operações de cruzamento, mutação, adição e exclusão são efetuadas. Para decidir se uma operação será efetuada, o algoritmo gera um número aleatório entre zero e um. Se o número gerado for menor que o valor da probabilidade de acontecimento da operação, definido no início do algoritmo, então a operação é executada. As operações de cruzamento e mutação são as comumente vistas em algoritmos genéticos. Já as operações de adição e exclusão permitem que o algoritmo acelerar o processo de descoberta de novas suítes, adicionando ou excluindo elementos de um indivíduo, também de forma aleatória.

3.5 Reactive Grasp

GRASP (*Greedy Randomized Adaptative Search Procedures*) é uma heurística baseada na busca gulosa, mas que utiliza uma abordagem aleatória. Possui duas fases distintas – construção e busca local (RESENDE e RIBEIRO, 2001) – e é definida como um algoritmo de multi-partida, que seleciona aleatoriamente N pontos de partida distintos e executa o procedimento de busca várias vezes a fim de obter a solução ótima global.

LOIOLA et al. (2009) propõem uma técnica de priorização de casos de teste baseada em GRASP. A técnica é dividida em 3 partes: construção, busca local e

atualização da melhor solução. Na fase de construção, uma primeira solução viável é construída por meio da aplicação de um algoritmo guloso. Como esse algoritmo tende a encontrar um ótimo local, a técnica usa uma estratégia gulosa com aleatoriedade. O algoritmo reinicia aleatoriamente após parar em um ótimo local.

Na fase da busca local, o objetivo é encontrar a melhor solução na vizinhança da solução atual, substituindo a solução atual pela melhor solução encontrada na sua vizinhança. Na última parte do processo, a solução ótima local é comparada com a melhor solução encontrada nas iterações anteriores. Se o ótimo local encontrado é melhor, então ele é definido para ser a melhor solução já encontrada. Caso contrário, não há atualização. O algoritmo está apresentado na Figura 7. O GRASP Reativo (LOIOLA et al. 2009) se apresentou significativamente melhor em termos de desempenho e cobertura quando comparado a Algoritmos Genéticos, Simulated Annealing, Guloso Simples e Adicional.

```
Inicializa probabilidades associadas a α (todas iguais a 1/n) em αSet
De k = 1 até totalIterações faça
        α ← Selecione α (αSet);
        solução ← construção(α);
        solução ← buscaLocal(solução);
        atualizeSolução(solução, MelhorSolução);
fim;
return best solution;
```

Figura 7. Algoritmo GRASP para priorização de casos de teste (LOIOLA et al. 2009)

3.6 Adaptação Bacteriológica

Mutação de testes é uma técnica que foi inicialmente concebida para criar dados de teste eficazes, introduzindo defeitos relevantes no sistema sob testes (MILLER et. al. 1992). Foi proposto no estudo de (DEMILLO et. al. 1978) e consiste na criação de um conjunto de versões defeituosas ou mutantes de um programa com o objetivo final de concepção de casos de teste que distinguem o programa original de todos os seus mutantes.

Na prática, as falhas são modeladas por um conjunto de operadores de mutação, onde cada operador representa uma classe de defeitos de software. Um mutante é criado pela inserção de um defeito no programa original. Quando se gera mutantes, de um conjunto de operadores de mutação, podem-se criar mutantes equivalentes.

BAUDRY et al. (2005) propõem um algoritmo baseado no paradigma da adaptação bacteriológica, que é inspirado na ecologia evolutiva. Ecologia Evolutiva é o estudo de organismos vivos no contexto de seu ambiente com o objetivo de descobrir como eles se adaptam (PIANKA, 1999). Seu fundamento é que, em um ambiente heterogêneo, não é possível encontrar um indivíduo que resolve sozinho todos os problemas do ambiente. Então, uma alteração na população como um todo é necessária. Isto significa que não é possível gerar um único caso de teste perfeito para matar todos os mutantes. Em vez disso, aperfeiçoa-se um conjunto global de casos de teste para que juntos matem o máximo de mutantes possível.

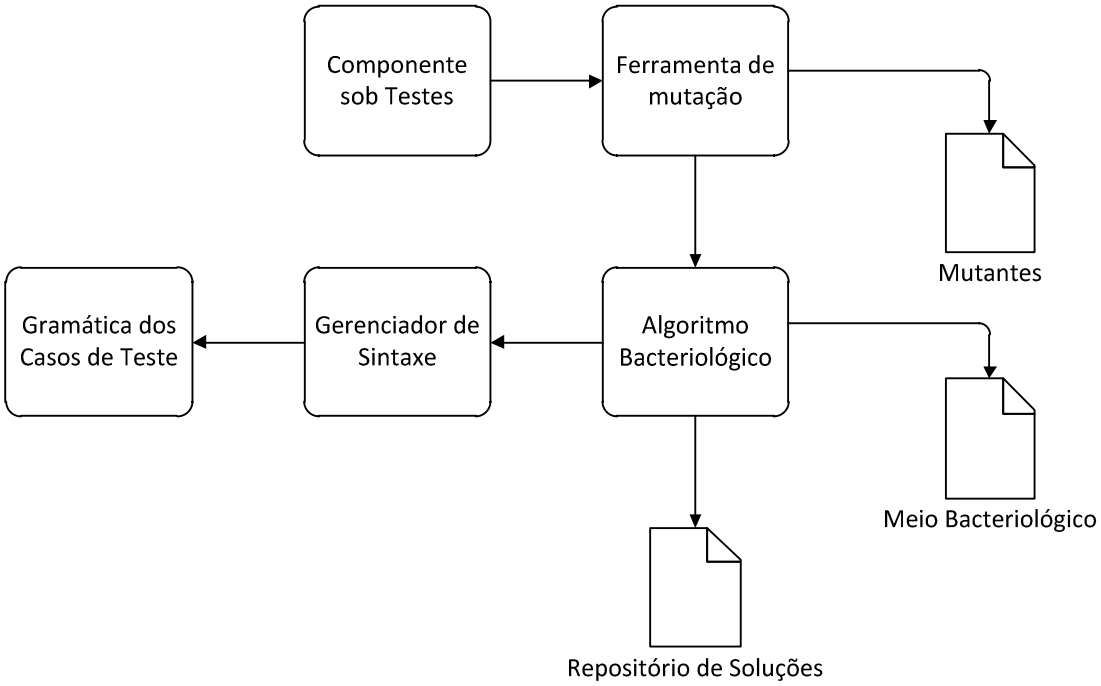

Figura 8. Algoritmo Baseado na Adaptação Bacteriológica (BAUDRY et al., 2005)

O algoritmo bacteriológico recebe como entrada um conjunto inicial de casos de teste e gera como saída um bom conjunto de casos de teste. O algoritmo evolui gradativamente (cada incremento é chamado de uma geração) e consiste de uma série de mutações em casos de teste para explorar o espaço de soluções. O algoritmo compõe o conjunto final de forma incremental, memorizando os casos de teste que podem melhorar a qualidade do conjunto (uma função de aptidão avalia a qualidade).

Conforme a execução do algoritmo, há dois conjuntos de testes: o conjunto de soluções que o algoritmo está construindo em meio bacteriológico (***Bacteriologic medium***) e um conjunto de casos de teste potencialmente interessantes (***Solution set***). Vários critérios de parada podem existir para o processo global: depois de certo número de gerações, quando o conjunto solução atinge um valor mínimo de aptidão,

se o valor do conjunto atual não mudou para um determinado número de gerações, e assim por diante. A Figura 8 mostra a arquitetura do algoritmo e como os elementos se relacionam durante a execução. O experimento realizado comparou resultados usando um Algoritmo Genético para eliminar mutantes. Na comparação o Algoritmo de Adaptação Bacteriológica converge mais rapidamente para o conjunto de testes ótimo, além de produzir menos mutantes repetidos. Os autores concluem dizendo que um estudo mais aprofundado precisa ser realizado.

3.7 Conclusão

Conforme abordado nesse capítulo, várias soluções heurísticas têm sido aplicadas para resolver o problema dos testes de regressão. A heurística gulosa apresentada escolhe o próximo melhor item disponível com a esperança que ao somar os melhores locais chegaremos à solução ótima. Hoje essa heurística é muito utilizada na inicialização de vários outros métodos, como no exemplo do GRASP Reativo apresentado na seção 3.5. Outras técnicas heurísticas como o Firewall não usam uma abordagem gulosa nem aleatória, mas conseguem atingir num bom tempo bons resultados.

A heurística genética tem sido aplicada em várias áreas como o principal recurso para chegar a uma solução boa. Na seção 3.4 a temos um exemplo de utilização da heurística genética, onde uma restrição de tempo de execução é imposta com o objetivo distinto de maximizar a cobertura dos casos de teste. Porém, não há dados comparativos com outras técnicas e tão pouco existe a relação com as últimas alterações realizadas no sistema sob testes.

Na heurística bacteriológica é apresentado um novo método, baseado na heurística genética, onde ao invés de procurar um conjunto de indivíduos tentamos modificá-los e torná-los mais “aptos” ao meio onde estão inseridos (problema). Dessa heurística vem um advento interessante sobre a parte de memorização dos indivíduos mais aptos. Essa memorização, no caso do problema de seleção de casos de testes, torna a convergência do algoritmo algo muito mais rápido que na comparação com algoritmos genéticos.

No capítulo 4 será apresentado o modelo formal do problema de seleção de casos de teste críticos, assim como a proposta de solução.

CAPÍTULO 4

TÉCNICA DE SELEÇÃO DE CASOS DE TESTE CRÍTICOS

4.1. Introdução

Devido ao seu alto custo, frequentemente menos esforço é dedicado às atividades de teste do que seria necessário para validar o software completamente. Em casos extremos, os testes são quase que praticamente eliminados do processo de desenvolvimento de uma ou mais versões do sistema. Mesmo empresas que têm um processo institucionalizado podem atribuir ao teste um papel secundário no contexto de algumas liberações de versão de um software, devido à pressão para a entrega de uma versão operacional do sistema.

