PROBLEMA DO PRÓXIMO RELEASE INTEGRADO À GERÊNCIA DE PROJETOS ATRAVÉS DE METODOLOGIAS ÁGEIS

IVAN ÍTALO ITUASSÚ

PROBLEMA DO PRÓXIMO RELEASE INTEGRADO À GERÊNCIA DE PROJETOS ATRAVÉS DE METODOLOGIAS ÁGEIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Geraldo Robson Mateus

Belo Horizonte

Junho de 2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Ituassú, Ivan Ítalo.

I91p Problema do próximo release integrado à gerência de projetos através de metodologias ágeis. / Ivan Ítalo Ituassú. –Belo Horizonte, 2017.
xxii, 71 f.: il.; 29 cm.

Dissertação (mestrado) – Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Geraldo Robson Mateus.

1. Computação – Teses. 2. Engenharia de software. 3. Otimização em engenharia de software. 4. Problema do próximo release. I. Orientador. II. Título.

CDU 519.6*61(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Problema do próximo release integrado à gerência de projetos através de metodologias ágeis

IVAN ÍTALO ITUASSÚ

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. GERALDO ROBSON MATEUS - Orientador
Departamento de Ciência da Computação - UFMG

PROF. CLARINDO ISAÍAS P. DA SILVA E PÁDUA
Departamento de Ciência da Computação - UFMG

PROFA. DANIELA CRISTINA CASCINI KUPSCH
Departamento de Ciência da Computação - CEFETMG

PROF. THIAGO FERREIRA DE NORONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de junho de 2017.

Dedico este trabalho à todos que o apoiaram e contribuiram durante sua realização. Aos meus pais, pelo apoio incondicional e esforços para fornecer-me os meios e cuidados para que eu pudesse seguir nesta longa jornada acadêmica. Ao meus amigos e colegas, que puderam compartilhar conhecimentos, experiências e momentos de alegria. Aos meus professores, que passaram adiante os ensinamentos e conhecimentos necessários para âmbito acadêmico, profissional e pessoal. Aos orientadores do PPGCC, em especial o professor Geraldo Robson Mateus, por apoiar e acompanhar este longo trabalho.

Agradecimentos

Agradeço aos meus pais e familiares, pelo apoio a este trabalho. Agradeço aos meus professores por contribuir com conhecimentos e materiais para os estudos. Agradeço também ao professor Geraldo Robson pelo interesse e orientação durante o trabalho. E finalmente, agradeço a instituição UFMG e todos os membros do DCC, por anos de compromisso e educação, fazendo parte da pessoa que sou hoje.

"O pessimista vê dificuldade em cada oportunidade.
O otimista vê oportunidade em cada dificuldade."
(Winston Churchill)

Resumo

O Problema do Próximo *Release* (NRP) é um problema de otimização da área de especificação de requisitos, estudada na linha de pesquisa de *Search Based Software Engineering* (SBSE). SBSE tem sido aplicada principalmente em problemas de estimação de custos, requisitos, testes, *debug*, gerência e *design*. O NRP é um problema NP-Difícil, sendo usadas heurísticas e meta-heurísticas para solucioná-lo eficientemente. A decisão de um *release* é importante durante o desenvolvimento de software para o cumprimento de prazos e orçamento. Embora seja um problema amplamente estudado no desenvolvimento tradicional de software, há poucos estudos quando este é aplicado à contextos onde se utilizam metodologias ágeis de desenvolvimento, como o *Scrum*. Este trabalho tem como objetivo abordar o problema partindo deste contexto ágil, analisando modelos e heurísticas existentes e propondo uma nova formulação para o problema, de forma que esta represente melhor a realidade de desenvolvimento de software de equipes ágeis. Um algoritmo foi também desenvolvido com auxílio de experimentos correspondendo a abordagem proposta, apresentando diferentes resultados. Uma análise de integração entre modelos e técnicas de outros problemas gerenciais de Engenharia de Software, como Alocação de Equipes e Estimação de Custos, foi proposta a fim de minimizar possíveis desvantagens da abordagem.

Palavras-chave: *Search Based Software Engineering*, Problema do Próximo *Release*, Meta-heurísticas, Métodos Ágeis, *Scrum*.

Abstract

The Next Release Problem (NRP) is an optimization problem of the requirements specification area, studied in the research field of *Search Based Software Engineering* (SBSE). SBSE has been mainly applied in cost estimation, requirements, testing, debugging, management and design problems. The NRP is an NP-Hard problem, in which heuristics and metaheuristics have been used to solve it efficiently. The decision of a release is important during the software development for meeting deadlines and budget. Although it is a widely studied problem in traditional software development, there are few studies when it is applied to contexts where is used agile development methodologies such as Scrum. This paper aims addressing the problem in the agile context by reviewing existing models and heuristics and proposing a new formulation of the problem that better represents the reality of software development of agile teams. Also, an algorithm was developed with support of experiments to match the proposed approach, presenting mixed results. An analysis of integration between models and techniques of other management problems of Software Engineering, such as Allocation of Teams and Cost Estimation, was proposed to minimize possible disadvantagens of this approach.

Keywords: Search Based Software Engineering, Next Release Problem, Metaheuristics, Agile Methods, Scrum.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Search Based Software Engineering (SBSE) é uma área de pesquisa que tem como objetivo trabalhar com problemas da Engenharia de Software transformando-os em problemas de otimização. Consequentemente, técnicas e métodos de busca são aplicados para resolvê-los, usando meta-heurísticas como algoritmos evolucionários, busca tabu e arrefecimento simulado [Harman & Jones, 2001]. SBSE tem sido aplicado em problemas de estimação de custos e requisitos, de testes e *debug*, de gerência e de *design* [Harman & Jones, 2001; Harman et al., 2009]. Muitos são os problemas de Otimização Combinatória, que se caracterizam pelo número exponencial de soluções. Existem vários casos que não há uma estratégia eficiente para resolver o problema em tempo polinomial. Assim, heurísticas e meta-heurísticas são aplicadas em problemas de otimização a fim de encontrar soluções próximas do ótimo, de maneira muito mais eficiente, usando, por exemplo, estratégias gulosas, buscas por vizinhança ou algoritmos evolucionários [Blum & Roli, 2003].

O Problema do Próximo *Release*, ou *Next Release Problem* (NRP), é um problema de otimização da área de especificação de requisitos que é muito utilizada no planejamento e gerência de projetos [Moura, 2015; Pitangueira et al., 2013]. O NRP tem como objetivo selecionar um subconjunto de requisitos ideais para cumprir o próximo *release*, lançamento ou versão do software, tentando maximizar o ganho e a satisfação dos clientes de maneira que o custo de desenvolvimento esteja no limite permitido [Bagnall et al., 2001]. Em suma, este problema aborda uma equipe de desenvolvimento que possui um conjunto de clientes, cada qual com seu valor de importância e sua lista de requisitos. A equipe de desenvolvimento ainda possui uma lista de todos os requisitos, com seus respectivos custos e uma lista de seus pré-requisitos.

Através de uma busca exaustiva, que calcula todas as combinações possíveis de requisitos, um algoritmo possuiria uma complexidade exponencial. Existem diversas

heurísticas para o problema que procuram métodos mais eficientes de tratá-lo, como as citadas no parágrafo anterior. No entanto, a aplicação das heurísticas varia de acordo com as restrições do problema. Quando o NRP não possui uma lista de pré-requisitos, ele é considerado *básico*. Isto acontece quando os requisitos não estão relacionados entre si ou quando o custo de cada requisito é adaptado à incluir os custos de seus pré-requisitos. Seguindo esta lógica, todo NRP pode ser formulado como básico [Bagnall et al., 2001]. O NRP ainda pode ser *independente*, o que significa que o desenvolvimento de um requisito de um cliente X é independente do desenvolvimento do mesmo requisito para o cliente Y, o que insere o retrabalho ao problema. Isto é, os requisitos podem ser implementados mais de uma vez para diferentes clientes. Mesmo que o NRP seja básico e independente, o problema ainda é *NP-Hard*, equivalendo-se ao Problema da Mochila (*Knapsack Problem*), onde os itens são os clientes, cujo valor de importância é o valor do itens, e a soma dos requisitos do cliente é o peso de cada item [Bagnall et al., 2001; del Sagrado et al., 2010b]. Baseado no trabalho de Bagnall et al. [2001], há uma grande quantidade de trabalhos na literatura sobre o NRP onde foi usado o Problema da Mochila como base para tratá-lo.

A definição de um *release* é uma decisão importante para uma empresa de software com uma cartela grande de clientes. *Releases* mal elaborados podem levar a perda de bons clientes, extrapolar o orçamento de desenvolvimento ou simplesmente não entregar os requisitos a tempo [Bagnall et al., 2001]. Embora o NRP seja normalmente aplicável em problemas de desenvolvimento de software tradicional na literatura, empresas que aplicam metodologias ágeis têm um bom potencial de utilização das soluções para esse problema [Moura, 2015; del Sagrado et al., 2010b]. O interesse das empresas deste contexto se dá pelo fato de que elas em geral trabalham com períodos curtos de *sprints* e *releases*, o que faz esse tipo de decisão ser algo cotidiano para estas organizações.

Uma vantagem das metodologias ágeis é arcar com os desafios de estimar os custos dos requisitos e produção da equipe, uma vez que possuem ferramentas e práticas voltadas para a medição de métricas durante o desenvolvimento [Moura, 2015; Greer & Ruhe, 2004]. Assim, a cada *sprint*, os requisitos e a produção têm seus valores reajustados conforme a *sprint* anterior. Portanto, seria de interesse dessas organizações experimentar diversas formas de tratar o NRP correspondente ao seu contexto, seja básico ou não, independente ou dependente, mono-objetivo ou multiobjetivo [Greer & Ruhe, 2004; del Sagrado et al., 2010b].

Este trabalho analisa modelos e heurísticas existentes aplicadas ao Problema do Próximo *Release*, levando em consideração o contexto de desenvolvimento ágil. Assim, uma nova formulação foi proposta para o problema, procurando não somente atender ao problema de maneira mais eficiente, mas relacioná-lo a outros problemas encontrados no

contexto ágil, como alocação de equipes e estimação de esforços. Esta nova formulação redefine o problema como Múltiplos *Releases*, podendo ser usada por equipes ágeis para tratar particularidades e restrições que são encontradas neste contexto, diferente dos modelos genéricos baseados no problema da mochila. Um experimento também foi aplicado utilizando uma heurística gulosa para avaliar um algoritmo polinomial para o problema.

1.1 Objetivos e Questões de Pesquisa

Este trabalho tem como objetivos principais a proposta de uma nova formulação para o Problema do Próximo *Release*, em contexto ágil de desenvolvimento de software, e a realização de experimentos comparando técnicas exatas e heurísticas aplicadas à formulação proposta. Deseja-se estudar formas de enriquecer esta nova formulação, explorando múltiplos *releases* e analisando possíveis integrações à outros problemas gerenciais, aproveitando técnicas e modelos já consolidados na literatura. Tem-se como hipótese que tal formulação e integração possam fornecer às equipes ágeis a possibilidade de utilizar métodos de otimização mais próximos à sua realidade de desenvolvimento, auxiliando-as no gerenciamento de projetos.

Dado esses objetivos, este trabalho levanta as seguintes questões de pesquisa:

Q1: Quais aspectos dos modelos existentes para o Problema do Próximo *Release* são compatíveis com o contexto ágil de desenvolvimento de software? No modelo proposto, quais novos aspectos foram inseridos?

