UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Guilherme Bolzan Monteiro

ANÁLISE DE APLICABILIDADE DE *SEARCH-BASED SOFTWARE ENGINEERING* EM GERÊNCIA DE REQUISITOS EM PROJETOS DE SOFTWARE

Santa Maria, RS
2018

Guilherme Bolzan Monteiro

ANÁLISE DE APLICABILIDADE DE *SEARCH-BASED SOFTWARE ENGINEERING* EM GERÊNCIA DE REQUISITOS EM PROJETOS DE SOFTWARE

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

ORIENTADORA: Prof.ª Lisandra Manzoni Fontoura

Trabalho de Graduação Nº 450
Santa Maria, RS
2018

Guilherme Bolzan Monteiro

ANÁLISE DE APLICABILIDADE DE *SEARCH-BASED SOFTWARE ENGINEERING* EM GERÊNCIA DE REQUISITOS EM PROJETOS DE SOFTWARE

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

Aprovado em 10 de dezembro de 2018:

Lisandra Manzoni Fontoura, Dra. (UFSM)
(Presidenta/Orientadora)

Giliane Bernardi, Dr. (UFSM)

Henrique Michel Persch, Me. (UFSM)

Santa Maria, RS
2018

RESUMO

ANÁLISE DE APLICABILIDADE DE *SEARCH-BASED SOFTWARE ENGINEERING* EM GERÊNCIA DE REQUISITOS EM PROJETOS DE SOFTWARE

AUTOR: Guilherme Bolzan Monteiro
ORIENTADORA: Lisandra Manzoni Fontoura

O desenvolvimento de *software* envolve uma série de etapas que têm como objetivo final a entrega do *software* funcionando, cumprindo com as necessidades do cliente e dentro do orçamento e prazo previstos para seu desenvolvimento. A etapa de requisitos tem como objetivo definir espeficação de características ou propriedades do sistema, assim como definir restrições de operação de acordo com as necessidades do cliente. A priorização de requisitos é um problema importante da Engenharia de Requisitos, pois busca implementar os requisitos que irão proporcionar maior valor agregado ao *software* em desenvolvimento a cada *release*. Ao se ter em mãos um documento com todos os requisitos solicitados pelo cliente, um dos papéis do Gerente de Projetos é priorizar os requisitos usando critérios para que os requisitos mais importantes sejam desenvolvidos primeiro. Engenharia de *Software* Baseada em Busca (*Search-Based Software Engineering* – SBSE) tem sido utilizada com sucesso para a resolução de problemas complexos relacionados à Engenharia de *Software*. A técnica SBSE apresenta alguns pontos positivos para a priorização de requisitos: é uma técnica genérica, robusta e que pode ser aplicada de forma direta no projeto em desenvolvimento. Portanto, o problema de priorização de requisitos pode ser formulado como um problema de busca. Este trabalho propõe utilizar as técnicas de busca multi-heurísticas: algortimos *Hill Climbing* e Genético, a fim de realizar a priorização de requisitos de um *software* com base em um conjunto de critérios ou métricas. Por fim, é proposta uma ferramenta que foi desenvolvida para apoiar o uso da abordagem de priorização dos requisitos. Essa ferramenta recebe como entrada uma lista de requisitos, cada um com suas métricas devidamente preenchidas, aplica os algoritmo de busca *Hill Climbing* e Genético sobre essa entrada e produz como saída uma lista com os requisitos classificados para serem implementados na próxima *release*. A validação da ferramenta foi feita com a realização de 50 testes, em que os algoritmos mostraram eficiência em produzir a resposta ótima do problema de priorização de requisitos.

Palavras-chave: Engenharia de *software*. Requisito. Busca. *Search-Based Software Engineering*. SBSE.

ABSTRACT

APPLICABILITY ANALYSIS OF SEARCH-BASED SOFTWARE ENGINEERING IN REQUIREMENTS MANAGEMENT IN SOFTWARE PROJECTS

AUTHOR: Guilherme Bolzan Monteiro
ADVISOR: Lisandra Manzoni Fontoura

The development of software involves a series of steps that aim to deliver the software functioning, fulfilling the needs of the client and within the budget and time frame for its development. The requirements phase has an objective to define specification of characteristics or properties of the system, as well as to define operating restrictions according to the client's needs. Requirements prioritization is an important problem to Requirements Engineering, because it seeks to implement the requirements that will bring higher added value to the software being developed at each release. By having a document with all the requirements requested by the client, one of the roles of the Project Manager is to prioritize the requirements utilizing some criteria so that the most important requirements are developed first. Search-Based Software Engineering - SBSE - has been successfully used to resolve complex problems related to Software Engineering. The SBSE technique has some positive points: it is a generic technique, robust and can be directly applied in the project under development. Therefore the requirements prioritization problem can be formulated as a search problem. This work proposes to utilize the techniques of multi-heuristic search: Hill Climbing and Genetic Algorithms, in order to make the requirements prioritization of a software based on a set of criteria or metrics. Lastly, it is proposed a tool that is been developed to support the use of the requirements prioritization approach. This tool receives as input a list containing the requirements, each one with its metrics filled, runs a search based algorithm on this input and generates as an output a list with the requirements classified to be implemented to the next release. The validation of the tool was made with the realization of 50 tests, on what the algorithms showed efficiency in produce the best answer of the requisites prioritization problem.

Keywords: Software Engineering. Requirement. Search. Search-Based Software Engineering. SBSE.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A Engenharia de Requisitos fornece uma base forte no desenvolvimento de um projeto de *software* (Pressman; Maxim, 2016). Os requisitos representam as necessidades do cliente para o sistema que será desenvolvido. Ao se definir os requisitos, é importante priorizá-los, para que os requisitos mais importantes do projeto sejam implementados primeiro, agregando o máximo de valor ao produto (Harman *et al*., 2012). Portanto, a priorização de requisitos é um problema importante da Engenharia de Requisitos.

Existem algumas técnicas de priorização de requisitos na literatura. Cita-se a técnica Moscow (Clegg; Barker, 1994), que classifica os requisitos como críticos, necessários, desejáveis e não desejáveis para o sistema. Outra técnica existente é a priorização baseada em valor, custo e risco (Wiegers, 1999), que propõe classificar numericamente cada requisito de acordo com o seu benefício, custo e risco e, após aplicar uma fórmula matemática sobre cada requisito, define-se sua prioridade.

A técnica *Search-Based Software Engineering* - SBSE - aplicada a requisitos propõe formular o problema de priorização de requisitos como um problema de busca e aplicar um algoritmo de busca sobre o espaço das possíveis soluções, para encontrar os requisitos com prioridade maior. A prioridade maior é definida de acordo com algumas métricas definidas pelos desenvolvedores do projeto. Zhang *et al.* (2008) cita algumas possíveis métricas: satisfação do cliente ao implementar o requisito, confiabilidade, segurança, custo, risco associado ao requisito, entre outras.

Após serem deifinidas as métricas dos requisitos (também chamadas de *fitness functions*), é aplicado um algoritmo de busca no espaço de possíveis soluções para encontrar a solução ótima, que contém o conjunto de requisitos que deverão ser implementados para a próxima *release* do *software* em desenvolvimento (Harman *et al*., 2012).

Dois dos algoritmos de busca comumente utilizados pela SBSE são os algoritmos *Hill Climbing* e Genético. O algoritmo *Hill Climbing* é um algoritmo de busca dito buscador local que compara a possível solução com as suas vizinhas, em busca de uma solução melhor que a atual e repete esse processo até encontrar uma solução ótima (Harman; Mansouri; Zhang, 2009). É um algoritmo de fácil implementação, que consome poucos recursos computacionais, porém deve ser rodado várias vezes para que possa encontrar a solução ótima global.

Algoritmo Genético é dito buscador global, já que analisa diferentes pontos do espaço de busca ao mesmo tempo, provendo uma robustez maior que o algoritmo *Hill Climbing*. A ideia é analisar diferentes pontos do espaço de busca, selecionar as melhores soluções de acordo com as métricas desejadas, produzir uma nova geração de soluções a partir das selecionadas como melhores soluções anteriormente, e compará-las novamente, nesse *loop* de execução. Este *loop* será executado um número determinado de vezes, a fim de encontar a solução ótima global.

O uso da técnica SBSE para priorização de requisitos tem vantagens, segundo Harman *et al*. (2012). Pode-se citar algumas: é uma técnica genérica, que pode ser aplicada de forma direta no *software* em desenvolvimento, é robusta, envolve todos os interessados no desenvolvimento do projeto, entre outras.