Este comportamento é ainda mais comum quando são encontrados erros em ambiente de produção, erros estes que podem bloquear a correta execução de operações do negócio suportado pelo sistema. Em tais situações, as atividades de manutenção corretiva são frequentemente executadas no próprio ambiente de produção, lançando a versão corrigida do software em produção sem testes apropriados. Dada a necessidade de resolver a falha, a equipe realiza a mudança e implanta imediatamente. Por conta disso, o risco de inserir mais defeitos no sistema e causar danos colaterais para a operação é ainda pior do que não testar uma mudança de emergência.

Para minimizar o risco de liberar uma versão incorreta ou de incluir mais defeitos, algum tipo de verificação deve ser aplicado sobre o software, mesmo em uma situação crítica de atualização. Por exemplo, pode-se testar todo o código referente às funcionalidades afetadas pelo código escrito ou alterado como parte da liberação de emergência. No entanto, separar manualmente os casos de teste para um conjunto específico de mudanças também pode ser uma tarefa demorada. Uma vez que estamos tratando de uma alteração emergencial, o tempo consumido é um critério decisivo. Outra desvantagem é que considerar apenas o código novo ou alterado pode não garantir que outras partes essenciais do sistema continuem consistentes após a alteração.

Assim, é preciso selecionar um conjunto de casos de teste que possa ser executado em um determinado período de tempo e que cubra as funcionalidades mais importantes que foram afetadas pela mudança de emergência, de acordo com a prioridade de cada funcionalidade. Além disso, podemos permitir maior controle sobre os tipos de teste (positivos ou negativos) que comporão o conjunto final dos testes. Com essa definição, podemos formalizar um problema de otimização combinatória, que é o tema desse trabalho de pesquisa. Conforme visto nos capítulos 2 e 3, testes é uma das principais áreas de pesquisa da SBSE. Neste capítulo serão apresentados a formalização do modelo do problema, um procedimento de coleta de dados e a descrição da adaptação de um algoritmo genético para o problema.

Este capítulo está organizado em seis seções. A primeira compreende esta introdução. A segunda parte detalha o modelo proposto para o problema, listando os componentes do mesmo e como se relacionam. A terceira parte detalha o processo de coleta dos dados necessários para a otimização. A quarta parte propõe um modelo formal para o problema de seleção dos casos de teste críticos. A quinta detalha uma proposta de solução para o problema. Por fim, a sexta parte contém a conclusão e considerações finais do capítulo.

4.2 Modelo do Problema

Dada uma alteração de emergência, estabelecemos o problema de interesse como a seleção de um conjunto de casos de teste que possa ser executado em um prazo curto e previamente definido e que maximize a cobertura das funcionalidades que sofreram alterações neste contexto. Assumimos que as funcionalidades afetadas pelas alterações podem ter prioridades distintas, considerando aqui uma perspectiva dos patrocinadores do negócio suportado pelo sistema. Também restringimos o problema aos testes de unidade, que são testes desenvolvidos para validar um módulo de código específico (BARTIÉ, 2002). Os módulos de código-fonte, por outro lado, implementam uma ou mais funcionalidades. A Figura 9 apresenta as relações entre esses elementos.

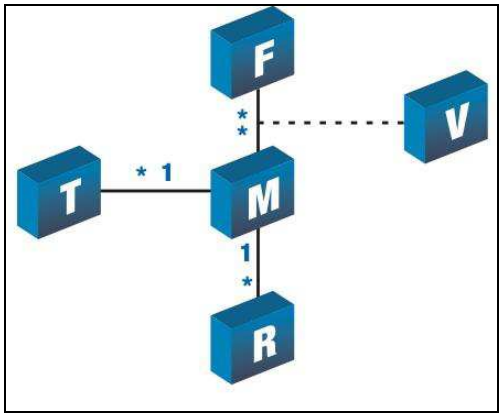

Figura 9. Modelo do problema

O problema de seleção de casos de teste críticos está relacionado ao problema NP-completo que determina os itens a incluir em uma coleção (KHURI et. al. 1994), onde cada item é descrito por um peso e um valor, de modo que o peso total do conjunto seja inferior a um determinado limite e seu valor total seja maximizado. Neste trabalho, um item é mapeado para um caso de teste. O peso de um item está relacionado ao tempo de execução do caso de teste. O limite de peso é a restrição de tempo máximo para execução do conjunto de casos de teste. Finalmente, o valor a ser maximizado é uma composição das prioridades das funcionalidades afetadas pela mudança emergencial com a cobertura dos casos de teste aplicáveis sobre os módulos de código-fonte que implementam estas funcionalidades.

Uma funcionalidade (F) representa um requisito funcional fornecido pelo software. Em geral, cada transação ou operação específica realizada pelo software é considerada uma funcionalidade. Em nosso contexto, uma funcionalidade é implementada por um conjunto de módulos de código-fonte (artefatos de análise e *design* não são tratados no método de seleção de casos de teste). Cada funcionalidade possui uma propriedade que indica a sua prioridade para a operação do sistema, ou seja, sua prioridade do ponto de vista dos patrocinadores. Quanto mais importante a funcionalidade no sentido de suportar o negócio, maior será o valor de sua prioridade. Recursos com a prioridade máxima serão sempre marcados como alterados, ou seja, receberão sempre atenção no processo de seleção dos casos de teste.

Um módulo (M) é um arquivo que contém parte do código-fonte que compõe o sistema. Cada módulo participa na implementação de uma ou mais funcionalidades. Assim, existe uma relação (V) entre as funcionalidades e os módulos de código-fonte, de modo que cada módulo implementa um percentual da funcionalidade. Para

a execução do método de seleção de casos de teste proposto, assumimos que os desenvolvedores são capazes de manter rastreabilidade entre os módulos de código-fonte e as funcionalidades providas pelo sistema. Essa relação normalmente é registrada por meio de uma ferramenta de controle de atividades, ou ferramenta de *issue tracking*. O Bugzilla[2] e o Trac[3] são exemplos de ferramentas que controlam as atividades dos desenvolvedores e associam essas atividades com um ou mais módulos. Isso é possível, no exemplo do Bugzilla ou Trac, realizando uma configuração no repositório do SVN[4] que muda o comportamento do mesmo. Quando um desenvolvedor precisa alterar um artefato que compõe o sistema o mesmo precisará de um *ticket* ou atividade aberta no controle de atividades, pois o sistema de controle de versões exigirá o identificador deste *ticket* como parte dos comentários associados à modificação no artefato. Caso contrário, o desenvolvedor não terá acesso ao artefato. Em outras palavras, qualquer alteração em artefatos do sistema tem que obrigatoriamente estar associada a um *ticket* ou atividade, garantindo a rastreabilidade dos artefatos.

Cada módulo tem um conjunto de revisões (R) e um conjunto de testes (T) associados a ele. Cada revisão associada a um módulo representa uma mudança sofrida por ele. Cada teste associado a um módulo cobre determinada parte do seu código-fonte. Uma vez que as funcionalidades são implementadas por módulos e que os módulos são verificados por casos de teste, os casos de teste cobrem indiretamente as funcionalidades providas pelo software.

Conforme apresentando no capítulo 2, casos de teste podem ser classificados como positivos ou negativos de acordo com a sua implementação e estratégia de procura por erros. Em nossa abordagem para selecionar casos de teste críticos, a proposta é permitir que o gestor atribua um peso para os testes positivos (Wp) entre 0% e 100%. O peso permite que o gestor decida se mais testes positivos ou negativos devem ser selecionados. Tal decisão pode depender dos tipos de mudanças sofridas pelo sistema durante a atualização de emergência. Ao selecionar o conjunto ideal de casos de teste, o método proposto leva em conta essas ponderações.

4.3 Procedimento de Coleta de Dados

Para conseguir as informações requeridas pelo método de seleção de casos de teste críticos, propomos usar dados sobre as últimas alterações sofridas pelos

[2] http://www.bugzilla.org/
[3] http://trac.edgewall.org/
[4] http://subversion.tigris.org/

módulos de código-fonte que compõem o sistema. Essa informação determina o conjunto de módulos que precisam ser testados. Se os casos de teste disponíveis relacionados a estes módulos necessitam de mais tempo para ser executados do que a restrição de tempo desejada, um procedimento de otimização é acionado para selecionar um subconjunto destes casos de teste de acordo com a sua cobertura e a prioridade das funcionalidades implementadas por módulos de código-fonte que eles validam.

Um sistema de controle de versão VCS (do inglês, version control system), ou ainda SCM (do inglês, source code management), no contexto da Ciência da Computação e da Engenharia de Software, é um software com a finalidade de gerenciar diferentes versões no desenvolvimento de um documento (MOLINARI, 2007). Subversion (ou SVN) é um dos muitos sistemas que são utilizados em projetos de software para gerenciamento de versão, sendo muito difundido na comunidade de código aberto por causa de sua distribuição sob uma licença open source. Projetos como o Apache e Debian usam o Subversion para apoiar o seu processo de gerenciamento de configuração. Para a coleta de informações sobre os módulos de código-fonte que foram alterados como parte da última liberação, capturamos informações do histórico de alterações mantido pelo Subversion. Ao ler o histórico de alterações, podemos identificar os módulos que compõem o código-fonte da aplicação e sua estrutura no sistema de arquivos (diretórios em que eles residem). Para cada módulo, o histórico de alterações possui as datas em que mudanças foram feitas neste módulo, o desenvolvedor que alterou o código e um comentário que descreve a mudança.

Enquanto os módulos de código-fonte que compõem o sistema podem ser identificados pelo histórico de alterações, as funcionalidades fornecidas pelo sistema podem ser diretamente obtidas a partir de uma ferramenta de registros de requisitos utilizada na indústria (como o Rational Clear Quest[5]) ou projetos open source. Portanto, se essa ferramenta estiver disponível, as funcionalidades podem ser recuperadas a partir dela.