Q2: Qual o desempenho de meta-heurísticas para solucionar o problema em comparação aos algoritmos de programação linear inteira (*Solvers*)?

Ainda pode-se destacar os seguintes objetivos secundários:

Análise de Meta-heurísticas: durante as análises do Problema do Próximo *Release*, foram pesquisadas as melhores meta-heurísticas implementadas para o problema, de forma a entender o quão efetivo os modelos existentes poderiam ser caso aplicados à realidade das equipes ágeis. A heurística com os melhores resultados na literatura foi implementada para experimentação.

Análise de Problemas de Otimização em Gerenciamento de Projetos: muitos estudos de SBSE são aplicados em problemas gerenciais. Como tem-se o objetivo integrar o NRP a problemas similares, buscou-se um contexto comum próximo da realidade em que as equipes ágeis desenvolvem. Técnicas aplicadas nestes problemas

possívelmente podem ser utilizadas para estender a formulação proposta.

Experimentação: experimentos analisaram o modelo proposto e o algoritmo polinomial implementado, avaliando eficiência, eficácia e adequação ao contexto ágil do modelo.

1.2 Contribuições

Dados os objetivos de pesquisa da seção anterior, define-se que este trabalho fornece as seguintes contribuições à literatura:

1. A primeira contribuição deste trabalho é uma nova formulação para o Problema do Próximo *Release* quando aplicado ao contexto ágil de Desenvolvimento de Software, incluindo alguns aspectos da metodologia *Scrum*. Esta formulação explorando múltiplos *releases* poderá ser usada por equipes ágeis para auxílio em seu planejamento e tomada de decisões, e pela comunidade científica como base de estudo em outras metodologias ágeis similares.

2. É apresentado um algoritmo usando a meta-heurística GRASP Reativo, desenvolvido para experimentação e comparação com a formulação proposta.

3. Os experimentos deste trabalho averiguam o desempenho e eficácia da formulação e algoritmo propostos. Para a comunidade científica, os dados destes experimentos podem ser entendidos como amostras avaliando a meta-heurística GRASP Reativo e a formulação do NRP.

4. A análise de integração de modelos procura inserir o problema estudado a um conjunto de problemas gerenciais clássicos estudados na Engenharia de Software, em especial no campo do SBSE. Esta análise espera trazer resultados qualitativos que justifiquem a importância da resolução deste e outros problemas gerenciais para a Engenharia de Software.

1.3 Estrutura do Documento

A documentação deste trabalho está distribuida da seguinte forma:

O Capítulo 2 apresenta um resumo de trabalhos relacionados da literatura, expondo, para melhor entendimento deste documento, os trabalhos mais relevantes do

problema estudado, do SBSE, das heurísticas e meta-heurísticas e das metodologias ágeis.

O Capítulo 3 apresenta o problema avaliado no contexto de interesse. Em suas seções são descritos este contexto, os aspectos considerados e a formulação proposta.

O Capítulo 4 apresenta a descrição e implementação do algoritmo GRASP Reativo desenvolvido para o problema em estudo. Uma análise de complexidade também foi realizada.

O Capítulo 5 compõe-se das atividades experimentais do trabalho. Testes e experimentos com a formulação e algoritmo foram aplicados em dezenas de instâncias. Uma seção ainda discute a relação deste problema com outros similares da área, dando espaço para a análise de integração proposta anteriormente.

O Capítulo 6 apresenta a conclusão do trabalho, apresentando uma síntese dos resultados observados e analisados, bem como uma breve discussão de trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Enquanto o capítulo anterior apresentou o problema original, os objetivos de pesquisa e a justificativa deste trabalho, esse capítulo procura abordar estudos na literatura de conceitos referentes ao tema deste trabalho, de forma que seja de melhor compreensão a utilização dos termos e métodos na formulação proposta. Espera-se que com o conteúdo apresentado neste capítulo o leitor tenha conhecimentos suficientes para entender a motivação e construção da nova formulação.

A seção 2.1 apresenta conceitos referentes à otimização em Engenharia de Software, área de pesquisa no qual este trabalho está incluído. A seção 2.2 explica algumas das principais meta-heurísticas usadas nos trabalhos da literatura. As meta-heurísticas são técnicas fundamentais aplicadas no problema de otimização apresentado. A seção 2.3 descreve especificações do Problema do Próximo *Release*, bem como a evolução do problema ao longo dos anos. Espera-se que fique evidente as novas contribuições para o problema em estudo. Por último, na seção 2.4 há a apresentação de alguns conceitos e metodologias ágeis, importantes para entender melhor o contexto de aplicação da formulação proposta.

2.1 Otimização em Engenharia de Software

Segundo Harman & Jones [2001], a Engenharia de Software requer conhecimentos e técnicas multidisciplinares para auxiliar desenvolvedores em seus problemas. Muitos desses problemas são de natureza exata, conseguindo ser modelados matematicamente. Nessa situação, existe a preocupação de encontrar soluções próximas da otimalidade ou de tolerância aceitável, sendo estes fatores determinantes na aplicação de meta-heurísticas na resolução desses problemas.

A utilização de métodos de otimização surgiram na Engenharia de Software muito antes do surgimento do conceito de *Search Based Software Enginnering*, sendo utilizado principalmente em estudos de testes de software. Em 1976, Miller & Spooner [1976] publicaram um trabalho que fez uso de métodos de maximização numérica para geração de dados de teste. Em 1992, Xanthakis et al. [1992] usaram um algoritmo genético, uma meta-heurística, para a geração de casos de teste. Outros trabalhos ocasionais surgiram na década de 90, mas somente após o trabalho de Harman & Jones [2001], com a definição do conceito de *Search Based Software Engineering*, é que a área começou a ter diversos estudos presentes na literatura.

Ao longo dos anos 2000, surgiram estudos de otimização sobre diversas áreas de Engenharia de Software, propondo diferentes modelos e formulações para problemas e utilizando diversas heurísticas e meta-heurísticas para resolvê-los. No campo de Engenharia de Requisitos, um dos mais focados em SBSE, estuda-se principalmente o planejamento e estimação de esforços para a implementação dos requisitos, como o trabalho de Ruhe & Saliu [2005]. Paetsch et al. [2003] defendem que esta área, embora seja aplicada a modelos tradicionais de desenvolvimento, também pode ser aplicada ao contexto ágil. Seu estudo representa um indício que pode-se integrar conhecimentos da Engenharia de Requisitos ao NRP, como por exemplo, técnicas de estimação de custos, como o tradicional COCOMO-II [Boehm et al., 2000b]. Uma análise mais detalhada desse campo e dessas técnicas estão presentes no Capítulo 5.

Testes de Software é outro campo muito pesquisado em SBSE, com trabalhos estudando a geração de casos de testes mesmo antes do surgimento da área [Miller & Spooner, 1976], ou propondo, mais recentemente, métodos para seleção ou priorização de casos de testes, como no trabalho de Li et al. [2007]. Segundo Zhang [2012], cerca de metade das publicações da SBSE são do campo de Testes de Software e *Debugging*.

O campo de gerenciamento de projetos está em crescimento com diversos trabalhos recentes sobre alocação de equipes e planejamento de atividades [Torres, 2010]. Peixoto et al. [2013] realizaram uma revisão sistemática de trabalhos referentes a este campo, identificando uma popularidade de algoritmos evolucionários e gulosos na resolução de problemas, pequena aplicação prática e real dos algoritmos, e limitações e ameaças à validade não devidamente discutidas. A similaridade dos problemas de cronogramas com o NRP favorece a integração entre seus modelos, de forma que possam ser aproveitados aspectos de interesse, como produtividade, habilidades e experiências [Ruhe & Saliu, 2005; Torres, 2010], discutidas em mais detalhes no Capítulo 5.

Muitos trabalhos destas e demais áreas, como manutenção, codificação, *design* e arquitetura podem ser encontrados em um repositório mantido por Zhang [2012], um dos principais nomes da área de SBSE.

2.2 Heurísticas e Meta-heurísticas

Heurísticas são algoritmos que não necessariamente retornam uma solução ótima para um problema, mas tendem a encontrar boas soluções em instâncias grandes, sendo que elas normalmente são obtidas com performance e custos aceitáveis para a maioria dos problemas [Talbi, 2009]. As heurísticas são construídas e modeladas para resolver um problema ou instância específica [Talbi, 2009]. As heurísticas podem ser classificadas como construtivas ou de refinamento (aprimoramento). Heurísticas construtivas são aquelas que, dado parâmetros de uma instância, constroem uma solução viável em tempo polinomial, enquanto heurísticas de refinamento são aquelas que, partindo de uma solução viável, procuram no espaço de soluções uma solução ainda melhor [Talbi, 2009]. Algoritmos gulosos e aproximativos são ótimos exemplos de heurísticas construtivas, enquanto algoritmos de busca local, como 2-opt e 3-opt, são exemplos de heurísticas de refinamento [Lenstra, 1997].

Meta-heurísticas, diferente das heurísticas tradicionais, podem ser aplicadas para resolver praticamente qualquer problema de otimização [Talbi, 2009]. Elas podem ser vistas como métodos generalizados que servem como uma estratégia de guia para modelar heurísticas em problemas mais específicos [Talbi, 2009]. Assim, diferente das heurísticas anteriores, é possível caracterizar facilmente diversas meta-heurísticas existentes.

A Busca Tabu foi uma das primeiras meta-heurísticas a surgir, sendo uma das mais simples. Proposta por Glover [1989], é uma meta-heurística que, partindo de uma solução viável inicial, guia um algoritmo que realiza uma busca local em uma vizinhança, procurando uma solução melhor que a anterior até que seja satisfeito um critério de parada. Armazena-se em memória características que permitam que o algoritmo não repita movimentos já realizados.

O Arrefecimento Simulado (*Simulated Annealing*), ou Têmpera Simulada, foi estudado por diversos autores desde a década de 80 [Kirkpatrick et al., 1983]. O algoritmo também procura por uma solução melhor do que a atual no espaço de soluções, de acordo com uma função objetivo baseada em uma variável T (temperatura). Quanto maior o valor de T, maior será a aleatoriedade na busca da próxima solução. O valor de T diminui à medida que o algoritmo progride para uma solução ótima local.

Outra meta-heurística popular é o *Greedy Randomized Adaptive Search Procedure* (GRASP) [Feo & Resende, 1989]. Diferente de outras meta-heurísticas, o GRASP foca mais na fase construtiva do que na de refinamento. O GRASP irá construir aleatoriamente soluções iniciais, de acordo com uma variável que representa o grau de aleatoriedade a cada iteração, aplicando uma busca local para cada solução encon-

trada, armazenando a melhor encontrada durante o procedimento. Com esta estratégia, busca-se encontrar diferentes boas soluções iniciais, para que a fase de refinamento encontre diversos ótimos locais.

Algoritmos genéticos são outro tipo de meta-heurística presente na literatura desde a década de 80, com os trabalhos de Goldberg & Holland [1988]. Este tipo de algoritmo trata-se de uma simulação do processo de evolução de espécies. Uma população é inicialmente selecionada para buscar soluções melhores. A fase de evolução ocorre por meio de gerações, que representam as iterações. A cada geração, alguns indivíduos da população são selecionados para a próxima geração, sofrendo uma recombinação (*crossover*) ou mutação para formar uma nova população, repetindo o processo.