Portanto, SBSE propõe representar o problema de priorização de requisitos como um problema de busca, definir *fitness functions*, que são os parâmetros que decidirão os requisitos prioritários e aplicar um algoritmo de busca (neste trabalho, serão utilizados os algoritmos *Hill Climbing* e Genético) no espaço das possíveis soluções para encontrar os

requisitos que deverão ser implementados para a próxima *release* do *software*, gerando um maior valor agregado ao *software* em desenvolvimento.

Trabalhos relacionados propõem utilizar outros algoritmos de busca, como o *Simulated Annealing* (Harman *et al*., 2012) para comparar o desempenho com os algoritmos *Hill Climbing* Genético. Outro fator de recente estudo é considerar que os requisitos possuem dependências entre si e este fator deve ser incluído como métrica ao aplicar SBSE para priorizar requisitos. Zhang; Harman (2010) citam algumas dependências entre requisitos, tais como: relação de valor, relação de custo, precedência e requisitos conflitantes, onde apenas um poderá ser implementado.

O objetivo deste trabalho é utilizar diferentes técnicas de busca multi-heurísticas (*Hill Climbing*, Algoritmo Genético) a fim de realizar a priorização de requisitos de um *software* com base em um conjunto de critérios ou métricas. Por fim, é proposta uma ferramenta para apoiar o uso da abordagem de priorização dos requisitos.

O Gerente de Projetos deve tomar várias decisões ao longo do desenvolvimento de um *software*. Uma dessas decisões envolve escolher quais requisitos serão implementados na próxima *release* do *software*, com o principal objetivo de maximizar a satisfação do cliente, minimizando os custos de desenvolvimento. Para auxiliá-lo a tomar tal decisão, a técnica SBSE propõe que o problema de priorizar requisitos seja formulado como um problema de busca. Ao aplicar um algoritmo de busca no espaço de possíveis soluções, o algoritmo irá produzir como saída os requisitos priorizados agregando mais valor ao produto.

Este trabalho está estruturado da seguinte maneira: o Capítulo 2 aprofunda o conhecimento sobre Requisitos de *software* e apresenta algumas técnicas para priorização de requisitos existentes na literatura. O Capítulo 3 introduz a técnica *Search-Based Software Enginnering*, enumera características e vantagens do uso da técnica. O Capítulo 4 apresenta como utilizar a técnica SBSE para priorização de requisitos. O Capítulo 5 apresenta a metodologia adotada no trabalho. O Capítulo 6 apresenta trabalhos relacionados a este projeto. O Capítulo 7 descreve a proposta de implementação da ferramenta, detalha as suas funções e como ela produz os requisitos priorizados, além de mostrar testes e resultados obtido com o uso da ferramenta. O Capítulo 8 traz a conclusão deste trabalho e possibilidades de trabalhos futuros envolvendo SBSE aplicado a priorização de requisitos.

2 REQUISITOS DE *SOFTWARE*

Segundo Sommerville (2009), requisitos de *software* são objetivos ou restrições estabelecidas por clientes e usuários do sistema que definem as diversas propriedades do sistema. Requisitos representam a capacidade necessária que o *software* deve possuir para que o usuário possa resolver um problema ou atingir um objetivo.

Neste contexto, Pressman; Maxim (2016) definem **Engenharia de Requisitos** como sendo o amplo espectro de tarefas e técnicas que levam a um entendimento dos requisitos. Ou seja, a engenharia de requisitos estabelece uma base sólida entre o projeto de *software* e sua construção. Sem ela, o *software* resultante tem grande probabilidade de não atender às necessidades do cliente.

Assim, o conjunto de requisitos de um *software* representa um acordo negociado entre todas as partes envolvidas no desenvolvimento do sistema.

Pressman; Maxim (2016) definem sete tarefas distintas que fazem parte da Engenharia de Requisitos. São elas:

a) **Concepção.** Trata-se da etapa inicial de um projeto de *software*. Nesta etapa, identifica-se a necessidade de um negócio ou é descoberto um novo serviço ou mercado potencial. É definido um plano de negócio para a ideia, tenta-se identificar o tamanho do mercado atingido, é feita uma análise de viabilidade e identifica-se uma descrição operacional da abrangência do projeto.

b) **Levantamento de requisitos.** É nesta etapa que é feito o contato com o cliente, possíveis usuários e demais envolvidos para se estabelecer os objetivos do sistema a ser desenvolvido. É uma etapa muito importante para o desenvolvimento do projeto, pois se requisitos forem levantados de forma equivocada, todo o desenvolvimento pode ser afetado negativamente.

c) **Elaboração.** As informações obtidas pela etapa anterior são refinadas na etapa de elaboração. É gerado um modelo de requisitos refinado, que identifica funções, comportamento e informações a respeito do *software* a ser desenvolvido.

d) **Negociação.** É comum clientes solicitarem mais do que é possível fazer em um projeto, dado recursos limitados do negócio. Além disso, diferentes clientes podem propor requisitos conflitantes. É nesta etapa que precisa-se conciliar esses conflitos e chegar a um acordo sobre quais requisitos terão prioridade maior no projeto.

e) **Especificação.** Na etapa de especificação, é gerado um documento que expõe os requisitos que serão implementados no sistema.

f) **Validação.** A etapa de validação examina os documentos gerados na etapa de especificação para garantir que os requisitos do *software* tenham sido declarados de forma não-ambígua, que as inconsistências, erros e omissões tenham sido detectados e corrigidos, que requisitos não sejam muito vagos, entre outros.

g) **Gestão de requisitos.** Em um projeto de *software*, os requisitos podem sofrer alterações, as necessidades do cliente podem mudar no decorrer do projeto. A etapa de gestão encarrega-se de identificar, controlar e acompanhar

> essas possíveis mudanças que podem ocorrer em projeto de desenvolvimento. Também é tarefa da gestão de requisitos a priorização dos requisitos a serem implementados para a próxima *release* do *software*, a fim de maximizar a satisfação do cliente, minimizando os custos de produção.

Após os requisitos serem devidamente coletados, gera-se um documento contendo todos os requisitos acordados entre cliente e desenvolvedores. Esses requisitos devem ser classificados de acordo com algum aspecto, a fim de medir a sua importância ao sistema como um todo. Para isso, podem ser utilizadas propriedades e métricas. Zhang *et al.* (2008) propõem algumas, tais como:

a) **Custo:** custo para implementar e incluir o requisito ao *software*, pode ser medido em dinheiro gasto, recursos computacionais gastos, etc.;

b) **Confiabilidade:** probabilidade de falha. Tempo médio para falhar;

c) **Satisfação do cliente**: quanto a implementação desse requisito irá satisfazer o cliente;

d) **Tempo para implementação:** tempo necessário para que o requisito seja implementado e incorporado ao sistema já existente.

Considerando tempo e custos limitados para o desenvolvimento do sistema ou para o lançamento da próxima *release* (versão prévia funcional do *software*, com alguns requisitos básicos implementados, que permitam a utilização de algumas funcionalidades do produto final), necessita-se priorizar os requisitos e implementar primeiro aqueles que são considerados mais importantes para o sistema e que não tenham um custo muito alto para o desenvolvimento e incorporação ao *software* já existente. Para isso, existem alguma técnicas propostas na literatura para priorização de requisitos. Veremos duas delas a seguir.

2.1 TÉCNICAS PARA PRIORIZAÇÃO DE REQUISITOS

Existem algumas técnicas que propõem a priorização de requisitos em um projeto de *software* e que são bem aceitas, produzindo bons resultados. Dentre essas técnicas destacam-se: a técnica Moscow e a técnica de priorização baseada em valor, custo e risco.

2.1.1 A Técnica Moscow

Proposta por Clegg; Barker (1994), a técnica Moscow tem como base a classificação dos requisitos em uma das seguintes letras: M, S, C ou W, onde cada uma delas tem um significado e uma importância diferente:

a) **M - *Must have*.** Requisitos que são críticos para o sistema. São itens de extrema importância ao projeto que se não forem entregues, o projeto não pode ser conside- rado concluído com sucesso. Os requisitos classificados como M são os requisitos mais importantes para o projeto.

b) **S - *Should have*.** Requisitos classificados como S são importantes, mas não são necessários para entrega neste momento. São itens que têm importância, mas não são críticos ao sistema.

c) **C - *Could have*.** Requisitos classificados como C são desejáveis, mas não são necessários para o sistema. São itens que podem melhorar a satisfação do cliente e só devem ser atendidos se houver tempo e recursos disponíveis.

d) **W - *Won't have*.** Requisitos classificados como W são os menos críticos, que darão o menor retorno sobre o investimento. Os envolvidos no desenvolvimento do projeto concordam que esses requisitos não devem ser implementados.

Ao adotar esse método de classificação, fica claro quais são os requisitos mais importantes e que devem ser implementados primeiro.