Posto que a abordagem proposta se restringe a testes de unidade, cada caso de teste está ligado a um e apenas um módulo de código-fonte. Ferramentas de gerenciamento de teste permitem aos desenvolvedores manter um repositório de casos de teste, por vezes relacionando-os com os módulos de código-fonte que são validados por eles. Mesmo que essas ferramentas não estejam disponíveis, os

[5] http://www-01.ibm.com/software/awdtools/clearquest/

casos de teste de unidade geralmente seguem regras rígidas de desenvolvimento, que incluem convenções de nomenclatura e restrições sobre os diretórios em que eles precisam ser salvos. Assim, o conjunto de casos de teste desenvolvido para um sistema pode ser facilmente recuperado e associado aos módulos que os casos de teste avaliam.

Finalmente, precisamos obter dados sobre a cobertura e o tempo de execução de cada caso de teste. A cobertura é uma métrica de completude de uma suíte de testes (MYERS, 1997), geralmente representada como um percentual dos artefatos relevantes que são cobertos pelos testes componentes da suíte. Este dado é lido dos resultados gerados por ferramentas de execução de testes que também são comumente utilizada na indústria. Ferramentas como o JCover (para Java) e NCover (para o .NET Framework) executam um conjunto de testes de unidade e coletam dados sobre o seu tempo de execução e cobertura. Essa informação fica disponível em arquivos que podem ser consumidos pelo processo de otimização.

4.4 Modelo Formal do Problema de Otimização

O problema de seleção de casos de teste críticos consiste em maximizar o alcance e a diversidade (testes positivos e negativos) da atividade de teste, respeitando uma restrição rigorosa sobre o tempo necessário para executar um conjunto de casos de teste e a prioridade das funcionalidades alteradas na última liberação de versão. Formalmente, temos:

- Seja F o conjunto de funcionalidades. Cada elemento $F_i \in F$ é descrito por um nome e uma prioridade;

  $F = \{F_i\}$

  F_i = (nome, prioridade)

- Seja Q = {P1, P2, P3, P4, P5} o conjunto de prioridades de funcionalidades que permite que cada elemento $q_i \in Q$ seja associado a um único peso inteiro positivo w_i onde $w_1 > w_2 > w_3 > w_4 > w_5$;

- Seja M o conjunto de módulos de código-fonte. Cada elemento $M_i \in M$ é descrito por um nome e um conjunto de revisões;

  $M = \{M_i\}$

  M_i = (nome, R)

- Seja R o conjunto de revisões associadas a um módulo de código-fonte M_i. Cada $R_i \in R$ é descrito por um identificador, uma data, uma descrição e um autor responsável por aquela revisão;

  $R = \{R_i\}$

  R_i = (número, data, descrição, autor)

- Seja T o conjunto de casos de teste. Cada caso de teste $T_i \in T$ é descrito por um nome, um tipo (positivo ou negativo), um tempo de execução, um percentual de cobertura sobre o módulo associado ao caso de teste e o próprio módulo;

  $T = \{T_i\}$

  T_i = (nome, tipo, tempo, cobertura, módulo)

- Seja V o conjunto de associações entre módulos de código-fonte e as funcionalidades. Um elemento $V_i \in V$ é uma associação entre duas instâncias destes conjuntos, sendo descrito por um módulo de código-fonte, uma funcionalidade e um percentual de propriedade. O percentual de propriedade define o quanto um módulo de código-fonte implementa uma funcionalidade;

  $V = \{V_i\}$

  V_i = (m, f, perc)

  onde $m \in M$, $f \in F$, $0 < perc \leq 100\%$,

- Seja S o conjunto de testes críticos, ou seja, o conjunto de casos de teste selecionados para apoiar o lançamento de uma versão de software sob restrições de tempo estritas e abrangendo os recursos afetados pelas mudanças mais recentes;

  $S = \{T_1, T_2, T_3, \ldots T_n\}$

O processo de otimização tem como objetivo selecionar um conjunto de testes suficientemente bom S* para que ele possa ser executado em um determinado intervalo de tempo (L) e maximize uma função de custo.

Maximize reward(S*) sujeito a time(S*) ≤ L

reward(s) = coverage(s) - penalty(s)

$$\text{cov}\,erage(s) = \sum_{t \in s} t.cobertura \cdot \left(\sum_{\substack{v \in V| \\ v.m = T.módulo}} v.perc * v.f.prioridade \right)$$

$$penalty(s) = \left| W_p - C_p \right| * 1000$$

As equações anteriores impõem uma série de restrições para a seleção de S*: (a) S* deve ser um subconjunto do conjunto de testes críticos S; (b) S* deve consumir menos do que L unidades de tempo para executar; e (c) nenhum outro subconjunto de S que possa ser executado até L unidades de tempo deve produzir um valor superior ao gerado por S* na função *reward*.

Para avaliar *reward* antes é necessário avaliar as funções *coverage*(s) e *penalty*(s). A função *coverage*(s) avalia um fator de cobertura global para um conjunto de casos de teste. Em seguida, a função *penalty*(s) ajusta este valor com uma função de penalidade. O fator de cobertura global para um determinado conjunto de casos de teste é a soma do fator de cobertura para cada caso de teste que compõe o conjunto. O fator de cobertura para um determinado caso de teste é calculado multiplicando-se a cobertura deste caso de teste pelo produto dos pesos das prioridades das funcionalidades cobertas pelo caso de teste, ponderados pela participação do módulo avaliado pelo caso de teste na implementação das funcionalidades.

Para avaliar a função de penalização, deve-se primeiramente determinar o equilíbrio desejado entre os casos de teste positivos e negativos no conjunto de testes a ser selecionado, pelo estabelecimento de valores de W_p (percentual desejado de testes positivos) e C_p (percentual observado de testes positivos). O módulo da diferença entre estes dois valores indica quão distante o percentual observado está do percentual desejado. Sendo assim, utilizamos este módulo como base da penalidade, multiplicando-o por um grande número (1000) para fazer com que o valor absoluto da penalidade seja significativamente maior do que o valor esperado para a função *coverage*. O valor escolhido como peso na penalidade foi obtido através de observação no experimento com as instâncias propostas, ou seja, vários valores foram experimentados e o mais coerente foi selecionado. Como estamos maximizando a função *reward*, descontamos a penalidade da função *coverage*, reduzindo seu valor quando a suíte for muito diferente do desejado.

4.5 Proposta de Solução Algorítmica

Conforme citado na seção 4.2, o problema de seleção dos casos de teste críticos pode ser analisado como um problema de otimização combinatória complexo. A complexidade e o número de combinações possíveis para uma instância do problema tornam o desenvolvimento de uma solução exata inviável, visto que o tempo disponível para encontrar a solução a ser aplicada é um recurso crítico do problema.

Nesse trabalho de pesquisa propomos a adaptação de um algoritmo genético, utilizando o framework de heurísticas JMetal (DURILLO et. al. 2010). O algoritmo genético proposto cria uma população inicial de cromossomos de forma aleatória. Nesta população, um cromossomo é definido por um conjunto de casos de teste e representa uma solução candidata, ou seja, uma solução que respeita as restrições impostas ao problema: nenhum caso de teste pode estar repetido dentro de uma solução, todos os casos de teste selecionados cobrem ao menos uma alteração e, em conjunto, respeitam o limite de tempo disponível sua execução.

Um cromossomo possui N genes, sendo N o número total de casos de teste que cobrem algum aspecto da alteração de emergência. Desta forma, cada gene representa um caso de teste e é representado por um valor zero ou um. O valor zero em um gene significa que o caso de teste correspondente não está incluído na solução que o cromossomo representa. O valor um determina que o caso de teste está incluído na solução representada pelo cromossomo. Para exemplificar a proposta, seja T o conjunto de casos de testes que cobrem uma determinada alteração. Considere que T tem 10 casos de teste. Sejam T1 e T2 duas possíveis soluções para o problema, ou seja, dois subconjuntos de T. As descrições C1 e C2 abaixo representam, respectivamente, os cromossomos de T1 e T2.

T = {T1, T2,T3,T4 T5, T6,T7,T8,T9,T10}

T1 = {T1, T3, T4, T5, T6, T8, T10}

C1: (1,0,1,1,1,1,0,1,0,1)

T2 = {T2, T4, T5, T6, T9, T10}

C2: (0,1,0,1,1,1,0,0,1,1)

A criação da população inicial é feita gerando cromossomos S aleatoriamente, onde um conjunto aleatório de genes é escolhido e passado para o cromossomo. O algoritmo avalia se o conjunto de genes gerado é uma solução viável ou seja, pode

ser executada dentro do tempo disponível. Em caso positivo, o indivíduo é criado e adicionado à população. Em caso negativo, o indivíduo é descartado. Esse procedimento continua até completar o número de indivíduos desejados na população.

Com a população inicial construída, calcula-se o valor de cada cromossomo S conforme definido na função *reward*. Em seguida, os cromossomos são ordenados em ordem decrescente de valor calculado para a função, sabendo-se que quanto maior o valor do cromossomo melhor é a solução. A partir da população inicial ordenada, é executado o processo de reprodução que define a geração de novos cromossomos ao longo das iterações.

A estratégia de seleção adotada foi baseada no conceito do Elitismo (KICINGER et al. 1978), onde um percentual dos melhores indivíduos (cromossomos) é escolhido para compor a nova geração. O método de Elitismo é um método de seleção que força o algoritmo a reter um certo número de melhores indivíduos a cada nova geração. Tais indivíduos poderiam ser perdidos se eles não fossem selecionados para reprodução ou se eles fossem alterados pela mutação.

Neste trabalho, a técnica de reprodução escolhida foi o *single-point crossover* . A escolha dessa técnica garante a diversidade da população e ajuda a garantir que nenhum ponto do espaço de busca tenha probabilidade zero de ser examinado (LIVRAMENTO, 2010).

Finalmente, após a reprodução aplicamos a operação de mutação. Nesta operação, os indivíduos e genes que sofrerão mutação são escolhidos aleatoriamente. Para cada gene do cromossomo escolhido um número aleatório entre [0, 1] é gerado. Se esse número for menor que a P_m (probabilidade de mutação) então o valor do gene é trocado pelo seu oposto. Em seguida, o algoritmo repete a avaliação para verificar se o novo indivíduo representa uma solução viável. Caso negativo, o mutante é descartado.