Por fim, uma outra meta-heurística muito aplicada é a Colônia de Formigas (*Ant Colony*), também conhecida como ACO (*Ant Colony Optimization*) [Dorigo, 1992]. Este tipo de algoritmo tenta imitar o comportamento de formigas que caminhariam por um grafo deixando *"rastros"* de bons caminhos encontrados. Outras formigas passariam, então, a seguir estes rastros. No entanto, com o passar do tempo, esses rastros iriam diminuir até desaparecer. Com isso, evita-se que o algoritmo sofra uma convergência precoce para uma solução ótima local. Outra vantagem deste tipo de algoritmo é que ele reage bem a mudanças dinâmicas em um grafo, sendo úteis para aplicações em tempo real.

2.3 Problema do Próximo Release

O primeiro e principal trabalho da literatura sobre o Problema do Próximo *Release* foi de Bagnall et al. [2001], tratando o NRP como um problema de otimização mono-objetivo, buscando maximizar a satisfação dos clientes baseado em uma lista de requisitos e pré-requisitos. Os autores demonstram a similaridade do problema com o problema da mochila. Além de apresentar o problema, incluindo uma formulação matemática para tal, o artigo traz experimentos de algoritmos exatos, gulosos e de busca local para tratar o problema, usando *Hill Climbing* e *Simulated Annealing*. Sua estrutura do NRP, que foi utilizada como base em diversos estudos que a seguiram, está exemplificada na figura 2.1.

Greer & Ruhe [2004] descrevem o planejamento de um *release*, onde o problema foi abordado da perspectiva de desenvolvimento incremental de software, através de uma abordagem baseada em algoritmos genéticos e técnicas iterativas. Ruhe & Saliu [2005] fizeram um novo trabalho onde incluiam aspectos humanos ao seu modelo, como intuição, capacidades, conhecimento e experiências.

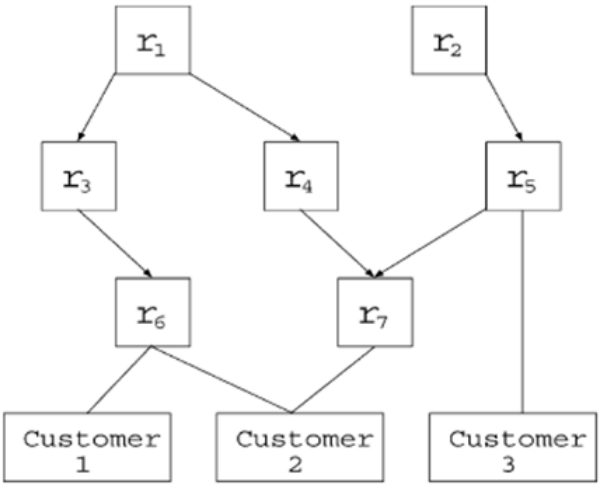

Figura 2.1. Estrutura de um NRP não básico e dependente [Bagnall et al., 2001]

Zhang et al. [2007] formulam o problema tratando-o como multiobjetivo, desejando, além de maximizar a satisfação dos clientes, minimizar os custos de implementação dos requisitos. Foram utilizadas meta-heurísticas para resolver o problema, onde os mais eficientes foram os algoritmos genéticos.

van den Akker et al. [2008] utilizaram uma ferramenta de otimização baseada em Programação Linear Inteira, apresentando uma nova formulação do NRP que assume que o conjunto ideal de requisitos não é aquele que somente maximiza a satisfação dos clientes, mas apresenta o máximo de receita projetada para os recursos disponíveis. Assim, uma nova variável é inserida no contexto, as receitas previstas. Uma notável característica deste trabalho é a estrutura modular da formulação, de forma que ela varie sob certas condições, como por exemplo, restrições sendo adicionadas para casos onde equipes e prazos são estendíveis. O trabalho faz referência a mecanismos para análise *what-if* para identificação de riscos para o NRP.

del Sagrado et al. [2010a] fizeram várias contribuições para o problema. Destaca-se o trabalho realizado em 2010, no qual é aplicado uma heurística de colônia de formigas (*Ant Colony Optimization*) em seus experimentos, comparando-a com outras meta-heurísticas já existentes, como algoritmos genéticos e *Simulated Annealing*.

Linhares et al. [2010] têm como contribuição a aplicação da meta-heurística GRASP Reativo no problema, comparando com outras meta-heurísticas usadas por outros autores em trabalhos anteriores. Os resultados de seus experimentos levaram à conclusão que o GRASP Reativo apresenta resultados melhores do que algoritmos genéticos e de têmpera simulada.

Por fim, um trabalho recente de Moura [2015] faz uma pequena adaptação à

formulação do NRP para o contexto de uma empresa que aplica desenvolvimento ágil. A autora adapta as variáveis originais do problema em similaridade ao Problema da Mochila, mostrando que o NRP pode ser aplicado para esse tipo de contexto. No entanto, neste trabalho não é apresentado nenhum modelo ou extensão à formulação original.

Tendo em mente este breve histórico de modelos e experimentos do problema em questão, nota-se uma carência de sua aplicação a outros contextos de desenvolvimento, enquanto diversas heurísticas já foram testadas. Justifica-se explorar outras perspectivas de forma a enriquecer este problema na literatura. Com isso, partindo da proposta de contexto de Moura e do modelo matemático extensível e modular de Akker, deseja-se a criação de uma formulação genérica o suficiente para atender o contexto de desenvolvimento ágil de software, e que permita que futuras extensões possam ser aplicadas para tratamento de casos específicos do contexto.

Como Sagrado e Linhares obtiveram os melhores resultados heurísticos, foi escolhido como experimentação a heurística do GRASP Reativo para implementação. Ressalta-se que em muitos experimentos as técnicas exatas obtiveram bom desempenho perante às heurísticas. Há uma coerência para este comportamento, uma vez que as técnicas exatas podem tirar proveito das características estruturais do NRP, em relação aos diversos avanços bem sucedidos para solucionar o Problema da Mochila [Bagnall et al., 2001].

2.4 Métodos Ágeis

Métodos ágeis são um conjunto de metodologias de desenvolvimento de software fundamentadas pelos princípios do Manifesto Ágil, que defendem simplicidade, flexibilidade, interação e melhoria contínua no processo de desenvolvimento de software [Beck et al., 2001]. Uma caractéristica de destaque dos métodos ágeis é o desenvolvimento em iterações de curtos períodos [Cohen et al., 2003]. Cada iteração representa a implantação de uma nova versão de software, sendo que novas prioridades serão determinadas para as iterações seguintes. Cada iteração também é representada por um fluxo de atividades ou tarefas, correspondendo a um processo ou projeto de software [Cohen et al., 2003]. Métodos tradicionais normalmente não se preocupam em garantir essas características. Ainda mais, métodos ágeis são por natureza mais adaptativos, enquanto os tradicionais são mais prescritivos [Kniberg & Skarin, 2010]. Tal dinamicidade dificulta a proposta de uma formulação robusta e bem definida.

No entanto, muito antes do surgimento do Manifesto Ágil já haviam trabalhos

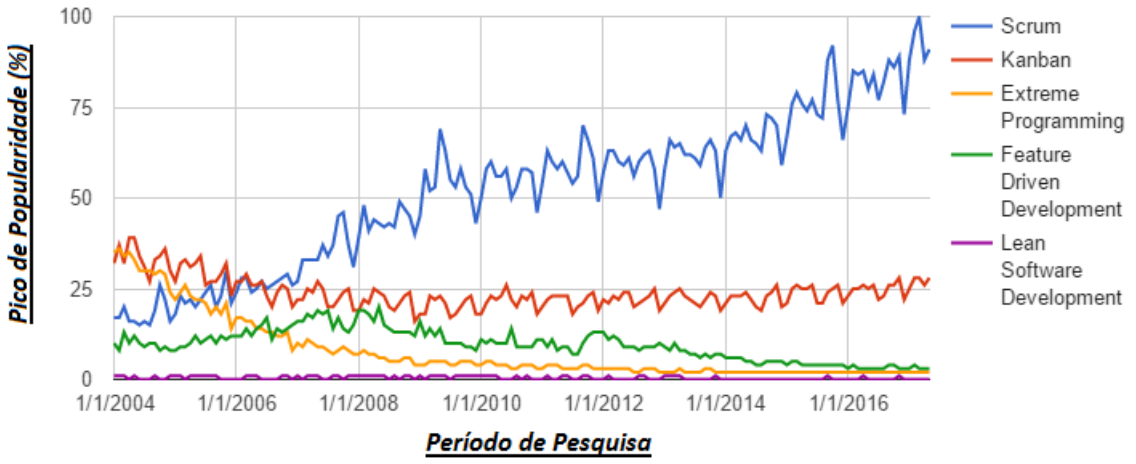

Figura 2.2. Popularidade de alguns métodos ágeis entre 2004 e 2016

que defendiam este tipo de metodologia, como apresentado por Victor [2003]. Desde então, diversos métodos têm surgido ou sido complementados com diferentes propósitos e técnicas para atender esses princípios. Alguns métodos focam mais em práticas como o *Extreme Programming* (XP) [Beck & Anders, 1999] e o *Agile Modeling* [Ambler, 2002], enquanto outros focam mais no gerenciamento de projetos, como o *Scrum* [Beedle et al., 1999] e o Kanban [Anderson, 2010]. A figura 2.2 mostra a popularidade de alguns métodos nos últimos anos, em função de suas buscas na Internet. Foi utilizado como parâmetro de consulta o termo *"Metodologias ágeis"* desde 2004, incluindo resultados de características acadêmicas, comerciais, profissionais e pessoais. Os resultados foram filtrados pelos maiores picos de popularidades dos métodos pesquisados.

É comum misturar técnicas e práticas dos diversos métodos existentes para atender especificações do desenvolvimento de software, sendo *Extreme Programming* e *Scrum* os métodos com os conceitos mais populares. Dentre as práticas mais comuns do XP estão a programação em pares, o *Test Driven Development* (TDD), a refatoração, os *releases* pequenos, a propriedade coletiva do código, o *Planning Game* e colaboração entre clientes e desenvolvedores [Beck & Anders, 1999]. O método XP continua sendo atualizado, adotando novas técnicas ou padrões observados por experiência de uso ao longo dos anos [Wells, 2009]. Enquanto o XP destaca-se pelos métodos práticos, o *Scrum* enfatiza técnicas gerenciais, como os papéis de *Scrum Master* e *Product Owner*, ciclos caracterizados como *sprints*, lista de requisitos e tarefas como Backlog, atividades como o planejamento e revisão de *sprint*, *Scrum* diário, e artefatos visuais, como o quadro de tarefas e o gráfico de *Burndown* [Beedle et al., 1999; Kniberg & Skarin, 2010]. Ambos os métodos são bem adaptativos e enxutos, com apenas 13 prescrições

para o XP e 9 prescrições para o *Scrum* [Kniberg & Skarin, 2010].

Outros métodos populares além dos já citados são o *Feature Driven Development* (FDD) [Palmer & Felsing, 2001], com desenvolvimento voltado para características, *Crystal Clear* [Cockburn, 2004], com métodos focados em pessoas, *Agile Unified Process* [Edeki, 2013], um modelo simplificado de RUP [Kruchten, 2004], e *Adaptive Software Development* [Highsmith, 2013], que propõe ciclos adaptativos no processo de desenvolvimento de software.