Algumas vantagens da técnica Moscow: utiliza uma linguagem simples e de fácil compreensão de todos os envolvidos no projeto, é um método fácil de trabalhar e envolve todos os envolvidos do projeto.

Como desvantagem, cita-se a subjetividade no momento de classificar os requisitos, pois não há um critério para definir a classificação correta de cada requisito.

Essa técnica é muito utilizada para priorização de escopo, priorização de requisitos, classificação de mudanças, etc. O PRINCE2 (*Project in Controled Enviroment*, em seu livro Gerenciando Projetos de Sucesso com PRINCE2), recomenda o uso dessa técnica em várias fases de um projeto de desenvolvimento de *software* (COMMERCE, 2011).

2.1.2 Priorização Baseada em Valor, Custo e Risco

Proposta por Wiegers (1999), a técnica de Priorização Baseada em Valor, Custo e Risco utiliza os critérios de valor, custo e risco como métricas para priorizar requisitos. Cada requisito será avaliado de acordo com esses três critérios, da seguinte maneira:

a) **Valor.** Refere-se ao benefício que cada requisito agrega ao cliente, ao ser implementado. O valor varia em uma escala de 1 a 9, onde 1 significa o mais baixo benefício e 9, o mais alto benefício.

b) **Custo.** Refere-se ao custo estimado pelos desenvolvedores para implementar o requisito. Varia em uma escala de 1 a 9, onde 1 significa o mais baixo custo e 9, o mais alto custo.

c) **Risco.** Refere-se ao risco estimado pelos desenvolvedores que a implementação do requisito pode representar ao projeto. Também varia em uma escala de 1 a 9, onde 1 significa o mais baixo risco e 9, o mais alto risco.

Essas métricas de valor, custo e risco são colocadas em uma tabela e é aplicada uma fórmula matemática que irá resultar na sua prioridade. Valores mais altos representam uma prioridade maior e correspondem aos requisitos que deverão ser implementados primeiro.

3 SEARCH-BASED SOFTWARE ENGINEERING

Search-Based Software Engineering ou SBSE é o nome dado para uma área de pesquisa de Otimização de Busca para a área de Engenharia de *Software*. SBSE é uma área de pesquisa, que tem mostrado boas aplicações em Engenharia de *Software*, em subáreas como requisitos, *design* e teste (HARMAN et al., 2012).

O objetivo de SBSE é formular problemas de Engenharia de *Software* como problemas de busca, onde possam ser utilizados algoritmos de busca, como *Hill Climbing* ou Algoritmo Genético, e métricas para buscar a solução ótima para o problema.

O termo *Search-Based Software Engineering* foi utilizado pela primeira vez por Harman; Jones (2001), no primeiro artigo publicado a utilizar Otimização de Busca como uma aproximação a Engenharia de *Software*.

Como mostrado por Harman *et al*. (2012), algumas questões envolvendo Engenharia de *Software* são feitas em uma linguagem que pede uma solução por Otimização de Busca, portanto podem ser resolvidas usando SBSE. Alguns exemplos:

a) Qual é o menor conjunto de testes que cobre a totalidade de funções desse *software*?

b) Qual é o conjunto de requisitos que apresenta o melhor balanço entre custos de desenvolvimento e satisfação do cliente?

c) Qual é a melhor maneira de alocar recursos para este projeto de desenvolvimento?

Segundo Harman *et al*. (2012), algumas características da SBSE são:

a) **É uma técnica genérica.** Podemos aplicar as técnicas SBSE para vários tipos de problemas, utilizando apenas duas definições: a representação do problema e as *fitness functions*, que representam o objetivo a ser otimizado.

b) **Robustez.** Os algoritmos de otimização SBSE são robustos, ou seja, ao escolher parâmetros para a busca, ele encontrará resultados mais satisfatórios do que se fosse utilizado uma busca puramente aleatória.

c) **Reunificação.** SBSE pode criar ligações entre áreas da Engenharia de *Software* que aparentemente não possuem ligações. Por exemplo, problemas envolvendo Requisitos e Testes podem aparentar ser tópicos sem relação. Mas utilizando as técnicas SBSE, percebe-se que são dois problemas de otimização altamente semelhantes: buscamos priorizar os requisitos para maximizar a satisfação do cliente ao menor custo e buscamos priorizar os testes, a fim de testar o máximo do software, ao menor esforço.

d) **Aplicação direta.** Em outras áreas da engenharia, como mecânica, química e elétrica, também são utilizadas técnicas envolvendo Otimização de Busca. Porém nessas áreas, é necessário a construção de modelos que simulem o artefato a ser otimizado, o que torna esse processo lento e caro. Por outro lado, *softwares* não tem existência física, são modelos virtuais, o que permite a aplicação das técnicas de otimização de forma direta e rápida.

e) **Envolvimento de todos os interessados no projeto.** A fase de classificação das métricas é uma fase subjetiva e deve contar com todos os envolvidos no projeto para que a classificação seja bem feita e que todos os envolvidos aceitem as classificações propostas.

A técnica proposta em SBSE envolve basicamente três passos:

a) Representar o problema;

b) Definir as *fitness functions*;

c) Aplicar o algoritmo de busca escolhido para gerar os resultados.

O primeiro passo é representar o problema, utilizando algumas métricas para que o algoritmo de busca possa buscar a solução ótima. Após, define-se *fitness function*, que é o parâmetro a ser otimizado. Por fim, aplica-se um algoritmo de busca para gerar o resultado com a otimização buscada. Neste trabalho, serão aplicados os algoritmos *Hill Climbing* e Genético. Veremos nas próximas seções algumas características dos algoritmos *Hill Climbing* e Genético e motivos por suas escolhas para utilização na ferramenta proposta por este trabalho.

3.1 O ALGORITMO *HILL CLIMBING*

O algoritmo mais simples para se utilizar as *fitness functions* é o *Hill Climbing*. O *Hill Climbing* começa escolhendo um ponto no espaço de busca de forma aleatória. Então, ele procura na vizinhança do ponto escolhido por soluções melhores. Vizinhança se refere aos pontos muito parecidos com o primeiro ponto, mas que diferem em algum aspecto. Se o algoritmo achar alguma solução na vizinhança que seja melhor que a atual escolhida, a busca se move para essa nova solução. Ele explora essa nova vizinhança em busca de melhores soluções. O algoritmo continua executando esse *loop* até que a vizinhança da atual solução não ofereça melhorias.

Essa solução é dita localmente ótima e pode não representar a escolha globalmente ótima. Para buscar a solução globalmente ótima, o algoritmo *Hill Climbing* pode ser executado várias vezes, quantas vezes os recursos computacionais permitirem.

A Figura 3.1 mostra o pseudocódigo do funcionamento do algoritmo *Hill Climbing*, onde S é o espaço das soluções, N(s) é o conjunto dos vizinhos de s e *fit* é a *fitness function*, a função a ser maximizada.

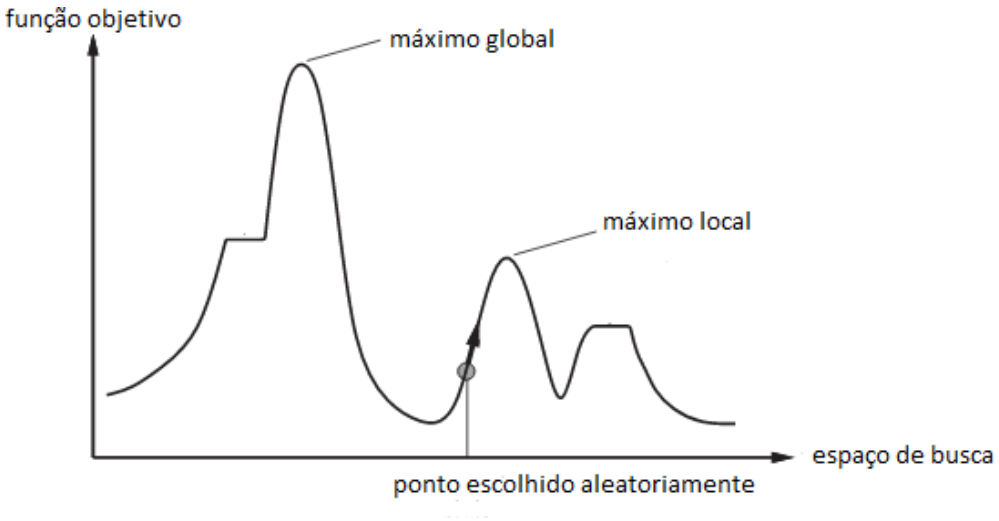
Na Figura 3.2, é possível ver a diferença entre o localmente ótimo e o globalmente ótimo, onde um ponto escolhido aleatoriamente pode chegar em um máximo local, que pode não ser o máximo global. Note que a solução encontrada nesse exemplo não é uma solução globalmente ótima. Por isso, é importante executar o algoritmo *Hill Climbing* múltiplas vezes.