Portanto, o procedimento de formação dos cromossomos para composição de uma nova geração consiste nas três etapas abaixo (Figura 3):

1. Selecionar os E% melhores indivíduos (cromossomos de maior valor) da população corrente;
2. R% dos novos indivíduos serão gerados pelo processo de reprodução. Para cada reprodução, seleciona-se aleatoriamente um cromossomo pertencente aos E% melhores indivíduos (pai 1) e outro cromossomo (pai 2) do conjunto total de

indivíduos da população corrente. Para cada gene do cromossomo filho gera-se um número real aleatório X ∈ [0, 1]. Se X for menor ou igual ao fator C% (percentual de Crossover), então o gene do pai 1 é selecionado. Caso contrário, seleciona-se o gene do pai 2. Assim, o cromossomo filho será composto por uma seqüência de genes dos dois cromossomos pais. Da mesma forma que na criação da população inicial, os indivíduos filhos resultantes do processo de reprodução são avaliados para saber se respeitam o critério de tempo. Caso uma operação de reprodução gere um indivíduo inválido, o mesmo será descartado e o processo será repetido até gerar um novo indivíduo único e válido;

3. (1- E% - R%) dos novos indivíduos são aleatoriamente gerados (mutantes).

O processo descrito acima é executado até que um dos critérios de parada seja alcançado. Os critérios de parada adotados neste trabalho são a avaliação de um número G de gerações ou encontrar um indivíduo de custo ótimo (ou seja, que cubra o mais próximo de 100% do código alterado dentro do limite de tempo imposto com o percentual escolhido do tipo de casos de teste).

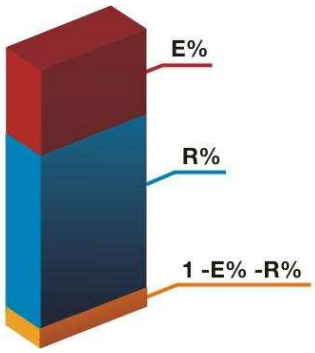

Figura 10. Ilustração de uma nova geração

4.6 Conclusão

Este capítulo apresentou o problema de seleção de casos de teste críticos, onde se deseja selecionar um conjunto de casos de testes que cubra as últimas alterações realizadas sobre um sistema de software. O conjunto deve respeitar a priorização dessas alterações e um limite de tempo pré-estabelecido, com o objetivo de permitir que sistemas complexos possam contar com um mínimo de testes quando a pressão para correção de falhas afeta o esforço necessário para conduzir a atividade de testes de forma adequada.

Nesse capítulo a proposta de solução apresentada consiste em um método de coleta dos dados necessários, um modelo formal e a adaptação de um algoritmo

genético para encontrar boas soluções para o problema. No capítulo 5 serão conduzidos experimentos com instâncias aleatórias do problema e com dados de projetos reais da indústria com o intuito de avaliar a eficácia da solução proposta.

CAPÍTULO 5

AVALIAÇÃO DO MÉTODO PROPOSTO

5.1. Introdução

Conforme detalhado no capítulo 4, o problema de seleção de casos de teste críticos possui alta complexidade e um número grande de possíveis soluções. Nesse capítulo apresentamos um estudo experimental baseado no procedimento de coleta de dados e no processo de otimização que foram descritos no capítulo 4 com o intuito de avaliar o método heurístico proposto para encontrar boas soluções para o problema. Levando em consideração as características do problema, dois tipos de instância foram definidos para uso no estudo experimental: instâncias criadas aleatoriamente e instâncias do mundo real.

Os detalhes do estudo experimental serão descritos neste capítulo, que está organizado em sete seções. A seção 5.1 compreende essa introdução. A seção 5.2 descreve o estudo experimental em linhas gerais e discute as questões de pesquisa que nortearam o desenvolvimento do mesmo. A seção 5.3 descreve as instâncias usadas no estudo experimental e suas nuances. A seção 5.4 lista os algoritmos utilizados, com uma breve descrição sobre os motivos que fizeram com que os incluíssemos no estudo experimental. Já a seção 5.5 mostra os resultados do estudo experimental e discute cada aspecto relacionado às questões que foram apresentadas na seção 5.2, dividindo essa seção entre instâncias aleatórias e instâncias reais. A seção 5.6 discute as ameaças a validade do estudo experimental, enfatizando as ações que foram tomadas para minimizá-las. Finalmente, a seção 5.7 traz as conclusões do capítulo.

5.2. Questões da Pesquisa

Um estudo experimental foi projetado e conduzido para avaliar a eficiência e a efetividade de três técnicas de busca propostas para o problema de seleção de casos de teste críticos. As três técnicas de busca avaliadas são Algoritmos Genéticos (Busca Heurística), Hill Climbing (Busca Local) e Busca Aleatória (busca

não sistemática). Daqui por diante, esses algoritmos serão chamados de GA, HC e RS, respectivamente.

No estudo experimental foi utilizado o jMetal (DURILLO, 2010), um framework desenvolvido em Java com o objetivo de oferecer suporte à implementação, execução e estudo e técnicas de busca heurística. O código-fonte desenvolvido como parte deste estudo experimental e as instâncias projetadas para avaliação das técnicas propostas estão disponíveis para download no endereço http://www.uniriotec.br/~marcio.barros/testcriticalchanges. Três questões de pesquisa foram exploradas no estudo experimental e são relatadas a seguir.

1. **RQ1 - Reward como critério de avaliação.** Qual algoritmo encontra o melhor valor para a função de *fitness*?

   Essa questão compara o GA, o HC e o RS baseando-se no resultado da função *Reward* apresentada no capítulo 4. *Reward* é utilizada como uma função de *fitness* na otimização mono-objetivo, guiando as técnicas de busca. É esperado que o GA encontre melhores valores de *Reward* que o HC e o RS.

2. **RQ2 - Custo como critério de avaliação.** Qual o custo computacional necessário para executar cada um dos algoritmos em termos do número de avaliações executadas da função de *fitness*?

   Essa questão compara as técnicas de acordo com a quantidade de avaliações necessárias para que cada uma encontre o seu melhor valor. É esperado que o GA encontre o seu melhor valor com menos avaliações do que o HC e o RS.

3. **RQ3 - Cobertura como critério de avaliação.** Qual é a cobertura do código-fonte encontrada por cada uma das técnicas de busca?

   Essa questão compara as técnicas de acordo com a cobertura que sua melhor solução provê sobre cada uma das instâncias selecionadas. É esperado que o GA encontre soluções que provêem melhor cobertura do que o HC e o RS.

5.3 Instâncias utilizadas

Visando responder às questões de pesquisa apresentadas na seção anterior, o estudo utilizou dois grupos de instâncias: instâncias aleatórias e instâncias reais.

O primeiro grupo foi composto por quatro instâncias geradas aleatoriamente, com 50, 100, 150 e 200 casos de teste, respectivamente. Foi utilizado um gerador de instâncias desenvolvido por nosso grupo de pesquisa para criar estas instâncias de forma sistemática. Os testes foram classificados como positivos ou negativos de acordo com uma distribuição uniforme definida com probabilidade de 50%, obtendo suítes compostas por aproximadamente metade dos casos de teste positivos e metade negativos. O tempo necessário para executar um caso de teste foi escolhido aleatoriamente, de acordo com uma distribuição de probabilidade uniforme variando de 0 a 60 segundos. A cobertura de um caso de teste também foi selecionada aleatoriamente, de acordo com uma distribuição de probabilidade uniforme que varia de 1% a 100% do módulo de código-fonte associado ao caso de teste. Um conjunto de módulos de código-fonte foi gerado e cada módulo foi associado a um número aleatório de casos de teste, selecionados a partir de uma distribuição de probabilidade uniforme variando de 1 a 10 casos da suíte de testes. Finalmente, um conjunto de funcionalidades foi gerado e cada qual foi associada com um número de módulos variando de 1 a 10 (associação uniformemente distribuída), onde cada módulo escolhido contribuiu de 1% a 100% (também uniformemente distribuído) para a implementação da funcionalidade a qual foi relacionado.

Para cada instância aleatória, cada técnica de busca foi executada 30 vezes em quatro configurações distintas, utilizando 10%, 20%, 50% e 80% do tempo necessário para executar todos os testes que compõem a instância como limite de tempo para a execução dos testes críticos. Assim, o estudo avaliou cada algoritmo utilizando instâncias de tamanho variável (número de casos de teste) e complexidade variável (fração de tempo disponível para executar o conjunto de testes). Em cada execução, coletamos o valor de cada medida de interesse (*reward*, esforço e cobertura) para a melhor solução. Em seguida, os valores médios e desvios-padrão para essas medidas foram calculados. Finalmente, um teste de inferência estatística não-paramétrico (*Wilcoxon-Mann-Whitney*) (FELTOVICH, 2003) foi utilizado para determinar, para cada par de técnicas de busca, se uma técnica apresentou resultados significativamente melhores do que a outra para cada medida de interesse.

O segundo grupo de instâncias usou uma biblioteca de código aberto, chamada FileHelpers, e um sistema de informação de uma companhia de seguros, chamado Syscommision, como exemplos de sistemas do mundo real.