Sendo que um dos objetivos do trabalho é a criação de uma formulação que represente melhor a realidade deste tipo de desenvolvimento, nota-se que é imprescindível que ele permita a representação de iterações, seja sob forma de *sprints* ou de diversas *releases*. Como discutido na subseção anterior, nenhuma formulação trata devidamente os aspectos citados nos parágrafos anteriores, motivando e justificando os objetivos deste trabalho.

Capítulo 3

Problema do Próximo Release Aplicado em Contexto Ágil

3.1 Contextualização

O Problema do Próximo *Release* quando aplicado ao contexto ágil de desenvolvimento de Software deve levar em consideração as particularidades deste tipo de desenvolvimento. Seguindo as considerações do Capítulo 2, o contexto da formulação proposta deve apresentar algum mecanismo que represente um conjunto de iterações que irão ocorrer para sucessivas versões de software. Ainda, a formulação deve manter os elementos e propósitos do NRP original, apresentados no Capítulo 1: equipe(s) desenvolvendo requisitos de clientes cada qual com sua importância para a organização, onde é desejado a maximização da satisfação dos clientes desde que o custo de desenvolvimento esteja dentro do limite permitido [Bagnall et al., 2001].

Embora a formulação proposta seja genérica o suficiente para ser adaptada a qualquer metodologia ágil utilizada, será considerado como base de aplicação neste trabalho o uso da metodologia *Scrum*, com o propósito de facilitar a explicação dos elementos da formulação e subsequentes experimentos. Neste contexto, uma empresa de desenvolvimento deverá implementar requisitos para atender seus clientes dado um prazo ou intervalo de tempo [Beedle et al., 1999]. O *Scrum* caracteriza-se por ser executado em diversas iterações como *sprints*, tendo como objetivo maximizar o valor obtido por atender os requisitos desenvolvidos [Beedle et al., 1999]. Uma *sprint* representa um ciclo de desenvolvimento da organização. Quando comparado ao Problema do Próximo *Release* clássico, a definição de *sprint* é bem próxima do conceito de *release*. Assim, neste caso específico, será considerado um único produto em desenvolvimento, onde

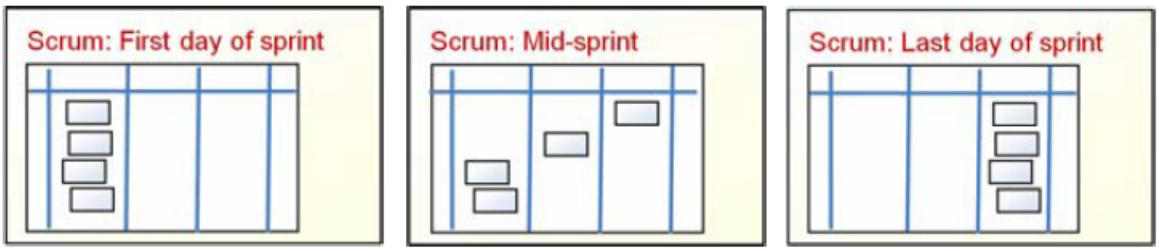

Figura 3.1. O fluxo de um Scrum Board [Kniberg & Skarin, 2010]

cada *sprint* representa um projeto para liberar novas funções deste produto a seus clientes.

A implementação de cada requisito segue as etapas de um processo de desenvolvimento de software ou atividades pré-definidas, atribuídas a um Quadro *Scrum*, que demonstra o fluxo dos requisitos trabalhados na *sprint* [Beedle et al., 1999]. A figura 3.1 demonstra esse comportamento. As colunas representam etapas ou atividades do processo, enquanto as linhas possuem as histórias ou requisitos da *sprint*, representados por *post-its*. Práticas gerenciais como o *Planning Poker*, as reuniões diárias e o planejamento de *sprints* podem fazer parte da rotina deste modelo de desenvolvimento [Kniberg & Skarin, 2010; Cohen et al., 2003]. Ao término da *sprint*, tem-se por hábito remover os requisitos do quadro e realizar uma reunião para escolha dos requisitos da próxima *sprint*, fazendo os ajustes necessários nas equipes, nos requisitos ou no processo [Kniberg & Skarin, 2010]. Dada esta última afirmação, entende-se que a formulação prevê várias *sprints*, podendo ser reexecutada a cada iteração. Assim, os dados da formulação podem ser atualizados, gerando novas e mais precisas soluções.

3.2 Aspectos Considerados

Nesta seção serão descritos os elementos que compõem a formulação ou que estejam indiretamente ligados à ela. Assim, muitas das afirmações desta seção representam como estes elementos são interpretados no modelo, não necessariamente sendo verídicos para outros contextos ou linhas de pesquisa da Engenharia de Software ou da Otimização Matemática. Além disso, alguns aspectos foram simplificados o suficiente para atender o grau de generalização desejado do modelo, melhor discutidas na subseção 3.5.4.

Há diversos tópicos de interesse no contexto descrito na seção anterior. Antes de detalhar cada um deles, é importante ter em mente as definições básicas desses tópicos. Referências de Sommerville [2011], a seguir são apresentados alguns conceitos clássicos da Engenharia de Software. Primeiramente, considere uma nova versão de

software, ou **release** como um conjunto de **requisitos** funcionais ou não funcionais implementados durante certo período. Todo requisito a ser implementado passa por um **processo** de desenvolvimento de software, composto por uma série de **atividades** como especificação, modelagem, implementação e testes. Há diversos modelos de processos de software que definem sua estrutura, como o cascata e o espiral. Uma atividade pode ser considerada como um conjunto de tarefas a serem executadas pelas **equipes** de desenvolvimento. Uma equipe é uma unidade capaz de executar as atividades para implementar requisitos. Por fim, as **sprints** representam os ciclos de desenvolvimento das equipes, cada qual com sua duração.

3.2.1 Atividades

As atividades, como normalmente chamadas em contexto de Engenharia de Software, são instruções estruturadas e bem definidas dentro de um processo de software, resultando em um comportamento ou ação [Rumbaugh et al., 2004; Sommerville, 2011]. Neste contexto ágil, considera-se atividades como um conjunto de tarefas a serem realizadas para cumprimento de uma etapa de desenvolvimento do requisito. As atividades, ou o conjunto de tarefas, a serem executadas dependem do processo de software definido pela organização. Especificação de requisitos, implementação e testes são alguns exemplos de atividades de desenvolvimento [Sommerville, 2011]. Uma ou mais equipes podem ficar alocadas para cumprir essas atividades. Por fim, uma vez que está sendo tratado um único produto nesta aplicação, considera-se que todos os requisitos passam pelo mesmo processo de software. Ou seja, todos os requisitos necessitam das mesmas atividades para serem implementados.

3.2.2 Processo de Desenvolvimento de Software

Um processo é um conjunto de atividades ou etapas parcialmente ordenadas (sob ações sequenciais, ramificadas, condicionadas ou iterativas) com a intenção de atingir uma meta. Em Engenharia de Software esta meta normalmente é entregar, da maneira mais eficiente possível, um produto de software capaz de atender as necessidades do negócio [Rumbaugh et al., 2004; Sommerville, 2011]. A complexidade do processo aumenta à medida que mais atividades e fluxos vão sendo incluídos [Rumbaugh et al., 2004]. Ao executar o *Scrum*, espera-se que seja definido um processo de desenvolvimento de software ou um conjunto de tarefas, para que seja medida a produtividade das equipes durante a *sprint*. Um exemplo de processo de software clássico é representado pelas seguintes atividades: especificação de requisitos, projeto/modelagem, implementação,

validação/testes e implantação/manutenção.

3.2.3 Equipes

As equipes são conjuntos de desenvolvedores responsáveis por determinadas tarefas no processo de desenvolvimento [van den Akker et al., 2008]. Em contexto *Scrum*, as equipes são tratadas como *times* [Beedle et al., 1999]. Esta diferenciação parte da ideia de que as pessoas envolvidas tendem a estar comprometidas com o negócio. Dependendo das técnicas utilizadas, a produtividade de cada equipe pode ser medida durante um certo número de *sprints* para auxiliar previsões e planejamentos futuros. No *Scrum*, é comum as equipes serem multifuncionais, estando aptas a executar quaisquer atividades durante a implementação dos requisitos [Kniberg & Skarin, 2010]. A especialização e a produtividade são alguns fatores que determinam a atribuição das equipes [Torres, 2010]. Por fim, uma equipe pode ser composta por um número indefinido de membros, idealmente entre três e nove pessoas. Em caso de duplas, costuma-se usar práticas de XP.

No contexto exemplificado para a formulação não será considerado a sobreposição de membros (uma pessoa pertencendo a mais de uma equipe) e trocas imprevistas entre membros. Portanto, caso haja mudanças entre membros das equipes durante as *sprints*, será necessário reexecutar a formulação com os novos dados. Normalmente, aloca-se uma equipe para um determinado projeto durante a *sprint*. No entanto, como neste caso trata-se apenas de um projeto e produto na *sprint*, as equipes podem trabalhar com o mesmo conjunto de requisitos.

3.2.4 Requisitos de Software

Um Requisito de Software é uma característica de projeto, propriedade ou comportamento de um sistema [Rumbaugh et al., 2004]. Requisitos são expressos como histórias no *Scrum*, sendo funções a serem implementadas pelas equipes de desenvolvimento para seus clientes. Na prática, um requisito pode gerar uma ou mais histórias a serem desenvolvidas [Kniberg & Skarin, 2010]. Cada requisito deve passar pelas atividades do processo de desenvolvimento [Beedle et al., 1999].

No contexto apresentado, para cada atividade, um requisito irá possuir uma determinada **duração** para sua realização (e.g. Custo de Tempo), variando de equipe para equipe. Além da duração, será denominado aqui um **esforço**, que representará os gastos de recursos que uma equipe necessita para executar aquela atividade (e.g. Custos Financeiros). Ao ser realizadas todas as suas atividades, um requisito será con-

siderado implementado. Cada requisito possui um ganho de recursos, ou valor, ao ser implementado, correspondente à sua importância para a organização. Atribuir valores e custos aos requisitos é um procedimento abstrato, mas existem percepções e técnicas populares para tal atribuição, como o *Planning Poker*. Os requisitos são divididos em duas categorias: funcionais e não funcionais. O primeiro são os requisitos que correspondem às funções do sistema (e.g. Gerar Relatórios, Criar Registros, Cadastrar Fornecedores), enquanto o segundo são os requisitos que caracterizam qualitativamente o sistema (e.g. Segurança, Desempenho, Robustez) [Sommerville, 2011].

3.2.5 Backlog

O Backlog é o conjunto de requisitos que podem ser implementados durante uma ou mais *sprints* [Kniberg & Skarin, 2010]. Não é estritamente necessário implementar todos os requisitos do Backlog durante o ciclo. No *Scrum*, os requisitos do Backlog vão sendo alocados às equipes para cumprir as diversas etapas de desenvolvimento. Como as equipes são multifuncionais neste caso, os requisitos podem ser desenvolvidos inteiramente por uma equipe ou serem repassados para outras equipes em etapas futuras.

3.2.6 Sprint

Sprint é cada um dos ciclos de desenvolvimento da empresa [Kniberg & Skarin, 2010]. É a definição mais próxima de *release* quando comparado ao problema original. Enquanto neste os requisitos são implementados dado o prazo do *release*, em contexto *Scrum* tem-se que os requisitos são implementados dado o prazo das *sprints*. A cada iteração, requisitos são selecionados do *Backlog* para serem implementados seguindo o processo de desenvolvimento da empresa [Beedle et al., 1999]. Todo ciclo possui um prazo (normalmente entre uma e quatro semanas), onde neste período são mensurados valores, custos e produtividade da implementação dos requisitos [Cohen et al., 2003]. No contexto apresentado considera-se que cada *sprint* representará um projeto para adição de funcionalidades à um produto. A cada *sprint*, pode-se reexecutar a formulação atualizando o Quadro *Scrum* e o *Backlog* com novos itens ou valores.