Figura 3.1 – Pseudocódigo do algoritmo *Hill Climbing*

```
   Entrada: S
   Saída: Solução local ótima
1  início
2      escolha aleatoriamente uma solução s ∈ S
3      repita
4          selecione s' ∈ N(s) tal que fit(s') > fit(s)
5          s ← s'
6      até fit(s) ≥ fit(s'), para todos s' ∈ N(s);
7  fim
```

Fonte: Adaptado de Harman *et al*. (2012)

Figura 3.2 – Funcionamento do algoritmo *Hill Climbing*



Fonte: Adaptado de Castro; Zuben (2004)

O algoritmo *Hill Climbing* foi um dos algoritmos escolhidos para serem implementados na ferramenta proposta por este trabalho por ser um algoritmo simples, que consome poucos recursos computacionais, mas ainda assim gera resultados satisfatórios e confiáveis para priorização de requisitos.

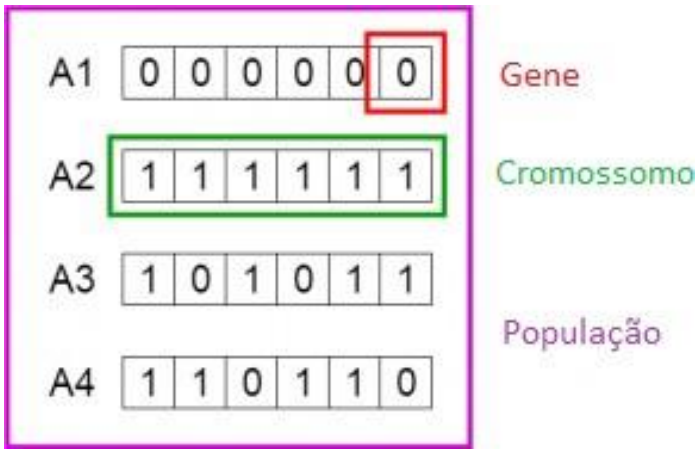
3.2 ALGORITMO GENÉTICO

Ao contrário do algoritmo *Hill Climbing*, Algoritmo Genético é dito buscador global. Ele analisa vários pontos do espaço de busca ao mesmo tempo, provendo uma robustez maior. O conjunto de soluções iniciais, escolhidas aleatoriamente, é denominado *população* e cada população sucessiva é denominada *geração*. Algoritmo Genético é inspirado

na Teoria Evolutiva de Darwin, e, para manter essa analogia, cada solução candidata é representada como vetor de componentes, denominados *indivíduos* ou *cromossomos*. A Figura 3.3 ilustra os conceitos de gene, cromossomo e população.

Geralmente, Algoritmo Genético utiliza um vetor, denominado vetor de decisão, representado de forma binária, ou seja, as soluções candidatas são codificadas para cadeias de 0s e 1s, onde 1 representa a presença de determinada característica e 0, a ausência de determinada característica. No caso deste trabalho, 1 irá representar um requisito que será implementado na próxima *release* e 0 representa um requisito que não será implementado na próxima *release*.

Figura 3.3 – Gene, cromossomo e população em Algoritmo Genético



Fonte: Adaptado de Stroski (2018)

O *loop* principal do Algoritmo Genético pode ser visto na Figura 3.4.

Figura 3.4 – Pseudocódigo do Algoritmo Genético

```
1  início
2      escolha aleatoriamente uma população inicial P
3      repita
4          avalie a fitness function de cada indivíduo de P
5          selecione pais oriundos de P de acordo com o mecanismo de seleção
6          faça o cruzamento dos pais para formar novos filhos
7          construa uma nova população P oriundos dos pais e filhos previamente
           calculados
8          aplique mutação em P
9      até condição de parada for atingida;
10 fim
```

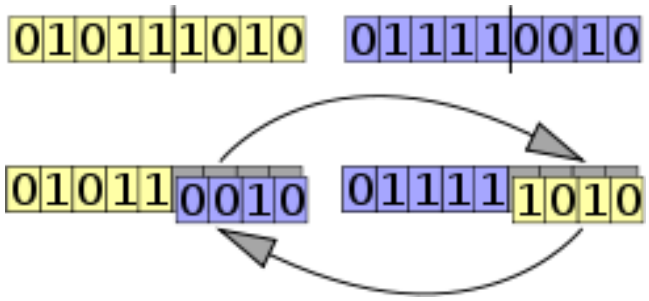
Fonte: Adaptado de Harman *et al*. (2012)

A população inicial é selecionada aleatoriamente no espaço das soluções candidatas. Ao avaliar a *fitness function* para cada indivíduo da população, apenas os indivíduos

mais aptos são selecionados para irem para os estágios de cruzamento, mutação e inserção na nova geração.

No estágio de cruzamento, elementos de cada indivíduo são recombinados para gerar dois novos filhos. A ideia é trocar informações entre dois pais para gerar filhos diferentes dos pais, para eles também serem avaliados na nova geração. Uma das maneiras de se realizar o cruzamento é a Recombinação em um Ponto, onde um único ponto para recombinação é selecionado em ambos os pais. Todos os dados além do ponto selecionado são trocados entre os pais. Os indivíduos resultantes são os filhos. A Recombinação em um Ponto pode ser vista na Figura 3.5.

Figura 3.5 – Recombinação em um Ponto em um Algoritmo Genético

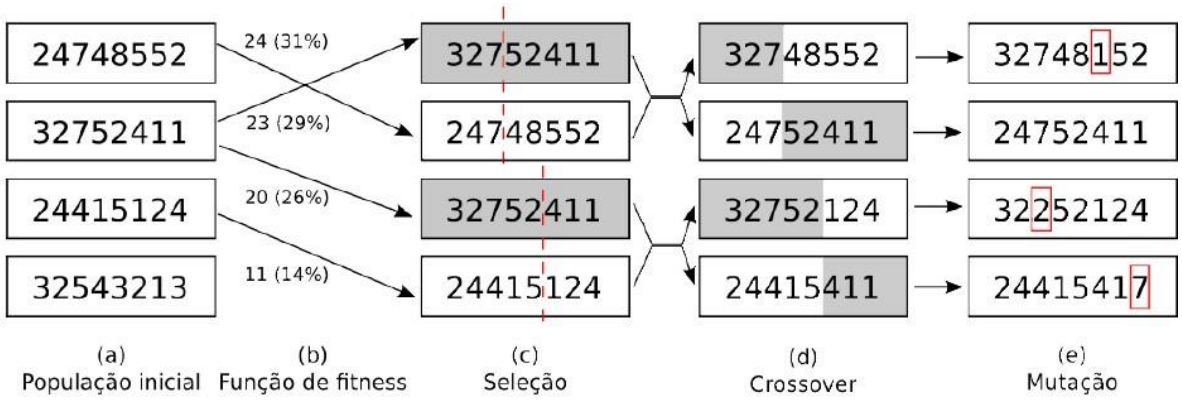


Fonte: Adaptado de Wikipedia (2018).

Após o estágio de recombinação, é aplicado mutação sobre os indivíduos da nova geração. Uma mutação é uma probabilidade de trocar 1 bit de cada indivíduo da população. O principal objetivo da mutação é diversificar a área de busca, a fim de evitar o problema de o algoritmo ficar trancado em algum máximo local e impedido de localizar o máximo global.

Esse *loop* do algoritmo genético repete até que alguma condição de parada seja alcançada, por exemplo, o número de gerações avaliadas. A Figura 3.6 ilustra as etapas do *loop* principal do algoritmo genético.

Figura 3.6 – Etapas do *loop* principal do Algoritmo Genético



Fonte: Nria; Salomão; Olivette (2013)

O próximo capítulo irá mostrar como aplicar as técnicas SBSE para priorização de requisitos em um projeto de *software*.

4 *SEARCH-BASED SOFTWARE ENGINEERING* APLICADO A GERÊNCIA DE REQUISITOS

Para utilizarmos as técnicas SBSE para priorização de requisitos, Harman *et al*. (2009) propõe considerarmos o Problema da Próxima *Release*. Este problema analisa quais requisitos podem ser implementados para maximizar a satisfação do cliente, enquanto minimizam-se os custos. O conjunto de requisitos que podem ser implementados é representado por um vetor de requisitos R, definido da seguinte maneira:

R = $\{r_1, r_2, ..., r_n\}$, onde cada r_i representa um requisito

São considerados dois parâmetros para avaliar cada requisito: o valor agregado que a implementação do requisito dará ao projeto (representada pela variável "valor") e o custo para implementar o requisito. Consideramos os seguintes conjuntos:

Custo = $\{custo_1, custo_2, ..., custo_n\}$ e

Valor = $\{valor_1, valor_2, ..., valor_n\}$, onde cada $custo_i$ e $valor_i$ representam o custo e o valor agregado do requisito *i*

Por serem classificadas de maneira subjetiva, as métricas valor e custo devem ser definidas por todos os participantes do projeto antes da aplicação da técnica SBSE, para garantir que todos os envolvidos no projeto concordem com os valores das métricas antes da aplicação da técnica.