FileHelpers é uma biblioteca .NET para leitura/gravação de dados fortemente tipados para/a partir de arquivos com registros de tamanho fixo ou delimitado (CSV, por exemplo). A biblioteca possui um conjunto de conversores para os tipos primitivos e pode ser estendida para oferecer conversores personalizados. Ele também suporta a importação/exportação de dados de diferentes fontes de dados (como Excel, SQLServer e MySQL). O código-fonte completo da biblioteca pode ser acessado por meio de um repositório Subversion no SourceForge. FileHelpers foi escolhida como uma instância representativa do mundo real porque a biblioteca possui um conjunto de testes relativamente grande (420 casos de teste), cobrindo cerca de 78% do seu código-fonte. Todos os casos de teste foram escritos em NUnit e as alterações feitas no código-fonte da biblioteca foram controladas por um sistema de rastreamento (TRAC) mapeado no procedimento de coleta de dados descrito no capítulo 4. Foi utilizada a última alteração feita no código-fonte para caracterizar uma alteração de emergência e aplicar o processo de otimização. Esta mudança afetou 13 módulos de código-fonte (de um total de 92 módulos) e envolveu 7 funcionalidades (de um total de 17). Todos os 352 casos de teste (100%) foram revistos como parte da alteração e sua execução leva cerca de 1 hora. O computador utilizado foi o mesmo dos testes da instância Syscommision e das outras instâncias (aleatórias e FileHelpers). O computador possui *chipset* Intel, um I3 com três gigabytes de memória RAM do tipo DDR3 e 520 gigabytes de disco rígido de 7200 rpm SATA. O computador estava executando Windows 7, versão de 64 bits e foi utilizado em regime de exclusividade.

Syscommision é um sistema para efetuar cálculos de impostos municipais sobre as comissões de venda de produtos de seguros realizadas pelos corretores da Mongeral Aegon. O sistema foi escrito em .NET com uma arquitetura web dividida em três camadas. Sua base de dados é Microsoft SQL Server e ele é alimentado por XML WebServices de vários outros sistemas da companhia. Ele foi escolhido como instância real porque sua gestão é realizada com o apoio de TRAC e SVN, sendo possível aplicar o método de coleta de dados proposto no capítulo 4. Também foi escolhido por representar um Sistema de Informação em uso na indústria. Syscommision possui uma suíte de testes unitários escritos em NUNIT. Sua suíte é composta por 970 testes, que cobrem cerca de 65% do seu código-fonte. Para essa instância foi utilizada uma alteração de emergência real, onde um erro foi encontrado no ambiente de produção durante a execução da operação de pagamento de impostos. Essa mudança afetou 4 módulos (de um total de 22) e 7

funcionalidades (de um total de 230). Nessa alteração, 475 testes foram incluídos por cobrir diretamente os módulos afetados. Estes testes levam cerca de duas horas para executar.

Para classificar os casos de teste da FileHelpers e do Syscommision em testes positivos ou negativos utilizamos uma estratégia baseada na convenção de nomes adotada por esses dois projetos: os desenvolvedores utilizam a string "_bad" como prefixo ou sufixo para identificar os testes negativos. Assim, todos os casos de teste cujos nomes começam com esse prefixo ou terminam com o sufixo foram classificados como testes negativos, enquanto qualquer outro teste foi classificado como um teste positivo. Depois de aplicar este procedimento de classificação, todos os casos de teste foram verificados manualmente a fim de certificar de que foram corretamente classificados.

5.4 Técnicas de Busca Utilizadas

Algoritmos Genéticos são técnicas de busca heurística para otimização inspiradas no processo natural de evolução de populações. Segundo (GOLDBERG, 1989), algoritmos genéticos usam regras de transição probabilística que lhes permitem analisar simultaneamente uma grande área do espaço de busca e trabalhar com um grande número de parâmetros complexos. Estas características permitem-lhes escapar de mínimos locais, tornando-os excelentes opções para oferecer soluções boas para problemas de otimização combinatória.

Nesse estudo experimental foi usada uma codificação binária para representar as soluções para todas as técnicas de busca. Cada solução foi composta de M bits, onde M é o número de casos de teste da instância que está sendo analisada. Se um caso de teste está selecionado na suíte de testes representada por uma solução, o seu bit é preenchido com um. Caso contrário, o bit é definido para zero.

O algoritmo genético foi configurado com os parâmetros a seguir. O tamanho da população foi definido como o número de casos de testes na instância. Elitismo foi utilizado para preservar os dois melhores indivíduos em uma população de descendentes. O operador de *crossover* escolhido foi o *single-point crossover* (com 80% de probabilidade de executar o crossover) e os indivíduos foram escolhidos para participar como pais em cada cruzamento pela técnica de torneio[6]. O operador

[6] Escolhe-se k (tipicamente 2) indivíduos aleatoriamente da população e entre eles o melhor (valor) é selecionado.

de mutação utilizado foi o *single-point mutation* (com probabilidade 0,5 / N, onde N é o número de indivíduos na população).

Um mesmo número de avaliações foi concedido para os três algoritmos no estudo experimental. Esse *bugdet* de avaliações é definido por uma constante (no caso, 100) multiplicada pelo tamanho da instância (quantidade de testes dentro da instância).

Hill Climbing é um algoritmo iterativo que parte de uma solução arbitrária (geralmente aleatória) para o problema e tenta encontrar uma solução melhor alterando, de forma incremental, um único elemento da solução em cada tentativa. De modo a que seu custo pudesse ser comparado às demais técnicas de busca, no estudo experimental o algoritmo foi modificado para reiniciar a busca de outro ponto aleatório quando encontrar um ótimo local, ou seja, quando não puder melhorar uma solução após visitar todos os seus vizinhos. O reinício aleatório continua enquanto houver *budget* de avaliações disponíveis. Esse tipo de implementação é chamado de *Hill Climbimg Random Restart* e foi utilizada por Praditwong et al. (PRADITWONG, 2010) para tratar o problema de clusterização de módulos em projetos de software.

Para assegurar que a seleção de casos de testes para alterações críticas não é trivial e que um processo de busca sistemática é necessário para encontrar soluções adequadas, os resultados do algoritmo genético e do Hill Climbing foram comparados com uma técnica de Random Search (Busca Aleatória). O Random Search é um algoritmo iterativo que busca soluções para um problema por amostragem aleatória, armazenando a melhor solução encontrada até o momento. Este algoritmo também executa até que o número de avaliações concedido acabe.

5.5 Análise dos Resultados

Esta seção apresenta os resultados do estudo experimental que comparou o desempenho do GA com o desempenho do HC e do RS em relação às questões de pesquisa definidas na seção 5.1. Ambos GA e HC encontraram suítes de testes aceitáveis para todas as instâncias utilizadas no estudo. Por outro lado, RS não foi capaz de encontrar soluções viáveis em todas as situações.

O estudo experimental foi executado utilizando o computador descrito na seção 5.3. O computador estava executando os experimentos em regime de exclusividade, ou seja, nenhum outro software que não o jMetal (sob Eclipse) estava sendo

executado durante os experimentos. O tempo de avaliação de uma instância executando os três algoritmos propostos foi, em média, de 3 horas de processamento.

Esta seção está organizada em duas partes: a primeira apresenta os resultados coletados a partir do estudo experimental que usou instâncias geradas aleatoriamente, enquanto a segunda apresenta os resultados coletados a partir do estudo que utilizou as instâncias do mundo real.

5.5.1 Instâncias aleatórias

O estudo experimental foi executado com instâncias de tamanho (50, 100, 150 e 200 casos de teste) e complexidade variável (10%, 20%, 50% e 80% do tempo necessário para executar os conjuntos de testes completos). A notação "xT y%" nas tabelas a seguir representa os resultados coletados a partir dos 30 ciclos executados para a configuração da instância com T casos de teste e que usa y% do tempo necessário para executar todos os testes.

A Tabela 1 apresenta a média e o desvio padrão das 30 execuções para a função *Reward* para as três técnicas de busca utilizadas. Valores em negrito nas colunas referentes ao GA e o HC são significativamente distintos do outro algoritmo com 95% de certeza. O teste estatístico foi utilizado apenas para comparar os resultados coletados a partir da execução do GA e do HC, uma vez que os resultados do RS foram claramente inferiores a ambos.

Tabela 1 – Valores de *Reward* coletados para as instâncias aleatórias

Instância	***GA***	***HC***	***RS***
50T 10%	**622 ± 25**	290 ± 93	n/a
50T 20%	**1.268 ± 33**	734 ± 134	n/a
50T 50%	**2.652 ± 97**	2.373 ± 117	2.020 ± 65
50T 80%	3.464 ± 1	3.464 ± 1	2.340 ± 98
100T 10%	**1.117 ± 52**	397 ± 152	n/a
100T 20%	**2.018 ± 60**	851 ± 232	n/a
100T 50%	**3.687 ± 61**	3.366 ± 218	2.771 ± 62
100T 80%	5.173 ± 5	5.174 ± 9	3.070 ± 91
150T 10%	**1.364 ± 159**	186 ± 138	n/a
150T 20%	**3.033 ± 131**	901 ± 222	n/a
150T 50%	**5.977 ± 84**	5.802 ± 179	3.958 ± 63
150T 80%	6.463 ± 265	**6.643 ± 190**	4.244 ± 83
200T 10%	**1.387 ± 239**	337 ± 159	n/a
200T 20%	**3.412 ± 212**	1.476 ± 408	n/a
200T 50%	7.326 ± 55	7.237 ± 226	4.808 ± 90
200T 80%	7.745 ± 186	**8.143 ± 209**	5.074 ± 102

Em relação à questão de pesquisa RQ1, podemos observar que o GA é capaz de produzir valores de *Reward* melhores do que o HC quando a restrição de tempo é pequena, isto é, quando o procedimento procura uma solução que use apenas 10% ou 20% do tempo necessário para executar todo o conjunto de testes. Em tais situações, RS geralmente não consegue encontrar soluções adequadas (resultados representados como "N/A" na Tabela 1).

No entanto, a diferença na eficácia entre GA e HC diminui conforme o tempo disponível para a execução da solução aumenta. Em tais situações, mesmo o RS é capaz de encontrar soluções relativamente boas, especialmente para instâncias com poucos casos de teste. Portanto, a hipótese apresentada juntamente com RQ1 – que o GA é capaz de encontrar soluções melhores do que o HC ou RS – foi confirmada para configurações com restrições de tempo rigorosas (cenário comum para apoiar mudanças críticas). Por outro lado, o HC é capaz de produzir soluções semelhantes e às vezes melhores quando a instância tem espaço para acomodar mais casos de teste (maior tempo disponível para execução).