3.2.7 Custos de Requisito

Existem diferentes formas de estimar o custo de um requisito, sendo normalmente atrelado à atribuições temporais ou financeiras. No *Scrum*, a estimação normalmente ocorre por consensos de equipe ou medições anteriores de produtividade [Cohen et al.,

2003]. Neste contexto, o custo de tempo é a **duração**, correspondente ao tempo necessário para a realização de todas as atividades do requisito. Já o custo financeiro está atrelado ao **esforço** da equipe, simulando o gasto de recursos da organização. Como limitação da *sprint*, o tempo total de implementação dos requisitos não pode superar o prazo da *sprint*. Da mesma forma, o esforço total da implementação dos requisitos não pode superar o ganho de recursos da organização (caso contrário resultaria em um déficit de recursos ao fim das *sprints*).

3.2.8 Planning Poker

Planning Poker é um método comum utilizado em métodos ágeis, de origem no XP, para se criar um consenso quanto à estimação de alguns parâmetros descritos anteriormente, como valores e custos, sendo normalmente mais usado para o último [Beck & Anders, 1999]. *"Cartas"*com valores numéricos são distribuídos entre os envolvidos, onde cada um deve sugerir um valor para o requisito. As pessoas com os valores mais divergentes deverão debater suas opiniões e realizar uma nova sugestão de valores, até que haja um consenso total. Este método é uma alternativa para fornecer os dados de entrada necessários para a formulação.

3.2.9 Clientes

Uma organização pode ter um ou mais clientes, que fornecem os requisitos para serem desenvolvidos. Em *Scrum* o papel destes clientes está associado ao *Product Owner* ou ao *Stakeholder* [Beedle et al., 1999]. Na maioria das metodologias ágeis, o acompanhamento do cliente durante a *sprint* é fundamental para participar da tomada de decisões. Dependendo da técnica utilizada, ele também pode participar da estimação de custos e valores dos requisitos. No problema original do Próximo *Release*, o valor para a empresa estava atrelado aos Clientes, enquanto neste contexto, como descrito anteriormente, faz mais sentido atrela-lo à implementação dos requisitos. Em outras palavras, o papel do cliente é representado indiretamente na formulação proposta.

3.2.10 Outras práticas ágeis

É provável que outras práticas ágeis estejam inseridas no contexto que, de certa forma, impactará nos aspectos anteriores. Embora essas demais práticas não estejam inseridas explicitamente na formulação do problema, é apropriado que sejam levadas em consideração, como o Planejamento de *sprints*, *Timeboxing*, *Scrum* Diário, Gráfico de *Burndown*, *Test Driven Development*, Refatoração, Medição de Software, entre outras.

3.3 Formulação do Problema

- Seja **A** o conjunto de atividades do processo de desenvolvimento de software.
- Seja **E** o conjunto de equipes disponíveis para executar as atividades de **A**.
- Seja **R** o conjunto de todos os requisitos de software a serem desenvolvidos, que passam por todas as atividades de **A**.
- Seja **P** o conjunto de *sprints* disponíveis para a implementação dos requisitos.
- Seja a matriz **T** tal que $\mathbf{t_{are}}$ é a duração, ou o tempo necessário, que a equipe *"e"* leva para executar a atividade *"a"* do requisito *"r"*.
- Seja a matriz **C** tal que $\mathbf{c_{are}}$ é o esforço, ou gasto de recursos, que a equipe *"e"* tem para executar a atividade *"a"* do requisito *"r"*.
- Cada requisito $r \in R$ possui um ganho de recursos $\mathbf{v_r}$ que mensura a importância do requisito r para a organização. Este ganho é obtido somente quando todas as atividades associadas a este requisito forem executadas.
- Cada equipe $e \in E$ possui uma disponibilidade $\mathbf{d_e}$ que corresponde ao tempo máximo de execução de atividades que aquela equipe pode dedicar por *sprint*.
- Cada *sprint* $p \in P$ possui um prazo $\mathbf{l_p}$ que corresponde ao tempo máximo de execução de atividades que uma *sprint* pode ter, dado o somatório de tempo de todas as atividades executadas pelas equipes.

Deseja-se maximizar o somatório dos ganhos $\mathbf{v_r}$ menos o somatório dos esforços $\mathbf{c_{are}}$ dos requisitos implementados, contanto que:

1. O tempo de execução das atividades na *sprint "p"* não ultrapasse $\mathbf{l_p}$.
2. O tempo de execução das atividades pela equipe *"e"* na *sprint "p"* não ultrapasse $\mathbf{d_e}$.
3. Cada atividade de cada requisito tenha no máximo uma equipe atribuída à ela.
4. Um requisito implementado deve ter todas as suas atividades executadas.
5. Os requisitos devem ser implementados seguindo a ordem $\mathbf{A_{1..n}}$.

Considera-se que não há uma lista de pré-requisitos para a implementação. Como explicado no Capítulo 1, as dependências podem ser simuladas inserindo o custo dos pré-requisitos em seus requisitos correspondentes (transformando o problema em um NRP Básico) [Bagnall et al., 2001].

Assim, tem-se :

- Atividades:

  A = $\{a_1, a_2, ..., a_{|A|}\}$, que é o conjunto das atividades do processo de desenvolvimento de software, por qual todo requisito deve passar para ser implementado. Uma atividade leva um certo tempo $"t"$ e esforço $"c"$ para ser executada.

- Equipes:

  E = $\{e_1, e_2, ..., e_{|E|}\}$, que são as equipes que participam do desenvolvimento.

- Requisitos:

  R = $\{r_1, r_2, ..., r_{|R|}\}$, que são todos os requisitos disponíveis do Backlog para serem implementados. Um requisito $"r"$ garante um ganho $"v_r"$ ao ser implementado.

- *Sprints*:

  P = $\{p_1, p_2, ..., p_{|P|}\}$, que são as *sprints* disponíveis para a implementação dos requisitos do Backlog. Não é necessário que todo o Backlog seja alocado, mas tem-se como objetivo obter o maior ganho possível dos requisitos implementados.

Que se relacionam como:

- Requisitos e Atividades:

  Todo requisito deve passar por todas as atividades de **A**. Após uma atividade $"a"$ do requisito $"r"$ ser concluída, ele deverá passar para a próxima atividade $"a+1"$ do conjunto **A**. Após cumprir todas as atividades de **A**, o requisito $"r"$ garante um ganho $"v_r"$ para a empresa.

- Equipes, Requisitos e Atividades:

  Por serem multifuncionais neste contexto, qualquer equipe poderia em tese executar qualquer atividade de qualquer requisito do projeto, dada sua capacidade e produtividade. Uma equipe precisa de um tempo $"t_{are}"$ e um esforço $"c_{are}"$ para executar a atividade $"a"$ do requisito $"r"$. O tempo de execução é usado para

calcular a viabilidade da implementação quanto ao prazo da *sprint*, enquanto o esforço deduz os custos de desenvolvimento dos ganhos dos requisitos implementados (ganhos de recursos menos o gasto de recursos).

- *Sprint*, Equipes, Requisitos e Atividades:

  As equipes irão executar as atividades dos requisitos em determinadas *sprints*. Uma atividade $"a"$ do requisito $"r"$ só pode ser executada uma vez, em uma única *sprint* $"p"$ por uma equipe $"e"$. O tempo de execução das atividades de uma equipe deve ser inferior ou igual à $"d_e"$, que é a disponibilidade de tempo que esta equipe tem por *sprint*. Já o tempo de execução das atividades de todas as equipes por *sprint* deve ser inferior ou igual à $"l_p"$.

Ainda pode-se caracterizar como alguns parâmetros do problema serão configurados:

- Estimação de Tempo, Ganho e Esforço das Atividades e Requisitos:

  A forma com que os custos dos requisitos serão estimados afetará os parâmetros de entrada do problema. Duas práticas comuns para se realizar esse procedimento são a aproximação do cliente *(Product Owner)* junto à equipe de desenvolvimento (para participar da definição do requisito como características, valor e custo) e a utilização de alguma técnica similar ao *Planning Poker*, descrito anteriormente, para consenso das propostas.

- Determinação da Disponibilidade das Equipes:

  Uma ou mais equipes poderão ser alocadas às atividades do processo. Podem ser levados em consideração dois fatores nessa atribuição: a complexidade da atividade e a produtividade da equipe. Espera-se que atividades mais complexas possuam mais equipes competentes para impedir um gargalo nesta atividade do processo. Novamente, a medição de *sprints* anteriores é a principal técnica para auxiliar essa distribuição.

- Determinação do Prazo das *Sprints*:

  Uma *sprint* costuma durar entre uma e quatro semanas. O tempo ideal para determinar este prazo pode ser feito por duas técnicas básicas. A primeira, novamente, é a medição do desempenho da *sprint* anterior, identificando se a quantidade de requisitos implementados é compatível com o prazo determinado. Outra técnica é estimar o prazo baseando-se no caminho crítico. Por definição, o

caminho crítico é o caminho mais longo de um grafo. Traduzindo para o contexto do problema, calcular o caminho crítico significaria calcular o pior caso deste problema, gastando o maior tempo possível para implementar todos os requisitos.

3.4 Modelo Matemático

Seja:

A = $\{1, ..., a, ..., |A|\}$ - o conjunto de atividades do processo de desenvolvimento;
E = $\{1, ..., e, ..., |E|\}$ - o conjunto de equipes para execução das atividades;
R = $\{1, ..., r, ..., |R|\}$ - o conjunto de requisitos do Backlog a serem implementados;
P = $\{1, ..., p, ..., |P|\}$ - o conjunto de *sprints* disponíveis para a implementação;

Dado que:
t_{are} - tempo para execução da atividade a do requisito r pela equipe e.
c_{are} - esforço para execução da atividade a do requisito r pela equipe e.
v_r - ganho (ou valor) adquirido pela implementação do requisito r.
d_e - disponibilidade de tempo da equipe e por *sprint*.
l_p - prazo (ou limite) da *sprint* p.

Com as seguintes variáveis:

$$x_{arep} = \begin{cases} 1, & \text{caso a atividade } a \text{ do requisito } r \text{ foi alocada à equipe } e \text{ para a } \textit{sprint } p; \\ 0, & \text{caso contrário.} \end{cases}$$

$$y_r = \begin{cases} 1, & \text{caso o requisito } r \text{ foi implementado;} \\ 0, & \text{caso contrário.} \end{cases}$$

- Função Objetivo:

Maximizar:

$$max \sum_{r \in R} (v_r y_r - \sum_{e \in E} \sum_{a \in A} \sum_{p \in P} c_{are} x_{arep})$$

que representa o somatório de todos os ganhos v_r dos requisitos implementados menos o esforço c_{are} das equipes que foi necessário para a execução de todas as atividades nas *sprints*.