Além disso, representamos o vetor de decisão como:

$$\dot{X} = < x_1, x_2, ..., x_n >$$

Nesse vetor, o *i*-ésimo elemento de $\dot{X}$ será 1 se o *i*-ésimo requisito será implementado e será 0 se o *i* -ésimo requisito não será implementado. Dado uma instância do vetor de decisão, $\dot{X}_1$, chamaremos de F($\dot{X}_1$) a *fitness function* que avalia o valor agregado dos requisitos a serem implementados por $\dot{X}_1$:

$$F(X_1) = \sum_{i=1}^{n} \text{valor}i \times \text{custo}i$$

De maneira semelhante, a função custo será:

$$custo(X_1) = \sum_{i=1}^{n} custoi \times xi$$

Para este trabalho, foi considerado um valor máximo para o custo, onde os requisitos implementados não poderão ultrapassar esse limite máximo para o custo, ou seja, qualquer vetor de decisão que considere requisito que, somados ultrapassem o custo limite, será descartado como possível solução. Então, ao aplicarmos a técnica SBSE queremos:

Maximizar $\sum_{i=1}^{n} \text{valor}i \times \text{x}i$ e garantir que $\sum_{i=1}^{n} custoi \times xi$ não ultrapasse o limite do custo

Portanto, o algoritmo de busca computa sobre os parâmetros **valor** e **custo** de cada requisito e gera como saída um vetor de decisão no formato de $\dot{X}$, onde o requisito r_i terá valor 1 se será implementado nesta *release* e 0 caso contrário. Inicialmente iremos considerar que não há qualquer dependência entre os requisitos.

5 TRABALHOS RELACIONADOS

Existem outros trabalhos propostos que têm como objetivo a priorização de requisitos em desenvolvimento de projeto de *software*. Neste capítulo, são citados alguns desses trabalhos e algumas de suas características.

5.1 SIMULATED ANNEALING

Harman *et al*. (2012) propõem o uso do algoritmo de busca *Simulated Annealing*, ou Recozimento Simulado, como um dos algoritmos de busca capazes de selecionar os requisitos com maior prioridade a serem implementados para a próxima *release*.

O funcionamento do *Simulated Annealing* tem como base uma técnica de busca local probabilística: uma solução é escolhida de forma aleatória no começo da execução do algoritmo, e essa solução atual é substituída por uma solução da vizinhança com base em uma *fitness function* e uma variável **T**. Quanto maior for a variável **T**, maior será a componente aleatória incluída na próxima solução escolhida. O uso da variável **T** permite que o algoritmo explore novos locais do espaço de possíveis soluções. A cada *loop* do algoritmo, o valor de **T** decrementa, o que faz o algoritmo convergir para uma solução ótima.

Simulated Annealing tem um funcionamento parecido com o algoritmo *Hill Climbing*, mas, com o uso da variável **T**, ele pode convergir mais facilmente para a solução global ótima, enquanto que o algoritmo *Hill Climbing* necessita de várias execuções para chegar na solução global ótima. Porém, o *Hill Climbing* é um algoritmo de mais fácil implementação, por não utilizar variáveis extras, consome menos recursos computacionais em sua execução e ainda assim gera resultados satisfatórios.

5.2 SBSE MULTI-OBEJTIVOS

A dependência entre requisitos é um importante elemento que direciona a escolha sobre quais requisitos serão implementados para uma próxima *release* do *software*.

De acordo com Carlshamre *et al*. (2001), a tarefa de definir uma seleção ótima de requisitos a serem implementados para a próxima *release* de um *software* é difícil, já que requisitos podem depender uns dos outros de maneiras complexas. A dependência entre requisitos ainda é um assunto pouco explorado quando considera-se a priorização de requisitos.

Pensando nisso, Zhang; Harman (2010) propuseram uma variação da SBSE que considera múltiplos objetivos a serem otimizados e inclui as dependências entre os requisitos do projeto. É considerado algumas possíveis relações entre requisitos:

a) **E.** Se um requisito R_1 for escolhido para ser implementado, então um requisito R_2 terá que ser implementado também.

b) **Ou exclusivo.** Os requisitos R_1 e R_2 são conflitantes e apenas um deles poderá ser implementado.

c) **Precedência.** Um requisito R_1 deverá ser implementado antes de um requisito R_2.

d) **Relação de valor.** Se um requisito R_1 é escolhido para ser implementado, isso irá afetar o valor do requisito R_2 para o cliente.

e) **Relação de custo.** Se um requisito R_1 é escolhido para ser implementado, isso irá afetar o custo para implementar o requisito R_2.

Essas relações são codificadas para então serem utilizadas como critério de escolha, junto das *fitness functions*. Este trabalho utiliza o algoritmo de busca NSGA-II (*Non-dominated Sorting Genetic Algorithm*), proposto por Deb *et al*. (2002), como principal algoritmo para realizar a busca no espaço das possíveis soluções.

O algoritmo NSGA-II basicamente é um algoritmo genético multi-objetivo e seu funcionamento é semelhante aos algoritmos genéticos de um objetivo já vistos na subseção 3.2.2, mas aplicado a vários objetivos. Por ser um algoritmo multi-objetivos, foi a escolha preferida para ser utilizada no trabalho proposto por Zhang; Harman (2010).

5.3 ALGORITMO GULOSO

Baker *et al*. (2006) propõem o uso de algoritmo guloso para selecionar os próximos componente a serem implementados para a próxima *release* de um projeto em desenvolvimento. Cada componente possível de ser implementado possui um peso *w* associado a ele. Os componentes são ordenados em ordem crescente de pesos e é aplicado um algoritmo guloso para selecionar quais componentes irão ser implementados para a próxima *release*. Existe um custo máximo, que não pode ser ultrapassado pela soma dos pesos dos componentes. O funcionamento do algoritmo guloso é simples: ele irá percorrer a lista dos componentes e, para cada componente, irá somar seu peso ao peso total até o momento. Se essa soma não superar o limite máximo do custo, esse componente será incluído em uma lista para ser implementado na próxima *release*, caso contrário, ele não será incluído na lista e, portanto, não será implementado na próxima *release*.

Essa ideia de selecionar componentes utilizando algoritmo guloso foi testada em comparação à técnica *Simulated Annealing*, vista anteriormente. Os dois algoritmos obtiveram resultados semelhantes quanto a escore dos componente selecionados e também em número de componentes selecionados.

6 IMPLEMENTAÇÃO DA PRIORIZAÇÃO DE REQUISITOS

Neste capítulo é descrita a implementação da priorização de requisitos definida neste trabalho. Essa implementação possui como requisitos:

a) Receber como entrada uma lista de requisitos, cada requisito contendo métricas custo e valor agregado;

b) Aplicar as técnicas vistas de SBSE e utilizar os algoritmos de busca *Hill Climbing* e Genético para priorizar requisitos;

c) Produzir uma saída contendo os requisitos selecionados para serem implementados para a próxima *release*.

A implementação foi feita na linguagem Java, e foi utilizada a IDE Netbeans, versão 8.2, para a implementação.

A entrada é um arquivo texto, no formato txt, que conterá o id do requisito, o custo para implementá-lo e o valor que a implementação do requisito acrescentará ao projeto. A implementação possui uma classe Requisito, com as variáveis id, custo e valor. O id do requisito e suas métricas são transformados em um vetor de requisitos, onde cada posição do vetor de requisitos contém o id do requisito, seu valor e seu custo. O vetor de requisitos é passado para a etapa da aplicação dos algoritmos de busca.

As métricas custo e valor da solução são calculadas baseadas no vetor de decisão gerado aleatoriamente e o cálculo é feito da seguinte maneira: para cada posição *i* do vetor de decisão, se o valor da posição do vetor de decisão for igual a 1, significa que o requisito correspondente a essa posição no vetor de requisitos seria implementado na próxima *release* e, portanto, seus valores das variáveis custo e valor são somados ao custo total e ao valor total daquela solução.

Custo e valor irão definir quais requisitos serão implementados para a próxima *release* da seguinte maneira: é definido um número para o custo máximo, e qualquer solução que ultrapasse este número máximo para o custo, irá ter o seu valor para a variável valor setado em zero, pois será assumido que as soluções que ultrapassarem o número máximo para o custo não poderão ser implementadas. Para as soluções que ficarem dentro do limite do custo máximo, a solução que apresentar o maior número para a variável valor será a melhor solução, quando for comparada com outras possíveis soluções. A aplicação dessa regra pode ser visto no trecho de código da Figura 6.1.