A Tabela 2 mostra a média e o desvio padrão do número de avaliações (custo computacional) da função *Reward* exigidos por cada algoritmo para encontrar a melhor solução para cada configuração ao longo de 30 execuções. Comparando-se o GA com o HC, valores em negrito representam que o algoritmo é capaz de encontrar sua melhor solução usando um número significativamente menor de avaliações. Novamente, o RS não foi comparado por ser incapaz de encontrar soluções com qualidade compatível às encontradas nos outros dois algoritmos. Em relação a RQ2, os resultados mostram que o GA exige um esforço adicional de tempo de processamento para encontrar as melhores soluções apresentadas na Tabela 1.

Como pode ser observado na Tabela 2, HC encontra a sua melhor solução com cerca de metade do número total de avaliações que o GA. No entanto, à medida que avançamos para configurações com 50% do tempo necessário para executar a suíte completa, a diferença em termos de eficiência na comparação GA e HC é reduzida para cerca de 41%. Além disso, em configurações com 80% do tempo necessário para executar a suíte de testes completa, essa diferença cai para cerca de 30%. Assim, enquanto a Tabela 1 mostra que a GA pode encontrar as melhores soluções para configurações com limites mais curtos, a Tabela 2 mostra que o algoritmo requer mais avaliações e, portanto, o processo consome tempo que poderia ser gasto executando de fato os testes.

Tabela 2 – Custo coletado usando instâncias aleatórias.

Instância	*GA*	*HC*	*RS*
50T 10%	4.073 ± 730	**2.343 ± 1.305**	n/a
50T 20%	3.935 ± 848	**2.524 ± 1.322**	n/a
50T 50%	3.919 ± 812	**2.497 ± 1.221**	2.168 ± 1.516
50T 80%	2.470 ± 865	**2.071 ± 1.216**	2.588 ± 1.405
100T 10%	9.711 ± 334	**6.372 ± 2.227**	n/a
100T 20%	9.598 ± 393	**4.617 ± 2.234**	n/a
100T 50%	9.655 ± 415	**5.596 ± 2.703**	5.163 ± 3.093
100T 80%	8.864 ± 837	**5.320 ± 1.490**	4.588 ± 2.841
150T 10%	14.837 ± 264	**9.186 ± 1.163**	n/a
150T 20%	14.843 ± 186	**8.766 ± 2.506**	n/a
150T 50%	14.805 ± 255	**9.659 ± 2.480**	7.324 ± 4.150
150T 80%	14.917 ± 225	**11.257 ± 1.088**	8.441 ± 4.366
200T 10%	19.790 ± 176	**16.818 ± 1.390**	n/a
200T 20%	19.716 ± 238	**15.796 ± 2.014**	n/a
200T 50%	19.702 ± 219	**15.391 ± 2.058**	10.184 ± 5.645
200T 80%	19.740 ± 158	**19.287 ± 896**	10.434 ± 5.058

Finalmente, a Tabela 3 apresenta a média e o desvio padrão para a cobertura proporcionada pela melhor solução encontrada para cada configuração e algoritmo ao longo das 30 execuções. Quanto à questão de pesquisa RQ3, podemos observar que o GA geralmente encontra soluções que proporcionam maior cobertura, mas HC também é capaz de encontrar soluções competitivas em termos de cobertura (inclusive, encontrou a melhor cobertura para a instância "200T 80%") quando a restrição de tempo para selecionar casos de testes é branda. Portanto, parece que o GA pode ser útil quando a restrição de tempo é muito rigorosa, o que é o caso para o problema de seleção de casos de teste para mudanças críticas.

Tabela 3 – Cobertura coletada usando instâncias aleatórias.

Instância	*GA*	*HC*	*RS*
50T 10%	**728 ± 21**	454 ± 113	n/a
50T 20%	**1.187 ± 50**	802 ± 123	n/a
50T 50%	1.829 ± 63	1.819 ± 75	1.654 ± 80
50T 80%	2.183 ± 2	2.184 ± 2	1.904 ± 142
100T 10%	**1.526 ± 70**	997 ± 140	n/a
100T 20%	**2.209 ± 69**	1.485 ± 251	n/a
100T 50%	**3.433 ± 74**	3.384 ± 126	3.019 ± 156
100T 80%	4.142 ± 22	4.151 ± 4	3.436 ± 178
150T 10%	**1.993 ± 142**	1.059 ± 208	n/a
150T 20%	**3.247 ± 135**	2.143 ± 213	n/a
150T 50%	**5.443 ± 103**	5.289 ± 135	4.493 ± 246
150T 80%	5.908 ± 133	5.930 ± 97	5.007 ± 269
200T 10%	**2.117 ± 204**	1.237 ± 229	n/a

200T 20%	**3.682 ± 161**	2.792 ± 346	n/a
200T 50%	6.487 ± 111	6.429 ± 152	5.421 ± 191
200T 80%	7.020 ± 155	**7.219 ± 129**	5.867 ± 243

Algumas conclusões podem ser tiradas com base nos resultados coletados a partir do estudo experimental que usou instâncias geradas aleatoriamente. Primeiro, o GA parece oferecer as melhores soluções para configurações com uma restrição de tempo estrita, tanto em termos dos valores da função *reward* quanto de cobertura. Por outro lado, HC pode encontrar suas melhores soluções em cerca de metade do número total de avaliações exigido pelo GA para encontrar a sua melhor solução. Portanto, parece que HC pode ser usado quando a restrição do tempo permite a execução de mais da metade do conjunto total de teste, mas GA deve ser usado quando os casos de teste devem executar em menos de 50% do tempo necessário para executar o conjunto completo.

5.5.2 Instâncias do Mundo Real

O estudo experimental utilizou as duas instâncias do mundo real, com 352 e 475 casos de teste, respectivamente, ambas analisadas com as mesmas configurações utilizadas no estudo experimental descrito na Seção 5.5.1 (10%, 20%, 50% e 80% do tempo total necessário para executar a suíte de testes completa).

A Tabela 4 mostra a média e o desvio padrão para 30 execuções da função *reward* em cada configuração e algoritmo para a instância FileHelpers. Valores em negrito representam medidas significativamente distintas do outro algoritmo. Quanto à questão de pesquisa RQ1, podemos observar que o GA foi capaz de produzir valores da função *Reward* melhores do que HC para todas as configurações, apesar de ter um grande número de casos de teste.

Tabela 4 – Reward coletado usando a instância FileHelpers

Instância	***GA***	***HC***	***RS***
10%	**116.142 ± 8.608**	83.340 ± 13.551	n/a
20%	**149.207 ± 5.979**	106.732 ± 18.642	n/a
50%	**183.622 ± 4.533**	143.033 ± 13.984	65.545.0 ± 1.784
80%	**192.889 ± 5.094**	148.012 ± 8.682	68.181 ± 2.219

Em relação à questão de pesquisa RQ2, os resultados mostram que o GA repetiu o comportamento nos estudos com instâncias aleatórias: como mostrado na Tabela 5, HC encontra a sua melhor solução usando menos avaliações do que o GA, especialmente para os casos com uma forte restrição de tempo para executar

os casos de testes selecionados. Os empates entre GA e HC podem ser explicados pelo grande número de casos de teste, que permite facilmente encontrar uma solução com alta cobertura e sem penalidade (por isso, valor de *reward* elevado) quando houver tempo suficiente para acomodar um grande número de casos de teste. Assim, apesar de ter "melhores soluções" distintas, GA e HC convergem para estas soluções com quase o mesmo esforço.

Tabela 5 – Custo coletado usando a instância FileHelpers

Instância	***GA***	***HC***	***RS***
10%	34.044 ± 351	**24.867 ± 7.994**	n/a
20%	34.070 ± 183	**29.652 ± 5.117**	n/a
50%	33.975 ± 272	33.074 ± 2.319	16.904 ± 10.567
80%	34.087 ± 198	33.710 ± 87	15.572 ± 10.152

Finalmente, a Tabela 6 apresenta a média e o desvio padrão para a cobertura proporcionada pela melhor solução encontrada para cada configuração da instância FileHelpers e algoritmo ao longo de 30 execuções. Quanto à questão de pesquisa RQ3, podemos observar que o GA sempre encontra a melhor cobertura e que o HC é incapaz de encontrar soluções competitivas em termos de cobertura. No entanto, a diferença de qualidade entre as soluções encontradas pelo GA e HC diminui à medida que mais casos de teste podem ser selecionados como parte da solução.

Tabela 6 – Cobertura coletada usando a instância FileHelpers

Instância	***GA***	***HC***	***RS***
10%	**12.635 ± 382**	10.890 ± 882	n/a
20%	**13.904 ± 293**	12.274 ± 830	n/a
50%	**15.150 ± 144**	13.875 ± 584	9.954 ± 205
80%	**15.441 ± 175**	14.114 ± 346	10.224 ± 246

A Tabela 7 mostra a média e o desvio padrão para 30 execuções da função *reward* em cada configuração e algoritmo para a instância Syscommision. Quanto à questão de pesquisa RQ1, podemos observar que o GA foi capaz de produzir valores melhores para a função *reward* do que HC e RS para todas as configurações exceto na última. Esse foi o único caso em que o RS superou os valores encontrados para GA e HC. Essa instância é a que mais possui tempo disponível (quase o total). Porém, esse comportamento pode sugerir que com uma restrição branda não há necessidade de uma busca heurística para encontrar uma solução adequada e que a mesma pode ser encontrada aleatoriamente.

Tabela 7 – Reward coletado usando a instância Syscommision

Instância	*GA*	*HC*	*RS*
10%	**11.603 ± 8.597**	n/a	n/a
20%	**8.925.023 ± 3.701**	n/a	n/a
50%	**20.230.132 ± 14.610**	17.227.040 ± 13.999	15.647.361 ± 9.929
80%	15.821.198 ± 15.774	15.883.002 ± 14.680	**18.006.883 ± 9.757**

Em relação à questão de pesquisa RQ2, os resultados mostram que as técnicas de busca repetiram o comportamento apresentado anteriormente nas instâncias aleatórias e na instância FileHelpers. Conforme mostrado na Tabela 8, o HC encontra a sua melhor solução usando menos avaliações do que o GA, novamente para os casos com uma maior restrição de tempo para executar os casos de testes selecionados.