- Restrições:

  O problema apresenta as seguintes restrições:

$$\sum_{a \in A} \sum_{r \in R} \sum_{e \in E} t_{are} x_{arep} \leq l_p \quad , \forall p;$$

  O tempo de implementação de todas as atividades de requisitos de cada equipe, para cada *sprint*, não pode ser superior ao limite de prazo daquele *sprint* l_p.

$$\sum_{e \in E} \sum_{p \in P} x_{arep} \leq 1 \quad , \forall a; \forall r;$$

  Representa a condição que cada atividade a do requisito r somente pode ser executada por uma equipe e durante a *sprint* p.

$$\sum_{a \in A} \sum_{r \in R} t_{are} x_{arep} \leq d_e \quad , \forall e; \forall p;$$

  O tempo de implementação das atividades dos requisitos por cada equipe em cada *sprint* deve ser menor ou igual ao tempo disponível pela equipe para cada *sprint* ($\mathbf{d_e}$).

$$\sum_{a \in A} \sum_{e \in E} \sum_{p \in P} x_{arep} = |A| y_r \quad , \forall r;$$

  Representa a condição que todas as atividades a de um requisito r devem ser executadas para que o requisito r seja considerado implementado. Para isso, a soma de todas as atividades executadas deve ser igual à cardinalidade do conjunto **A**.

$$\sum_{e \in E} \sum_{j=1}^{p} x_{(a-1)rej} \geq \sum_{e \in E} x_{arep} \quad , \forall p; \forall r; \forall a \geq 2;$$

  Representa a condição que uma atividade a do requisito r só pode ser executada após a execução da atividade *a-1* do requisito r. Como ambos tempo de execução de uma atividade e a disponibilidade de tempo das equipes não são afetadas por ordem de implementação, pode-se considerar que esta restrição precisa validar apenas em qual *sprint* uma atividade é executada. Logo, pode-se garantir que se as atividades i, j e k são executadas em uma *sprint*, então qualquer combinação

entre elas também pode ser executada nesta mesma *sprint*. No entanto, essa combinação poderia não ser permitida se elas fossem executadas em *sprints* diferentes. Assim, para toda atividade a de um requisito r executada em uma *sprint* p, alguma equipe deve ter executado a atividade *a-1* do requisito r em alguma *sprint* igual ou inferior à p. Desta forma, garante-se a viabilidade de executar as atividades em ordem.

$x_{arep} \in \{0,1\}$, $y_r \in \{0,1\}$,
$t_{are} \geq 0$, $c_{are} \geq 0$, $v_r \geq 0$, $d_e \geq 0$, $l_p \geq 0$ $\quad ,\forall a; \forall r; \forall e; \forall p;$

Representa as restrições de integralidade e não negatividade dos elementos da formulação.

3.5 Outras Características

Esta seção apresenta alguns esclarecimentos à particularidades não discutidas diretamente nos tópicos anteriores.

3.5.1 Aplicabilidade

O problema e a formulação apresentados podem ser aplicados, à certo nível, em praticamente qualquer metodologia ágil de desenvolvimento, sendo o *Scrum* utilizado como um exemplo de base de aplicação. A abordagem utilizada trata de vários aspectos encontrados em ambiente real dessa metodologia, como as *sprints*, as tarefas, as histórias e os times. Os parâmetros necessários para a formulação normalmente são baseados em dados históricos de *sprints* anteriores, tornando o uso da abordagem consistente em termos de valores de requisitos, custos e desempenho das equipes.

Embora haja um enfoque para alguns aspectos da metodologia *Scrum*, a abordagem pode ser aplicada em outros ambientes reais de desenvolvimento. Outras metodologias ágeis adaptáveis podem usar a estrutura do *Scrum* como método de planejamento ou cronograma de suas atividades. Modelos tradicionais de desenvolvimento podem também beneficiar-se do modelo, contanto que eles abranjam desenvolvimento incremental, ou em *releases* curtos, e equipes multifuncionais.

O *Scrum Master*, ou algum papel similar ao de gerente de projetos em outras metodologias, serão os principais beneficiados pelo uso da abordagem, que fornecerá auxílio na tomada de decisão e apoio no planejamento das atividades das equipes.

3.5.2 Desempenho

A formulação apresenta equações complexas tanto para a função objetivo como para as restrições, não sendo muito trivial para simples algoritmos de força bruta. A formulação depende de uma extensa fonte de dados como entrada para a resolução do problema: os ganhos dos requisitos, a disponibilidade das equipes, o prazo das *sprints*, a lista de atividades do processo de desenvolvimento, o desempenho e custo de cada equipe para cada atividade de cada requisito.

A formulação foi executada em um *LP Solver*, uma aplicação para resolver problemas matemáticos de programação linear inteira dada uma formulação descrita. Quando aplicada a um *Solver*, seu desempenho pode variar de acordo com o volume e as características dos dados da entrada. Como as restrições são muito específicas, normalmente os *Solvers* possuem bom desempenho mesmo para instâncias grandes. Mais detalhes do desempenho da formulação em *Solver* são descritos no Capítulo 5.

3.5.3 Extensibilidade

Uma característica importante da estrutura deste problema é sua relação com outros problemas clássicos de Engenharia de Software e Gerenciamento de Projetos. A formulação não abrange diretamente todos os aspectos de interesse para os usuários deste problema. Desta forma, é ideal que outras formulações de interesse sejam estendidas ao Problema do Próximo *Release*. No caso específico da instância analisada neste trabalho, seria interessante acrescentar extensões que permitessem resolver problemas de alocação de equipes e estimação de requisitos.

A primeira seria fundamental para aprimorar possíveis heurísticas que buscassem as melhores atribuições para a *sprint*. Com esta extensão, seria possível obter resultados melhores para o Problema do Próximo *Release*, levando em consideração outras variáveis como produtividade, experiências e evolução das equipes. Já a estimação de custos de requisitos seria uma ferramenta útil para validar ou apoiar os valores fornecidos na entrada do problema, tornando-os mais precisos. Isso iria tornar as soluções do Problema do Próximo *Release* mais fidedignas à realidade. Durante Capítulo 5 será discutido em detalhes tal abordagem, quando uma prévia integração de modelos e técnicas será analisada.

Além desse tipo extensão, como a formulação é de estrutura modular similar ao trabalho de van den Akker et al. [2008], seria ideal a adição de restrições e variáveis para o tratamento de particularidades específicas do contexto de aplicação da formulação. Desta forma, a formulação apresentada seria uma base genérica para atender o suficiente diversas metodologias ágeis, enquanto novas extensões permitiriam especializar a

formulação. Por exemplo, para casos onde há multiprojetos no *release*, pode-se remover a restrição de ordem das atividades e incluir subconjuntos de requisitos para cada projeto (como pode ocorrer no próprio *Scrum*), ou caso haja uma estratégia *bottleneck* para controle de produção (e.g. *Work-In Progress*), pode-se adicionar essa restrição às atividades executadas sob dado intervalo de tempo, como ocorre no Kanban [Kniberg & Skarin, 2010].

3.5.4 Limitações

É necessário também destacar as limitações da formulação proposta, bem como os aspectos simplificados ou não tratados. Leve em consideração as extensões tratadas na subseção anterior, as quais não fazem parte do escopo inicial da formulação, mas são desejáveis em futuras modificações.

Primeiramente, não são tratadas neste problema as possíveis distrações ou interrupções que as equipes podem sofrer ao longo do desenvolvimento. Considere como distração quaisquer ocorrências que atrapalhem ou prolonguem a execução das atividades pelas equipes, como faltas, dificuldades técnicas, acidentes e retrabalho. Uma forma de relevar essa limitação é considerar um valor base adicional ao tempo e custo de desenvolvimento das atividades na entrada do problema.

Uma segunda limitação é a não maleabilidade ou sobreposição de membros das equipes durante as *sprints*. Dado certo prazo durante o planejamento, é possível que as equipes sofram algum reajuste durante o desenvolvimento, seja por trocas de membros, ausências não planejadas ou licenças para afastamento do trabalho. Neste caso, é possível que instâncias que envolvam muitas *sprints*, representando um longo prazo, sejam mais suscetíveis à esse problema. Para lidar com essa ocorrência, pode-se reexecutar a formulação a cada *sprint* atualizando as informações necessárias das equipes, requisitos e atividades.

Uma terceira limitação é a ausência de uma margem de erro nos valores e cálculos do problema. Principalmente pelo fato do problema exigir um grande volume de dados na entrada, é importante que a veracidade destes dados seja relevante, os quais poderiam ter uma confiança controlada via margem de erro ou associada a uma função de probabilidade.

Uma quarta limitação são as considerações e exigências do problema. A formulação base não é indicada para situações de multiprojetos dada a composição das equipes, atividades e requisitos. Nele são considerados que todas as equipes possuem uma disponibilidade em todas as *sprints*, sendo que estas equipes são multifuncionais, podendo executar qualquer atividade de qualquer requisito. É considerado que todos

os requisitos passam pelas mesmas atividades de desenvolvimento, devendo ser implementadas em ordem. O problema também exige que os custos do desenvolvimento sejam fornecidos previamente. Muitas vezes, esses dados não são facilmente obtidos ou destrinchados pelas empresas, podendo comprometer a veracidade dos dados. No entanto, algumas limitações podem ser relevadas ou anuladas fornecendo dados específicos na entrada do problema. Por exemplo, caso uma equipe não consiga exercer um tipo de atividade, bastaria informar em seus dados um tempo de desenvolvimento superior ao prazo da *sprint*. Desta forma, ela nunca seria considerada em uma solução.

Capítulo 4

Algoritmo baseado em GRASP Reativo

Neste capítulo será apresentado um algoritmo para solucionar a formulação proposta no capítulo anterior. Como o problema descrito pertence à classe NP-Difícil, o número de operações para encontrar soluções exatas cresce exponencialmente. No entanto, dadas as circunstâncias da aplicação, boas soluções conseguem satisfazer as necessidades dos interessados, não precisando arcar com a complexidade em se encontrar a melhor solução para um caso específico. Desta maneira, heurísticas e algoritmos aproximados se tornam válidos na busca de soluções para este problema. O objetivo de algoritmos que usam essas técnicas é fornecer soluções próximas da solução ótima do problema em complexidade polinomial, com um tempo de execução aceitável. O algoritmo apresentado neste capítulo foi implementado usando a meta-heurística GRASP Reativo.

A seção 4.1 apresenta uma descrição e contextualização de uso da meta-heurística escolhida. A seção 4.2 descreve a implementação do algoritmo, apresentando um pseudocódigo e a explicação da estrutura utilizada. Por fim, a seção 4.3 apresenta uma análise de complexidade do algoritmo, para estimar seu desempenho espacial e temporal.

4.1 A Meta-heurística GRASP Reativo

O GRASP Reativo é uma variação da meta-heurística original de Feo & Resende [1989], proposta inicialmente por Prais e Ribeiro para resolver um problema de decomposição de matrizes [Prais & Ribeiro, 2000]. Retomando a discussão de meta-heurísticas da seção 2.2, o GRASP é um algoritmo guloso que irá construir soluções iniciais de forma aleatória até satisfazer um critério de parada, no qual um parâmetro fornece o grau

de aleatoriedade da heurística. Esse parâmetro costuma ser fornecido na entrada do problema, sendo previamente definido. Indica-se então uso de alguma heurística de busca para aprimorar as soluções iniciais.

A ideia do GRASP Reativo é usar mais inteligentemente este parâmetro, procurando o grau de aleatoriedade ideal para um problema específico. Desta forma, este parâmetro de aleatoriedade é calculado durante a execução do algoritmo, sofrendo um autoajuste em cada iteração de acordo com a qualidade da última solução encontrada. Desta forma, o algoritmo tende a seguir o nível de aleatoriedade que vem construindo as melhores soluções iniciais [Prais & Ribeiro, 2000]. Assim como no GRASP original, a variação reativa também utiliza uma heurística de busca local para aprimorar as soluções iniciais.