As classes Hill e Genetico são encarregadas de aplicar, respectivamente, os algoritmos de busca*Hill Climbing* e Genético para buscarem a solução ótima de priorização dos requisitos e retornam para a função principal (*main*) do programa a resposta encontrada. Essa resposta encontrada, denominada vetor de decisão, será um vetor com o mesmo tamanho do vetor de requisitos, onde cada posição desse vetor será 0 ou 1, onde 0 indica que o requisito correspondente à essa posição no vetor de requisitos **não será** implementada na próxima *release*, enquanto que 1 indica que o requisito correspondente à essa posição no vetor de requisitos **será** implementada na próxima *release*. Por exemplo, se o vetor resposta for {1, 0, 0, 1}, significa que os requisitos que ocupam as posições 1 e 4 do vetor de requisitos serão implementados, enquanto que os requisitos que ocupam as posições 2 e 3 do vetor de requisitos não serão implementados. O Diagrama de Classes da ferramenta pode ser visto na Figura 6.2.

Veremos um passo a passo detalhado da execução da ferramenta:

Figura 6.1 – Cálculo das variáveis custo e valor e teste da regra se o custo da atual solução não ultrapassa o custo máximo

```
//calcula custo e valor atual
for (i = 0; i < n; i++){
    if(rand[i] == 1){
        custo += req[i].custo;
        valor += req[i].valor;
    }
}

if (custo > customax){
    valor = 0;
}
```

Fonte: Criado pelo autor

a) Inicializa o vetor de requisitos;

b) Lê arquivo txt e preenche o vetor de requisitos com id, valor e custo de cada requisito;

c) Chama o algoritmo *Hill Climbing*, que irá retornar a resposta ótima encontrada pelo algoritmo;

d) Chama o Algoritmo Genético, que irá retornar a resposta ótima encontrada pelo algoritmo;

e) Compara as respostas encontradas pelos algoritmos *Hill Climbing* e Genético e escolhe a melhor resposta como resposta final;

f) Cria um arquivo de saída contendo os IDs dos requisitos que deverão ser implementados para a próxima *release*.

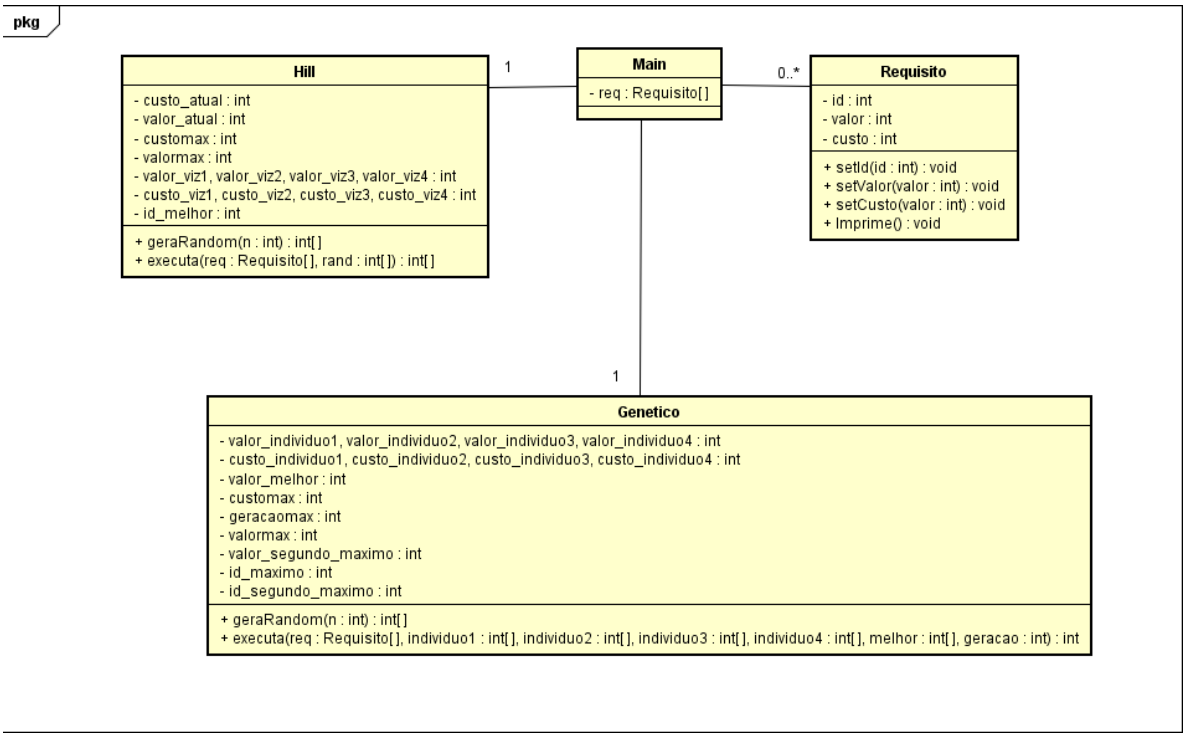
Nas próximas seções são descritos alguns detalhes do funcionamento dos algoritmos de busca *Hill Climbing* e Genético.

6.1 HILL CLIMBING

O primeiro passo do algoritmo *Hill Climbing* é gerar uma solução aleatória, que será onde o algoritmo irá começar as comparações com os seus vizinhos. A função que gera a solução aleatória pode ser vista na Figura 6.3.

Após gerada a solução aleatória, é chamada a função principal do algoritmo *Hill Climbing*, que inicialmente irá calcular o número para as variáveis valor e custo, considerando a regra anteriormente vista de que se a solução apresentar custo maior que o custo máximo, sua variável valor será setada para zero.

Figura 6.2 – Diagrama de Classes da Ferramenta



Fonte: Criado pelo autor

O próximo passo é gerar os vizinhos da atual solução. A ferramenta gera quatro vizinhos: dois antecessores e dois sucessores da solução atual. Para definir os antecessores e sucessores, considera-se o vetor de decisão como sendo um número binário e os antecessores são definidos subtraindo-se 1 e 2 do valor correspondente ao vetor de decisão. Os sucessores são definidos de maneira semelhante, somando-se 1 e 2 ao valor correspondente ao vetor de decisão. Um exemplo do cálculo dos vizinhos pode ser visto no Quadro 6.1.

Quadro 6.1 – Cálculo dos vizinhos do *Hill Climbing*

Vizinho -2	Vizinho -1	Solução Atual	Vizinho +1	Vizinho +2
1001 - 2 = 0111	1001 - 1 = 1000	1001	1001 + 1 = 1010	1001 + 2 = 1011

Fonte: Criado pelo autor.

Após a geração dos vizinhos, é calculado, para cada vizinho, seu custo e valor, seguindo a regra do custo máximo anteriormente discutida. Após calculados custo e valor para cada vizinho, são feitas comparações entre os quatro vizinhos e a solução atual para definir qual destas cinco possíveis soluções tem valor maior. Se um dos vizinhos tiver valor maior, a função principal do *Hill Climbing* é chamada recursivamente, passando como parâmetro esse vizinho com o maior número para a variável valor, que será a solução atual da próxima chamada. A função principal do algoritmo *Hill Climbing* é, então, executada mais uma vez. Esse *loop* será repetido até que os vizinhos da solução atual não apresentem melhora na variável valor em relação a solução atual, assim encerrando a execução do *Hill Climbing* e retornando para a função *main* o vetor de decisão correspondente a solução

Figura 6.3 – Função para gerar uma solução aleatória para o algoritmo *Hill Climbing*

```
public int[] geraRandom(int n){
    //gerador aleatorio
    int[] rand = new int[n];
    Random gerador = new Random();
    for (int i = 0; i < n; i++){
        rand[i] = (gerador.nextInt(2));
    }
    return rand;
}
```

Fonte: Criado pelo autor

atual. Essa comparação dos vizinhos com a solução atual e a chamada recursiva do *Hill Climbing* podem ser vistos no trecho de código da Figura 6.4.

Na ferramenta desenvolvida, a execução do algoritmo *Hill climbing* está dentro de um *loop*, que tem por objetivo executar o algoritmo dez vezes, o que proporciona uma resposta melhor do que se o algoritmo fosse executado apenas uma vez. A melhor resposta dentre as dez execuções do algoritmo *Hill climbing* é gravada em um vetor resposta na função *main* da ferramenta.

6.2 ALGORITMO GENÉTICO

Para o Algoritmo Genético da ferramenta, o tamanho de cada população a ser comparada por geração é quatro indivíduos e o número de gerações que o algoritmo irá produzir é dez. Esses números foram atingidos através de testes de execução, que mostraram que uma população maior, assim como um número maior de gerações não garantem melhoras no resultado encontrado pelo algoritmo.