Tabela 8 – Custo coletado usando a instância Syscommision

Instância	*GA*	*HC*	*RS*
10%	**47.043 ± 545**	n/a	n/a
20%	**47.286 ± 394**	n/a	n/a
50%	47.198 ± 396	46.803 ± 140	23.498 ± 15.107
80%	47.348 ± 193	46.816 ± 116	20.985 ± 14.096

Finalmente, a Tabela 9 apresenta a média e o desvio padrão para a cobertura proporcionada pela melhor solução encontrada para cada configuração e algoritmo da instância Syscommision ao longo das 30 execuções. Quanto à questão de pesquisa RQ3, podemos observar que o GA sempre encontra a melhor cobertura e que o HC é incapaz de encontrar soluções competitivas em termos de cobertura. No entanto, a diferença entre as soluções encontradas pelo GA e o HC diminui à medida que mais casos de teste podem ser selecionados como parte da solução.

Table 9 – Cobertura coletada usando a instância Syscommision

Instância	*GA*	*HC*	*RS*
10%	**2.756 ± 418**	n/a	n/a
20%	**7.294 ± 452**	n/a	n/a
50%	**16.439 ± 197**	15.169 ± 387	12.982 ± 303
80%	17.767 ± 245	16.857 ± 366	13.949 ± 359

A capacidade de tirar conclusões com base nos estudos que utilizaram instâncias reais é limitada devido ao uso de apenas duas instâncias. No entanto, é interessante observar que se existem vários casos de teste para validar um pequeno número de módulos e há bastante tempo disponível para selecionar um subconjunto destes testes, a cobertura completa pode ser alcançada por várias soluções sem

penalidade. Em tais situações, o esforço exigido pelo GA e HC é similar, mas muitas vezes o GA encontra soluções com melhor valor de *reward* e cobertura.

5.6 Ameaças à Validade do Estudo

A validade externa diz respeito ao grau em que as conclusões de um estudo experimental podem ser generalizadas a toda a classe de indivíduos a partir da qual a amostra foi extraída (WOHLIN *et al* 2000). As fontes de ameaças à validade externa em estudos experimentais na área de Engenharia de Software Baseada em Buscas (SBSE) incluem a falta de uma definição clara das instâncias do problema, a ausência de uma estratégia de seleção de dados e a falta de instâncias de tamanho e complexidade crescentes (BARROS et. al. 2011) . Para tratar essas ameaças, as instâncias foram disponibilizadas para fins de pesquisa (ver seção 5.1), o procedimento de coleta de dados adotado foi discutido no capítulo 4 e aplicado no estudo experimental e foram utilizadas instâncias de quatro tamanhos e quatro medidas de complexidade distintas.

Validade interna é o grau em que se pode tirar conclusões sobre o efeito causal das variáveis independentes sobre as variáveis dependentes no âmbito de um estudo experimental (PRADITWONG, 2010). As fontes de ameaças à validade interna em estudos experimentais na área de Engenharia de Software Baseada em Buscas incluem a falta de informação sobre os parâmetros utilizados na execução das técnicas de busca, a falta de uma discussão sobre a instrumentação do código-fonte utilizado no estudo, a falta de uma descrição clara dos procedimentos de coleta de dados e a ausência de instâncias do mundo real no estudo. Estas ameaças foram abordadas usando instâncias aleatórias e do mundo real, apresentando-se um procedimento de coleta de dados por meio do qual as instâncias do mundo real foram construídas, tornando o código-fonte disponível para fins de pesquisa e descrevendo-se precisamente a parametrização dos algoritmos utilizados no estudo experimental.

Ameaças à construção tratam das relações entre teoria e observação. As principais fontes de ameaças à validade em construção de em estudos experimentais na área de Engenharia de Software Baseada em Buscas incluem a falta de uma avaliação da validade da métrica de custo, a falta de uma avaliação da validade das métricas de eficácia e de uma discussão sobre o modelo que suporta o processo de otimização. Estas ameaças foram tratadas ao discutir o modelo do problema de otimização (ver capítulo 4), usando-se uma métrica aceitável para

estimar o esforço despendido pelos algoritmos (número de avaliações da função de *fitness* necessárias para encontrar a melhor solução) e ao detalhar as duas métricas de eficácia utilizadas no estudo (cobertura e *reward*, como descrito na Seção 5.2).

Finalmente, as ameaças às conclusões validam a relação entre o tratamento e o resultado em um estudo experimental. As principais fontes de ameaças as conclusões em estudos experimentais na área de Engenharia de Software Baseada em Buscas incluem desconsiderar a natureza aleatória dos algoritmos de busca heurística, a falta de uma base de comparação relevante para avaliar um algoritmo de busca heurística e de testes de inferência estatística. Estas ameaças foram abordadas neste estudo pela execução de vários ciclos para cada configuração e algoritmo (30 execuções), apresentando média e desvio padrão para cada medida de eficácia e eficiência e comparando esses valores usando um teste estatístico não-paramétrico (Wilcoxon-Mann-Whitney).

5.7. Conclusões

Foram apresentados dois estudos experimentais comparando a eficácia e a eficiência de uma abordagem de busca heurística, uma pesquisa local, e uma pesquisa não-sistemática sobre a busca de soluções para o problema de seleção de casos de teste para alterações críticas.

Foi observado que a abordagem de busca heurística (GA) encontrou as melhores soluções em termos de cobertura e diversidade do conjunto de testes selecionados, especialmente quando a restrição imposta foi uma pequena fração do tempo necessário para executar a suíte de testes completa. Foi observado também que a busca local (HC) pode fornecer soluções competitivas com menos esforço computacional no caso de um maior período de tempo disponível, permitindo a acomodação de mais casos de teste da suíte selecionada.

CAPÍTULO 6

CONCLUSÕES

6.1. Considerações Finais

Neste trabalho foi apresentada uma técnica para selecionar casos de teste para alterações críticas, geralmente feitas em ambiente de produção do software sob testes com um limite de tempo bem definido e sempre menor do que o necessário para executar todos os testes. Tratando-se de um problema de análise combinatória complexo, o problema de seleção dos casos de teste críticos pode fazer uso de heurísticas para encontrar soluções boas dentro de um tempo razoável. Vários estudos anteriores sugerem o uso de heurísticas e implementam diversos casos em estudos experimentais, conforme mostrado nos capítulos 2 e 3.

Para avaliar o uso de heurísticas na busca de soluções para o problema de seleção de testes críticos foi conduzido um estudo experimental com dois grupos de instâncias distintas. O primeiro grupo era composto por instâncias geradas aleatoriamente por um gerador de instâncias próprio. Já o segundo grupo contou com dois sistemas do mundo real, utilizados na indústria de seguros brasileira. O objetivo do experimento foi avaliar o uso de Algoritmos Genéticos na busca de soluções do problema de seleção de testes críticos. Para base de comparação dos resultados foram utilizadas duas outras técnicas de busca: uma busca local multi-partida com uso de estratégia chamada Hill Climbing e uma busca aleatória que, apesar de não se tratar de uma busca heurística, é comumente utilizada nesse tipo de experimento como base para validação de resultados.

No primeiro grupo de instâncias foi observado, de maneira geral, que o Algoritmo Genético encontra as melhores soluções. Porém, em diversas situações em que o limite de tempo foi igual ou maior a 50% do tempo necessário para executar os casos de teste disponíveis, o Hill Climbing encontra soluções muito próximas do Algoritmo Genético, porém com um custo em avaliações bem menor. No segundo grupo de instâncias o comportamento se manteve. Mesmo sendo observadas algumas nuances entre o comportamento das três técnicas utilizadas,

na média os resultados se mantiveram similares aos encontrados nas instâncias do primeiro grupo.

Aplicar a técnica em softwares em desenvolvimento e em evolução constante envolve algumas atividades conforme descrito no capítulo 4, a saber: (i) Coletar os dados necessários sobre as entidades do modelo do problema, seguindo o procedimento proposto; (II) Relacionar os testes, classificá-los e definir seus percentuais de cobertura sobre os módulos de código-fonte; (III) Relacionar as funcionalidades aos módulos que as implementam; (IV) Definir o percentual de testes positivos desejados na suíte; e (V) Definir o limite de tempo disponível para execução dos testes.

No contexto de uma alteração crítica é importante prover um mecanismo que possibilite ao gestor do sistema ou ainda a equipe de testes diminuir o risco de erros de tal forma que alguns testes possam ser executados sobre o código alterado. O método proposto permite que o usuário da técnica calibre o modelo com o percentual de tipo de teste, objetivando mitigar esse risco. Testes positivos atestam corretude enquanto testes negativos atestam estabilidade. Um gestor envolvido com a alteração pode determinar a melhor abordagem de testes visando cobrir um desses aspectos levando em consideração dois fatores: (i) quais os tipos de alterações foram realizadas; e (II) tempo disponível para executar os testes.

6.2. Contribuições

As principais contribuições desse trabalho são:

- Definição e formalização do problema de seleção de testes críticos, adaptado do problema original de seleção de casos de teste, no contexto de Engenharia de Software Orientada a Buscas (*Search Based Software Engineering - SBSE*);
- Definição e implementação de uma técnica para seleção de testes críticos baseada em Algoritmos Genéticos;
- Definição de um procedimento para coleta de dados para execução da técnica proposta, permitindo sua repetição automática;
- Definição, planejamento, execução e análise dos resultados de um estudo experimental para avaliar a viabilidade da abordagem proposta;

6.3. Limitações e Perspectivas Futuras de Trabalho

Todo estudo experimental possui limitações e ameaças a sua validade. Conforme mostrado no capítulo 5, algumas ações foram tomadas para tratar às ameaças a validade externa, interna, construção e conclusão do estudo. Porém, podemos destacar algumas limitações:

- tempo é um critério decisivo no contexto de alterações críticas. Portanto, o tempo para análise e descoberta de uma solução deve ser subtraído do tempo disponível para executar os testes e este precisa ser o mínimo possível;
- conceito de testes positivos e negativos, utilizado na modelagem do problema, é recente e ainda pouco discutido no contexto de Engenharia de Software;
- A classificação dos testes é manual, ou seja, caso a suíte não possua um padrão claro para identificá-los, será necessário fazê-lo a cada caso. Esse tipo de classificação pode conter erros que prejudicariam os resultados da técnica.