O parâmetro de aleatoriedade, identificado normalmente por α, é definido segundo os possíveis valores de um conjunto A = $\{\alpha_1,\alpha_2,...,\alpha_n\}$. Inicialmente, todos estes valores tem a mesma probabilidade p_k de serem escolhidos. A cada iteração do GRASP um valor $\alpha_k \in A$ é escolhido para ser o parâmetro de aleatoriedade daquela iteração. As probabilidades p_k então são recalculadas de acordo com a qualidade da solução construída. Os valores α_k que produzem soluções melhores terão maior probabilidade de serem escolhidas nas próximas iterações [Kampke et al., 2011]. Ainda, pode-se utilizar um parâmetro de amplificação θ, que identifica o grau de impacto que a qualidade das soluções terão no cálculo das probabilidades p_k.

A figura 4.1 representa a estrutura geral de um GRASP Reativo. As linhas de 1 a 6 representam a inicialização das variáveis e parâmetros do algoritmo. *Score* é uma variável que representa o peso das escolhas de aleatoriedade durante as iterações. Dentro do *loop* principal (linhas 7 a 22) são realizadas, respectivamente, as etapas de construção da solução inicial (linha 9), busca local (linha 10), armazenamento da melhor solução encontrada (linhas 11 a 14) e a manipulação dos pesos da escolha aleatória, baseada na qualidade da solução retornada (linhas 15 a 20). As estratégias de construção, aprimoramento e manipulação de pesos ficam a critério do pesquisador.

4.2 Implementação

Esta seção descreve aspectos da implementação do algoritmo GRASP Reativo desenvolvido para o Problema do Próximo *Release*. O algoritmo foi implementado utilizando puramente a linguagem C, por questões de simplicidade, conformidade e flexibilidade, permitindo seu uso e adaptação a outros futuros projetos, softwares, sistemas ou linguagens de programação.

Procedimento *GR* (*CritérioParada*)

1 $f_{min} \leftarrow +\infty$;
2 $iter \leftarrow 1$;
3 Defina $A = \{\alpha_1, \alpha_2, ..., \alpha_v\}$
4 **para** $k \leftarrow 1$ *até* v **faça**
5 $count[k] \leftarrow 0$; $score[k] \leftarrow 0$; $p_k \leftarrow 1/v$;
6 **fim-para;**
7 **enquanto não** *CritérioParada* **faça**
8 $\alpha \leftarrow$ Selecione $\alpha_k \in A$ com probabilidade de escolha p_k;
9 $s_1 \leftarrow$ Construção_Solução(α);
10 $s_2 \leftarrow$ Busca_Local(s_1);
11 **se** $f(s_2) < f_{min}$ **então**
12 $s \leftarrow s_2$;
13 $f_{min} \leftarrow f(s_2)$;
14 **fim-se**
15 $count[k] \leftarrow count[k] + 1$; $score[k] \leftarrow score[k] + f(s_2)$;
16 **se** $iter \mod \gamma = 0$ **então**
17 $avg[k] \leftarrow score[k]/count[k]$ para todo $k \in \{1,2,...,v\}$;
18 $\sigma \leftarrow \Sigma\ (f_{min}/avg[k])^{\theta}$ para todo $k \in \{1,2,...,v\}$;
19 $p_k \leftarrow (f_{min}/avg[k])^{\theta}/\sigma$ para todo $k \in \{1,2,...,v\}$;
20 **fim-se**
21 $iter \leftarrow iter + 1$;
22 **fim-enquanto;**
23 retorne s;
fim *GR*;

Figura 4.1. Estrutura genérica de um GRASP Reativo [Kampke et al., 2011]

A implementação divide-se em partes menores, descritas detalhadamente nas subseções seguintes. A primeira etapa do algoritmo se propõe a inicializar variáveis, instanciar classes, ler os dados de entrada e realizar a ordenação desses dados. A segunda etapa consiste em aplicar a heurística construtiva do GRASP Reativo para criar soluções inicias. A terceira etapa consiste em aplicar a heurística de aprimoramento para melhorar a solução encontrada, comparando com a melhor solução encontrada até o momento. A última etapa consiste na apresentação dos resultados da execução do algoritmo.

O algoritmo 1 apresenta uma estrutura geral da implementação, com a breve descrição das etapas supracitadas. O algoritmo 2 apresenta o pseudocódigo da etapa de inicialização. Os algoritmos 3 e 4 apresentam as heurísticas utilizadas nesta implementação.

Algorithm 1 NRP Algorithm - Main Structure

```
1: procedure MAINPROCEDURE
2:
3:     ▷ inicializar variáveis, instânciar classes, ler entrada e ordenar vetores
4:     startProcedure;
5:
6:     while stopCriteria = false do
7:         ▷ aplicar heurística construtiva do GRASP Reativo
8:         runReactiveGRASP;
9:
10:        ▷ aplicar busca local
11:        runLocalSearch;
12:
13:        if actualSolution.getValue > betterSolution.getValue then
14:            changeBetterSolution(actualSolution);
15:    end-while
16:
17:    ▷ apresentar resultados
18:    showResults;
19: end-procedure
```

4.2.1 Atribuições

Uma **atribuição** é a principal estrutura criada para o algoritmo, baseada na formulação do capítulo anterior. Existem quatro aspectos chaves considerados na implementação do algoritmo: atividades, requisitos, equipes e *sprints*. A combinação entre esses elementos também remetem a um específico custo de tempo de execução da atividade e um esforço pela equipe. Essas possíveis combinações são representadas através de atribuições. Em outras palavras, uma atribuição corresponde à escolha de uma atividade de um requisito que será executada por uma equipe em uma *sprint*, com determinados custos de tempo e esforço. Baseado nessa estrutura, o algoritmo retorna como solução **uma lista com as melhores atribuições viáveis encontradas**.

A estrutura por atribuições pode ser representada de forma similar às variáveis de decisão da formulação, pela junção de quatro vetores: $x[a][r][e][p]$, que representam respectivamente a atividade, o requisito, a equipe e a *sprint* escolhidos. Por possuir

Requisitos	Sprints / Equipes / Atividades	P_1				$P_{...}$	$P_{\|P\|}$
		E_1	E_2	$E_{...}$	$E_{\|E\|}$	...	$E_{\|E\|}$
R_1	A_1	x_{1111}	x_{1121}	...	x_{11e1}	...	x_{11ep}
	A_2	x_{2111}	x_{2121}	...	x_{21e1}	...	x_{21ep}
	$A_{...}$	...	...	...	...	...	...
	$A_{\|A\|}$	x_{a111}	x_{a121}	...	x_{a1e1}	...	x_{a1ep}
$R_{...}$	...	...	...	...	...	...	...
$R_{\|R\|}$	$A_{\|A\|}$	x_{ar11}	x_{ar21}	...	x_{are1}	...	x_{arep}

Figura 4.2. Matriz que demonstra todas as possíveis atribuições de uma instância

quatro dimensões, a representação visual desta estrutura pode não ser trivial. Para facilitar o entendimento de futuras explanações, a figura 4.2 representa uma simplificação dessa estrutura em formato de matriz. Para cada instância é gerada uma matriz correspondente, representando todas as possíveis atribuições daquela instância. A solução do algoritmo, portanto, será uma lista composta de elementos dessa matriz.

4.2.2 Parametrização

Existem diversos parâmetros que conduzem o comportamento do algoritmo em tempo de execução. Os dois parâmetros mais importantes são o **grau de aleatoriedade** (α) e o **critério de parada**. Como descrito na seção 4.1, o parâmetro α é calculado durante a execução do algoritmo, tendendo a um valor que proporcione soluções melhores, enquanto o critério de parada determina quando a heurística deve parar de gerar novas soluções. Pode-se, por exemplo, determinar o critério de parada como um certo número de iterações, configurado previamente ou dinâmicamente para atender dado limite de tempo. Quanto maior o número de iterações, maior a probabilidade do GRASP retornar soluções melhores, dado que a cada iteração o parâmetro α se torna mais preciso.

O grau de aleatoriedade do parâmetro α é a principal essência do GRASP Reativo. Embora a sua utilização seja flexível, normalmente adota-se o comportamento onde o parâmetro possui um valor mínimo e um valor máximo, como por exemplo, de 1 à 100. Assim, o valor mínimo representaria a melhor escolha gulosa de uma atribuição viável, dado o critério guloso, enquanto no valor máximo o algoritmo faria esta escolha de

forma totalmente aleatória.

Nesta implementação, foi escolhido uma abordagem similar a esta do parágrafo anterior, onde o parâmetro α representa o percentual das atribuições que estarão presentes na aleatoriedade para cada escolha. Essas atribuições serão inseridas em uma **lista de candidatos**. Assim, por exemplo, se existem no problema 50 atribuições possíveis e o parâmetro α possuir valor 10 (equivalente a 10%), seria inserida na lista de candidatos as 5 melhores atribuições para aquela escolha gulosa do GRASP. Uma dessas 5 atribuições seria então aleatoriamente escolhidas para entrar na solução. Se α fosse 100 (equivalente a 100%), qualquer uma das 50 atribuições poderia ser sorteada, já que todas estariam presentes na lista de candidatos para aquela escolha gulosa.

Cada valor possível de α tem uma probabilidade de ser escolhido durante uma iteração do GRASP. Inicialmente, a probabilidade da escolha é a mesma para todas as possibilidades de α. Se um α escolhido conseguir construir uma solução melhor que a atual, aumenta-se a probabilidade deste α ser sorteado nas próximas iterações.

Este aumento na probabilidade é dimensionado por um **peso**, um outro parâmetro requisitado pelo algoritmo. O peso pode ser fixado previamente ou ser dinâmico. Neste último caso, o peso se torna maior para o quanto melhor é a solução construída se comparada à melhor solução anterior. O peso e o número de iterações são dois parâmetros diretamente relacionados ao qual eficazmente o algoritmo pode chegar a soluções boas. Um grande número de iterações e um pequeno peso torna o algoritmo mais preciso, porém mais lento em proporções ainda maiores.

Durante os experimentos foram analisadas diversas configurações para estes parâmetros. Concluiu-se que as configurações e o tamanho das instâncias são muito relevantes na determinação destes valores. Instâncias com muitas atribuições e requisitos funcionam melhor com pesos maiores e um parâmetro α com percentuais menos abrangentes, enquanto instâncias menores funcionam melhor com um grande número de iterações e uma aleatoriedade bem variável. Desta forma, foi adicionada à esta implementação uma lógica que trata o valor destes parâmetros em função do tamanho da entrada do problema.

4.2.3 Inicialização

Antes do uso das heurísticas gulosas e de busca local, há algumas etapas importantes na inicialização da implementação, representadas no pseudocódigo do Algoritmo 2. A primeira etapa é a criação das variáveis, parâmetros e instâncias das estruturas do algoritmo (linhas 3 e 4). Todo esse procedimento é trivial. A segunda etapa é a leitura dos dados de entrada da instância, que contém informações das atividades, requisitos,

equipes, *sprints* e atribuições (linha 5). De forma opcional, é possível também informar orientações ou parâmetros específicos para o grau de aleatoriedade, pesos e número de iterações do GRASP. Para a implementação testada foi utilizado um simples formato de lista em texto. A terceira etapa da inicialização é a ordenação das atribuições sob prescrição do critério guloso, o procedimento de maior complexidade desta parte do algoritmo (linhas 7 a 17).