Assim como no algoritmo *Hill Climbing*, o primeiro passo do Algoritmo Genético é inicializar de maneira aleatória a população inicial, que será composta de quatro indivíduos. Após gerada a população inicial, é chamada a função principal do Algoritmo Genético. A função principal do Algoritmo Genético irá calcular o custo e o valor de cada indivíduo da população a ser avaliada, também seguindo a regra do custo máximo. Depois disso, são feitas comparações entre os quatro indivíduos para definir os dois melhores indivíduos da população. Esses dois melhores indivíduos irão para as próximas etapas do Algoritmo Genético.

A próxima etapa é a etapa de cruzamento: a partir dos dois indivíduos anteriormente selecionados como melhores indivíduos da população, serão gerados dois filhos. Os filhos serão gerados a partir da Recombinação em um Ponto e o ponto para recombinação é a metade do tamanho do vetor de decisão. Por exemplo, se o tamanho do vetor de decisão for dez, o primeiro filho terá em suas posições um a cinco os genes do primeiro pai e em suas posições seis a dez os genes do segundo pai. Para o segundo filho,

Figura 6.4 – Definição de qual solução entre os quatro vizinhos e a solução atual tem o maior número para a variável valor no *Hill Climbing*

```
switch(id){
    case 1:
        //vizinho+1 tem valor maior
        rand = exec.executa(req, rand1);
        break;
    case 2:
        //vizinho-1 tem valor maior
        rand = exec.executa(req, rand2);
        break;
    case 3:
        //vizinho+2 tem valor maior
        rand = exec.executa(req, rand3);
        break;
    case 4:
        //vizinho-2 tem valor maior
        rand = exec.executa(req, rand4);
        break;
    default:
        //solucao atual tem valor maior
        break;
}
```

Fonte: Criado pelo autor

invertem-se os pais: o segundo pai gera os genes da primeira metade do filho, e o primeiro pai gera os genes da segunda metade do filho. A Figura 6.5 mostra o trecho de código correspondente a etapa de cruzamento, onde os vetores aux1 e aux2 representam, respectivamente, o primeiro e o segundo pais. O Quadro 6.2 ilustra um exemplo de pais e os filhos gerados por esses pais. Na coluna dos filhos, a barra vertical (|) apenas ilustra o ponto da recombinação.

Quadro 6.2 – Exemplos de pais e seus filhos gerados pela Recombinação em um ponto

Pais	Filhos
1011001011	10110|11001
0010111001	00101|01011

Fonte: Criado pelo autor.

Os dois pais e os dois filhos gerados a partir desses dois pais serão os quatro indivíduos da próxima geração do Algoritmo Genético. Após a etapa de cruzamento, a próxima etapa do algoritmo é a etapa de mutação. Na etapa de mutação, cada gene de

Figura 6.5 – Etapa de cruzamento no Algoritmo Genético

```
//crossover
int[] filho1 = new int[n];
int[] filho2 = new int[n];
for(i = 0; i < (n / 2); i++){
    filho1[i] = aux1[i];
    filho2[i] = aux2[i];
}
for(i = (n / 2); i < n; i++){
    filho1[i] = aux2[i];
    filho2[i] = aux1[i];
}
```

Fonte: Criado pelo autor

cada indivíduo da nova população possui 10% de chance de sofrer mutação. Testes foram realizados para valores entre 5% e 15% de taxa de mutação, mas os resultados obtidos foram semelhantes e, por isso, preferiu-se utilizar o valor intermediário entre as taxas de mutação analisadas.

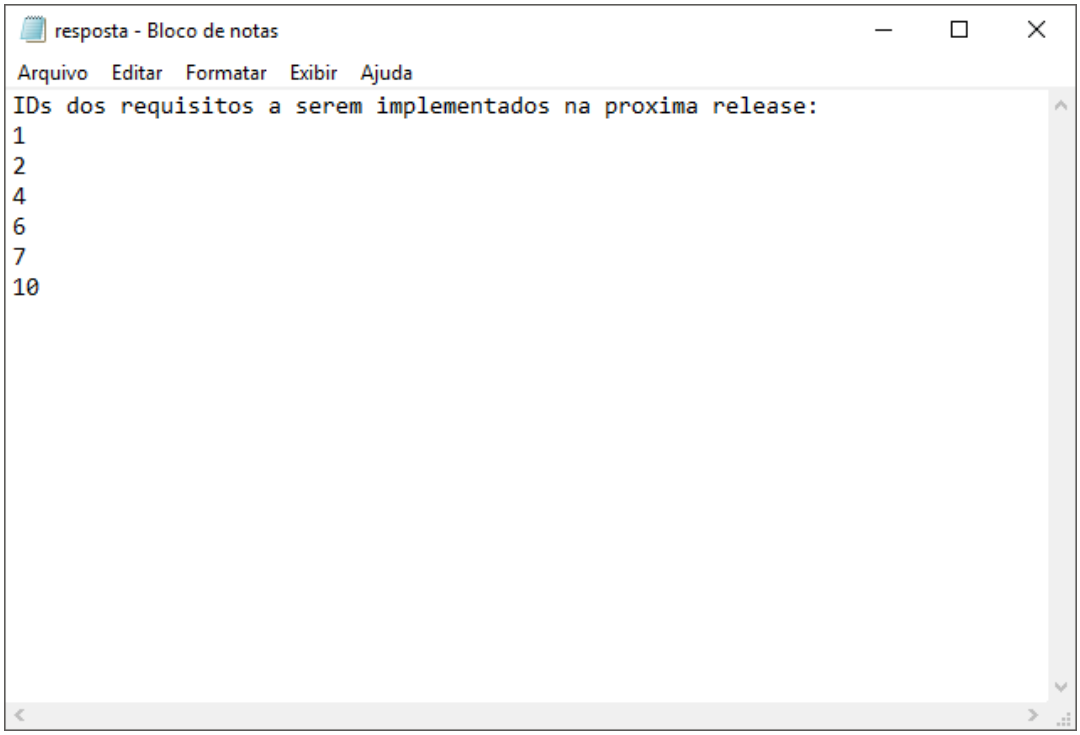
Após a etapa de mutação, é chamada recursivamente a função principal do Algoritmo Genético, passando como argumentos os indivíduos da nova população, anteriormente calculados. Esse *loop* será executado dez vezes, que é o número de gerações definidos para executar nessa ferramenta.

É importante ressaltar que é guardado em um vetor chamado de **melhor**, a melhor solução encontrada até o momento pelo Algoritmo Genético. Esse vetor se faz necessário porque a etapa de mutação pode alterar a melhor resposta encontrada e fazer com que o Algoritmo Genético retorne uma resposta pior do que a melhor resposta encontrada durante sua execução. Ao final da execução das dez gerações do Algoritmo Genético, esse vetor **melhor** é retornado para a função *main* da ferramenta.

6.3 PARTE FINAL DA EXECUÇÃO

Após executar os algoritmos *Hill Climbing* e Genético, a função *main* da ferramenta tem a melhor resposta gerada pelo *Hill Climbing* e a melhor resposta gerada pelo Algoritmo Genético. Então, é feita uma comparação entre as duas respostas para encontrar a melhor resposta geral. Essa melhor resposta geral é escrita em um arquivo texto, que conterá os ids dos requisitos que serão implementados na próxima *release*. Um exemplo de resposta gerada após a execução da ferramenta pode ser visto na Figura 6.6.

Figura 6.6 – Exemplo de resposta gerada pela ferramenta



Fonte: Criado pelo autor

6.4 TESTES E RESULTADOS

Para a realização de testes e obtenção de resultados, foram considerados dez requisitos, cujos valores de custo e valor variam entre um e quinze. Para o custo máximo, foi considerado o número cinquenta. A etapa de testes definiu os valores que foram utilizados para as seguintes variáveis:

a) O número de execuções do algoritmo *Hill Climbing*: 10;

b) O número de vizinhos analisados por vez no algoritmo *Hill Climbing*: 4;

c) O número de gerações avaliadas para o Algoritmo Genético: 10;

d) O número de indivíduos analisados por geração no Algoritmo Genético: 4;

e) A taxa de mutação de cada gene para cada indivíduo da nova geração no Algoritmo Genético: 10%.

Foram testados outros valores para essas variáveis, mas esses valores diferentes não representaram mudança significativa no resultado final obtido pelos algoritmos de busca.

Após definidos os valores finais para as variáveis anteriormente citadas, foram realizadas 50 execuções da ferramenta para a geração de resultados. A melhor solução, em termos da variável valor, é a solução {1011011011}, que apresenta 49 de valor a um custo de 46. O desempenho dos algoritmos de busca *Hill Climbing* e Genético pode ser visto no Quadro 6.3.