Uma evolução natural desse trabalho seria automatizar o procedimento de coleta dos dados. Visto ser uma etapa manual e que requer um tempo considerável, a aplicação da técnica em um ambiente real da indústria não serial possível sem o desenvolvimento de um aplicativo para tal. Outras possíveis evoluções do trabalho podem ser citadas:

- A proposta de solução só leva em consideração testes de unidade. Uma alteração no modelo para contemplar outros tipos de teste pode ser relevante. Incluir no modelo, por exemplo, testes de aceitação, permitiria aumentar as opções de cobertura das funcionalidades alteradas;
- A efetividade da cobertura encontrada nos experimentos não foi avaliada, ou seja, validar se a cobertura encontrada pela melhor solução da técnica realmente encontra mais defeitos. Assim, é importante conduzir um estudo sobre a efetividade da técnica contra a garantia procurada;
- Aplicar técnicas heurísticas novas que mostram resultados superiores ao Algoritmo Genético. LOIOLA et al. (2009) apresenta uma técnica baseada em GRASP que poderia ser adaptada para estudo em experimento similar.

REFERÊNCIAS

BARROS, M, NETO, A. *Threats to Validity in Search-based Software Engineering Empirical Studies*. 2011.

BARTIÉ, A (2002), *Garantia da qualidade de software*, São Paulo, Elsevier.

BAUDRY B., FRANCK F., JEAN-MARC J. E YVES L. *Automatic test cases optimization: a bacteriologic algorithm*. IEEE Software, 22(2):76--82, March 2005.

LOIOLA C., CARMO R., FREITAS F., CAMPOS G. e SOUZA J., *Automated Test Case Prioritization with Reactive GRAS*", Hindawi Publishing Corporation, Advances in Software Engineering, Volume 2010, Article ID 428521, 18 pages, doi:10.1155/2010/428521, 14 October 2009.

CHEN Y., ROSENBLUM D., VO KP. *TestTube: A system for selective regression test*. Proveedgins of the 16th International Conference on software Engineering (ICSE 1994), ACM Press, 1994, 211-220.

Wohlin C., Runeson P., Host M., Ohlsson M.C., Regnell B.. *Wesslen, Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

PIANKA E.R., *Evolutionary Ecology*, Addison Wesley, 1999.

FELTOVICH, N. *Nonparametric Tests of Differences in Medians: Comparison of the Wilcoxon–Mann–Whitney and Robust Rank-Order Tests*. Experimental Economics, Springer Netherlands, 2003.

GOLDBERG, D.E., *Genetic Algorithms in Search, Optimization & Machine Learning*. Massachusetts: Adison Wesley Logman, 1989.

HARROLD, M. J.; SOFFA, M. L. *Selecting and using data for integration test*. IEEE software, v. 8, n. 2, p. 58-65, Mar 1991.

HOLGER H. e STÜZLE T., *Stochastic Local Search Foundations and Applications*", 2005

VOAS J. M. e MILLER K., *The Revealing Power of a Test Case*. Software Testing, Verification and Reliability, vol. 2, pp. 25-42, 1992.

DURILLO J.J., NEBRO A.J., ALBA E. *The jMetal Framework for Multi-Objective Optimization: Design and Architecture*. Lecture Notes in Computer Science, July 2010, pages 4138-4325.

PRADITWONG K., HARMAN M., YAO X. *Software Module Clustering as a Multi-Objective Search Problem*. IEEE Transactions on Software Engineering.

KOREL B, KOUTSOGIANNAKIS G, TAHAT L. *Application of system models in regression test suite priorization*. Proceedings of IEEE International Conference on Software Maintenance (ICSM 2008, 2008, 247-256).

LASKI J, SZERMER W. *Identification of program modifications and its applications in software maintenance*. Proceedings of the International Conference on Software Maintenance (ICSM 1992), IEEE Computer Society Press, 1992, 282-290.

LEUNG HKN, WHITE L. *Insigth into regression test*. Proceeedings of international conference on software maintenance (ICSM 1989), IEEE Computer society press 1989, 60-69.

LIVRAMENTO, S. *Algoritmo genético para o problema de localização de recursos em rede telefônica*. Master's thesis, Universidade Estadual de Campinas (2004).

FOWLER M., *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

RESENDE M. e RIBEIRO C.. *Greedy randomized adaptative search procedures*. in Handbook of Metaheuristics, F. Glover and G. Kochenberger, Eds., pp. 219–249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

HARROLD M.J. e ROTHERMEL G., *Empirical Studies of a Prediction Model for Regression Test Selection*. IEEE Trans. On Software Eng., vol. 27, no. 3, Mar. 2001.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, July 1991.

MALISHEVSKY A, ROTHERMEL G, ELBAUM S. *Modeling the cost-benefits tradeoffs for regression testing techniques*. Proceedings of the International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society Press, 2002, 230-240.

MANSOUR N. *Regression Test Selection for C# Programs*. Advances in Software Engineering, Volume 2009, Article ID 535708, May 2009.

MARK H., MCMINN P. *A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation*. IEEE Transactions on Software Engineering. 36(2): 226-247, 2010.

MOLINARI, L., 2007, *Gerência de Configuração - Técnicas e Práticas no Desenvolvimento do Software*. Florianópolis. Visual Books.

MYERS, B. *Object-Oriented software construction*. 2nd ed. Upper Saddle River: Prentice Hall, 1997.

MYERS, GLENFORD. *The Art of Software Testing*. John Wiley & Sons, 1979. Primeira edição.

NYMAN, JEFF. *Positive and Negative Testing*. GlobalTester, TechQA, http://www.sqatester.com/methodology/PositiveandNegativeTesting.htm, last access in March 29th, 2010.

PRADITWONG, K., HARMAN, M., YAO, X.. *Software Module Clustering as a Multi-Objective Search Problem*. IEEE Transactions on Software Engineering, Early Access Papers, 2010.

PRESSMAN, S., ROGER (2005). *Software Engineering - A Practitioner's Approach*. 6 ed., New York, McGraw-Hill.

DEMILLO R., LIPTON R., SAYWARD F. *Hints on Test Data Selection : Help For The Practicing Programmer*. IEEE Computer, vol. 11, pp. 34-41, 1978.

KRISHNAMOORTHI R., S.A.SAHAAAYA ARUL MARY. Regression Test Priorization using Genetic Algorithms, International Journal of Hybrid Information Technology. Vol.2, No.3, July, 2009.

KICINGER R., ARCISZEWSKI T.e KENNETH A. DE JONG. *Evolutionary computation and structural design: A survey of the state of the art*. Computers & Structures, 83(23-24) : 1943-1978.

REEVES CR. *Modern Heuristic Techniques for Combinatorial Problems*.McGraw-Hill 1995, Chapter 7.

ROTHERMEL G, HARROLD M, RONNE J, HONG C. *Empirical studies of test suíte reduction*. Software Testing, Verification and Reability. Dezembro de 2002.

ROTHERMEL G, HARROLD M. *A Framework for evaluation regression test selection techniques*. Proceedings of the 16th International Conference on software Engineering (ICSE 2008), ACM Press, 2008; 201-210.

ROTHERMEL G, HARROLD M. *A safe, efficient algorithm for regression test selection*. Proceedings of International Conference of software Maintenance (ICSM 1993), 358-367.

ROTHERMEL G, HARROLD M. Analyzing regression test selection techniques. IEEE transactions on software Engineering, Agosto de 1996, 22(8), 529-551.

ROTHERMEL G, HARROLD M. *Selecting tests and identifying test coverage requirements for modified software*. Proceedings of International Symposium on software Testing and Analusis (ISSTA 1994), ACM Press 1994, 169-184.

ROTHERMEL G, UNTCH RJ, CHU C. *Priorizing test cases for regression testing*. IEEE Transactions on Software Engineering, Outubro de 2001.

LIN S. *Computer Solutions of the Travelling Salesman Problem*. Bell System Technical J., vol. 44, pp. 2245-2269, 1965.

KHURI S., BÄACK T., HEITKÄOTTER J.. *The zero/one multiple knapsack problem and genetic algorithms.* In Proceedings of the ACM Symposium on Applied Com-puting, pages 188{193. ACM Press, 1994.

SRIKANTH H, WILLIAMS L, OSBORNE J. *System test case priorization of new and regression test cases*. Proceedings of the International Symposium on Empirical Software Engineering, 2005, 64-73.

SRIVASTAVA A, THIAGARAJAN J. *Effectively prioritizing tests in development environment*. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002), ACM Press, 2002; 97-106.

TAHA AB, THEBAUT SM, LIU SS. *An approach to software fault localization and revalidation based on incremental data flow analysis*. Proceedings of international Computer software and Applications conference (COMPSAC 1989), IEEE Computer Society Press, 1989; 527-534.

WALCOTT KR., SOFFA ML., ROOS RS., KAPFHAMMER GM.. *Time aware test suite riorization*, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006), ACM Press 2006, 1-12.

WHITE L, JABER K, ROBINSON B. *Extended Firewall for regression testing: na experience report*. Journal of Software Maintenance and Evolution 2008, 20(6):419-433.

WHITE LJ, LEUNG HKN. *A firewall concept for both control-flow and data-flow in regression integration testing*. Proceedings of International Conference on Software Maintenance (ICSM 1992), IEEE Computer Society Press, 1992; 262-271.

YOO S., HARMAN M.. *Regression Testing Minimization, Selection and Priorization – A Survey*. Centre for Research on Evolution, Search & Testing. King's College London. Outubro de 2009.

LI Z., HARMAN M., e HIERONS R.. *Search Algorithms for Regression Test Case Prioritization*. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 4, APRIL 2007