Quadro 6.3 – Desempenho dos Algoritmos *Hill Climbing* e Genético nos testes de validação

	Hill Climbing	**Genético**
Obteve a melhor resposta	28	22
Obteve a resposta ótima	7	5
Média de "valor"	45,15	43,2
Média de "custo"	45,05	47,2

Fonte: Criado pelo autor.

De maneira geral, o algoritmo *Hill Climbing* apresentou um desempenho melhor em relação ao Algoritmo Genético, mas os dois algoritmos de busca obtiveram resultados satisfatórios ao buscar a priorização de requisitos: 12 das 50 execuções da ferramenta levaram a resposta ótima, o que representa 24% do número de execuções. Além disso, foram obtidas outras respostas próximas a resposta ótima diversas vezes, o que mostra que se os algoritmos de busca não necessariamente irão encontrar a resposta ótima, ao menos irão encontrar uma resposta próxima da resposta ótima.

7 CONCLUSÃO E TRABALHOS FUTUROS

A técnica *Search-Based Software Engineering* propõe formular problemas de Engenharia de Software como problemas de busca, nos quais possam ser aplicados algoritmos de busca para se chegar a uma resposta ótima do problema. Uma das áreas em que se tem aplicado com sucesso a técnica SBSE é na priorização de requisitos. A priorização de requisitos tem como principal objetivo selecionar os requisitos mais importantes para o *software* em desenvolvimento para serem implementados para a próxima *release*, garantindo, assim, um maior valor agregado do produto, respeitando o limite de custo.

A proposta aprensentada neste Trabalho de Conclusão de Curso foi a implementação de uma ferramenta que recebe como entrada uma lista de requisitos, aplica a técnica SBSE para priorizar os requisitos e gera como saída os requisitos que deverão ser implementados para a próxima *release*.

Para a execução deste trabalho, antes foi realizado um estudo sobre priorização de requisitos e sobre a técnica SBSE. Foi definido quais algoritmos de busca foram utilizados para o desenvolvimento da ferramenta e quais as vantagens de se utilizar esses algoritmos.

Após a análise da literatura e de artigos disponíveis, foram obtidos trabalhos relacionados que ofereceram ideias de como utilizar a ferramenta para priorização de requisitos.

Para o desenvolvimento da ferramenta, foram utilizados os algoritmo de busca *Hill Climbing* e Genético. O *Hill Climbing* foi escolhido por ser um algoritmo simples, que utiliza poucos recursos computacionais, mas que cumpre sua tarefa de gerar soluções ótimas. O Algoritmo Genético foi escolhido por ser um algoritmo mais completo, que analisa vários pontos do espaço de busca ao mesmo tempo e chega na resposta ótima com mais facilidade.

Para validação da ferramenta, foram realizados 50 testes, onde, em cada teste, o algoritmo *Hill Climbing* foi executado 10 vezes e o Algoritmo Genético executava uma vez e gerava 10 gerações de indivíduos. Em 24% das vezes, a ferramenta alcançou a resposta ótima e em várias outras execuções, a resposta obtida foi uma resposta próxima à resposta ótima. Em termos de cada algoritmo, o algoritmo *Hill Climbing* obteve a melhor resposta 28 vezes, obtendo uma média de valor de 45,15 e uma média de custo de 45,05, alcançando a resposta ótima 7 vezes. O Algoritmo Genético obteve a melhor resposta 22 vezes, obtendo uma média de valor de 43,2 e uma média de custo de 47,2, tendo chegado na resposta ótima 5 vezes.

Com os dados dos testes, conclui-se que a ferramenta obteve sucesso em priorizar requisitos, pois em 24% das execuções, ela obteve a resposta ótima do problema, e ficou perto da resposta ótima outras diversas vezes, servindo de guia para se alcançar a resposta ótima do problema de priorização de requisitos.

Para trabalhos futuros, cita-se a inclusão de mais métricas para a avaliação dos requisitos e a inclusão da dependência entre requisitos como critério para a priorização dos requisitos.

REFERÊNCIAS BIBLIOGRÁFICAS

ACHIMUGU, P.; SELAMAT, A.; IBRAHIM, R.; MAHRIN, M. A systematic literature review of software requirements prioritization research. **Information and Software Technology.** v. 56, n. 6, p. 568 – 585, 2014. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584914000354>.

BAKER, P.; HARMAN, M.; STEINHÖFEL, K.; SKALIOTIS, A. Search based approaches to component selection and prioritization for the next release problem. In: **Proceedings of the 22Nd IEEE International Conference on Software Maintenance**. Washington, DC, USA: IEEE Computer Society, 2006. (ICSM '06), p. 176–185. ISBN 0-7695-2354-4. Disponível em: <http://dx.doi.org/10.1109/ICSM.2006.56>.

CARLSHAMRE, P.; SANDAHL, K.; LINDVALL, M.; REGNELL, B.; NATT OCH DAG, J. An industrial survey of requirements interdependencies in software product release planning. In: IEEE. **Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on**. [S.l.], 2001. p. 84–91.

CASTRO, L. de; ZUBEN, F. von. **"Hill Climbing" e "Simulated Annealing"**. Unicamp, 2004. Acessado em 14 de setembro de 2018. Disponível em: <ftp: //ftp.dca.fee.unicamp.br/pub/docs/vonzuben/ia707_02/topico1_02.pdf>.

CLEGG, D.; BARKER, R. **Case Method Fast-Track: A Rad Approach**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN 020162432X.

COMMERCE, O. of G. **Gereciando projetos de sucesso com PRINCE2**. [S.l.]: Stationery Office, 2011. ISBN 9780113313471.

DEB, K.; PRATAP, A.; AGARWAL, S.; MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. **IEEE transactions on evolutionary computation**, IEEE, v. 6, n. 2, p. 182–197, 2002.

HARMAN, M.; JONES, B. Information and software technology. **Search based software engineering**. [S.l.: s.n.], 2001. p. 833–839.

HARMAN, M.; KRINKE, J.; REN, J.; YOO, S. Search based data sensitivity analysis applied to requirement engineering. In: ACM. **Proceedings of the 11th Annual conference on Genetic and evolutionary computation**. [S.l.], 2009. p. 1681–1688.

HARMAN, M.; MANSOURI, A.; ZHANG, Y. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. **Technical Report TR-09-03**, 2009.

HARMAN, M.; MCMINN, P.; SOUZA, J.; YOO, S. Search based software engineering: Techniques, taxonomy, tutorial. In: **Empirical software engineering and verification**. [S.l.]: Springer, 2012. p. 1–59.

NRIA SALOMãO OLIVETTE ARTERO, A. . C. L. . C. C. **Roteamento de Veículos Utilizando Otimização por Colônia de Formigas e Algoritmo Genético**. Researchgate, 2013. Acessado em 20 de outubro de 2018. Disponível em: <https://www.researchgate.net/publication/300661461_Roteamento_de_Veiculos_Utilizando_Otimizacao_por_Colonia_de_Formigas_e_Algoritmo_Genetico>.

PRESSMAN, R.; MAXIM, B. Engenharia de requisitos. **Engenharia de Software: Uma Abordagem Profissional**. [S.l.]: AMGH Editora Ltda., 2016. cap. 8, p. 131–165.

SOMMERVILLE, I. Requirements engineering. **Engenharia de Software: Uma Abordagem Profissional**. [S.l.]: Pearson Education Inc., 2009. cap. 4, p. 82–117.

STROSKI, P. N. **O que é algoritmo genético?** 2018. Acessado em 20 de outubro de 2018. Disponível em: <http://www.electricalelibrary.com/2018/04/13/o-que-e-algoritmo-genetico/>.

WIEGERS, K. First things first: prioritizing requirements. **Software Development**, Prentice-Hall, v. 7, n. 9, p. 48–53, 1999.

WIKIPEDIA. **Recombinação (computação evolutiva)**. Wikipedia. Acessado em 15 de setembro de 2018. Disponível em: <https://pt.wikipedia.org/wiki/Recombina\%C3\%A7\%C3\%A3o_(computa\%C3\%A7\%C3\%A3o_evolutiva)>.

ZHANG, Y.; FINKELSTEIN, A.; HARMAN, M. Search based requirements optimisation: Existing work and challenges. In: SPRINGER. **International Working Conference on Requirements Engineering: Foundation for Software Quality**. [S.l.], 2008. p. 88–94.

ZHANG, Y.; HARMAN, M. Search based optimization of requirements interaction management. In: IEEE. **Search Based Software Engineering (SSBSE), 2010 Second International Symposium on**. [S.l.], 2010. p. 47–56.