Algorithm 2 NRP Algorithm - Initialization

```
1:  procedure startProcedure
2:
3:      readVariables; ▷ cria e inicializa variaveis
4:      readParameters; ▷ lê parâmetros para o GRASP
5:      readInput; ▷ lê os dados de entrada (atribuições no formato x[a][r][e][p])
6:
7:      if params.getHeuristic = CBR then
8:          ▷ calcula o "greedy value"das atribuições por custo-benefício de requisito
9:          calculateGreedyValue(CBR);
10:     else
11:         ▷ calcula o "greedy value"das atribuições por custo-benefício de atribuição
12:         calculateGreedyValue(CBA);
13:     end-if
14:
15:     ▷ ordena as atribuições pelo "greedy value"utilizando algum algoritmo de ordenação
16:     sortAlocations;
17:     setInitialWeight; ▷ peso 1 para cada atribuição como padrão
18: end-procedure
```

Para a ordenação, deverá ser calculado antes um determinado valor que será utilizado como critério a definir as melhores atribuições, chamado normalmente de critério guloso ou *greedy choice property*. Foram analisados em experimentos dois critérios gulosos diferentes para serem submetidos à ordenação:

- Melhor custo-benefício por requisito (CBR), priorizando as atribuições dos requisitos com melhor custo-benefício entre o ganho gerado pelo requisito e o tempo e esforço médio de execução de suas atividades.

- Melhor custo-benefício por atribuição (CBA), priorizando as atribuições com melhor custo-benefíco individual entre o ganho gerado pelo seu requisito e o tempo e esforço de execução de sua atividade.

Em suma, o primeiro critério utiliza uma abordagem por média dos valores das atribuições, enquanto o segundo critério utiliza uma abordagem individual, considerando uma atribuição por vez.

No primeiro critério, a média de tempo e esforço para a implementação de um requisito será calculada levando em consideração todas as equipes e *sprints*. Para encontrar essa média, soma-se todo o tempo e esforço de todas as atribuições de um requisito, dividindo posteriormente pela quantidade de equipes. O ganho do requisito será então subtraído do esforço médio do mesmo, dividido pelo tempo médio de execução de suas atividades.

Considerando $x[a][r][e][p]$ como a atribuição da atividade a, do requisito r, pela equipe e durante a *sprint* p, o valor guloso (*greedy value*) do critério *"CBR"* é definido como:

$$greedyValue(x[a][r][e][p]) = (r_{value} - r_{average.effort}) \div (r_{average.timecost})$$

Ou, em notação matemática compatível com a formulação:

$$greedyValue(x[a][r][e][p]) = (v_r - \frac{1}{E}\sum_{a \in A}\sum_{e \in E} c_{are}) \div (\frac{1}{E}\sum_{a \in A}\sum_{e \in E} t_{are}) \quad , \forall r;$$

Uma vantagem deste critério é que, pelo fato de não considerar individualmente uma atribuição, todas as atribuições de um requisito possuirão o mesmo valor guloso. Portanto, serão vizinhos na lista de atribuições ordenada. Desta maneira, percebe-se que este critério prioriza a total implementação dos melhores requisitos. No entanto, uma clara desvantagem se deve ao fato de não ser levado em consideração as equipes destas atribuições. Desta forma, mesmo se o requisito apresentar uma boa média de tempo e esforço, suas atividades podem acabar sendo alocadas a uma equipe com performance ruim.

Já no segundo critério, os valores analisados são individuais, onde nenhuma média é calculada. Neste caso, considera-se somente subtrair o esforço de uma atribuição do ganho do requisito representado por ela, dividindo posteriormente pelo tempo de execução daquela atribuição.

Considerando $x[a][r][e][p]$ como a atribuição da atividade a, do requisito r, pela equipe e durante a *sprint* p, o valor guloso (*greedy value*) do critério *("CBA")* é definido como:

$$greedyValue(x[a][r][e][p]) = (r_{value} - x[a][r][e][p]_{effort}) \div (x[a][r][e][p]_{timecost})$$

Ou, em notação matemática compatível com a formulação:

$$greedyValue(x[a][r][e][p]) = (v_r - c_{are}) \div (t_{are}) \quad , \forall a; \forall r; \forall e;$$

Uma vantagem deste critério é que, pelo fato de considerar individualmente cada atribuição, as atribuições de melhor custo-benefício serão priorizadas, levando em consideração atividades, equipes e *sprints*, o que o outro critério não faz. Desta forma, garante-se que boas atribuições entrarão para a solução inicial. No entanto, uma clara deficiência deste critério é a não consideração da implementação total de um requisito, ou seja, é desejável que a solução execute todas as atividades de seus requisitos, para que seja contabilizado seu ganho. Neste critério, não há uma garantia para esta pendência, permitindo a execução das atividades de um requisito em momentos dispersos, possibilitando a não total implementação do requisito.

Após o cálculo desses valores, qualquer algoritmo de ordenação pode ser usado para organizar a lista de atribuições na melhor ordem, como Inserção, Seleção ou *Quicksort* (recomenda-se o último pela menor complexidade). Esta lista ordenada será usada então na fase construtiva, onde o GRASP fará as escolhas gulosas.

4.2.4 Heurística Construtiva

A segunda parte do algoritmo é a aplicação de heurísticas para criar soluções iniciais, representada pelo Algoritmo 3. O principal componente desta etapa é a meta-heurística do GRASP Reativo, descrito previamente. Após a execução dos procedimentos da inicialização, todos os elementos necessários para o GRASP já estão devidamente instanciados.

Inicia-se então um *loop* baseado no número de iterações previamente calculados ou informados na parametrização (linhas 3 a 6). Em cada *loop*, uma solução viável será criada baseada no critério guloso escolhido, ou, no pior caso, uma solução vazia. A solução inicial será então submetida a etapa de busca local, a fim de aprimorá-la e compará-la a melhor solução encontrada até então.

Para a criação da solução inicial, serão inseridas atribuições aleatórias de uma lista de candidatos, construídas a cada escolha da solução, guiadas pelo parâmetro α, pelo critério guloso e pelas restrições do problema (linhas 16 a 34). O algoritmo continua a inserir atribuições na solução enquanto houver candidatos viáveis (linha 16). O parâmetro α irá determinar o tamanho da lista de candidatos para esta solução do GRASP (linhas 9 a 13), sendo alterado a cada nova iteração. O critério guloso irá determinar a ordem em que as atribuições entram na lista de candidatos (linhas 19 e 20).

As restrições determinam se a inserção da atribuição na lista de candidatos é válida para aquele momento (linhas 22 a 25). Finalmente, com a lista de candidatos construída, uma atribuição desta lista será sorteada para ingressar na solução do GRASP (linha 34).

As restrições do algoritmo são as mesmas estipuladas na formulação do problema:

1. A possível adição da atribuição à solução não pode extrapolar o limite de prazo daquela *sprint*.

2. A possível adição da atribuição à solução não pode extrapolar a disponibilidade daquela equipe para aquela *sprint*.

3. Se alguma atribuição da solução em construção já possuir a execução da atividade a do requisito r, não será necessário adicionar outra atribuição com essa mesma configuração. Isto é, uma atividade de um requisito só pode ser implementada uma vez.

4. Só será adicionada uma atribuição da atividade a do requisito r à solução se já existir uma atribuição da atividade *a-1* na mesma, com exceção da primeira atividade. Isto é, as atividades de um requisito precisam ser executadas em sequência.

5. Após não ser mais possível inserir novas atribuições à solução, será calculado o ganho dos requisitos implementados e o esforço gasto para tal feito. Apenas serão contabilizados os ganhos dos requisitos que tiveram todas as atividades executadas.

Em síntese, a cada iteração do loop, para cada escolha gulosa de qual atribuição irá ser inserida na solução inicial desta iteração, uma lista de candidatos com atribuições viáveis é construída. Os elementos desta lista são verificados para certificar que não infrinjam nenhuma restrição do problema. Após a escolha gulosa, o ciclo se repete até que não haja mais candidatos. Esta solução inicial é então submetida a uma busca local e comparada à melhor solução até o momento.

4.2.5 Busca Local

É aconselhável o uso de uma heurística de aprimoramento para explorar as soluções construídas pelo GRASP (Algoritmo 4). Nesta implementação, foi utilizada uma busca local por *Hill Climbing*. A proposta desta estratégia é explorar a **vizinhança** da solução original, buscando soluções próximas com pequenas variações (movimentos)

Algorithm 3 NRP Algorithm - Constructive Heuristic

```
1: procedure runReactiveGRASP
2:
3:     if numIteration ≥ params.getStopParam then
4:         stopCriteria ← true;
5:     else
6:         numIteration++;
7:
8:     ▷ randomiza o percentual de α para a iteração (entre 0.001% e 100%)
9:     alpha ← random(sumWeight) ÷ allocationCount;
10:
11:    ▷ determina o tamanho da lista de candidatos, baseado no parâmetro α (candidadatesList.size ←
       allocationCount × α). O tamanho da lista varia entre 1 até o número total de atribuições (allocationCount)
12:    candidatesList.clearInstance;
13:    candidatesList.setSize(alpha);
14:
15:    ▷ escolhas gulosas do GRASP. Repete enquanto houver candidatos, ou seja, a solução é viável.
16:    while foundCandidates = true do
17:        ▷ constrói a lista de candidatos, enquanto houver escolhas viáveis ou até completar da lista de candidatos
18:        for i ← 1 to candidatesList.getSize do
19:            ▷ seja X o conjunto de todas as atribuições x[a][r][e][p] ordenadas pelo critério guloso
20:            for each x ∈ X do
21:                ▷ confere se a atribuição x é viável na solução. Isto é, caso sorteada para entrar na solução, ela
                   não poderá infringir nenhuma restrição do problema. Se for viável, ela entra para a lista de candidatos.
22:                if checkCandidate(x) = true then
23:                    candidatesList.Insert(x);
24:                    noCandidateFound ← false;
25:                end-if
26:            end-foreach
27:            if noCandidateFound = true then
28:                break;
29:        end-for
30:        if candidatesList.IsEmpty = true then
31:            foundCandidates ← false;
32:        else
33:            ▷ Sorteia um dos candidatos e insere na solução inicial
34:            actualSolution.Insert(pickRandomCandidate(candidatesList));
35:    end-while
36: end-procedure
```

da mesma, verificando se elas conseguem melhorar o resultado final, sem infringir as restrições do problema. O processo se repete até que nenhum aprimoramento consiga ser mais alcançado, chegando ao **ótimo local** da vizinhança (linha 4).

Algorithm 4 NRP Algorithm - Refinement Heuristic

```
1:  procedure runLocalSearch
2:
3:      ▷ Continua busca local enquanto encontrar soluções melhores
4:      while continueLocalSearch = true do
5:          continueLocalSearch ← false;
6:          ▷ compara as atribuições da solução, invertendo as equipes e averiguando se há melhor retorno.
7:          for i ← 1 to actualSolution.getSize do
8:              bestValue ← −1;
9:              bestAllocation ← −1;
10:             for j ← (i + 1) to actualSolution.getSize do
11:                 ▷ Verifica se o valor da solução ao trocar as equipes das atribuições "i"e "j"é melhor
12:                 newValue ← swapedTeamsValue(actualSolution[i], actualSolution[j]);
13:                 if newValue > 0 then
14:                     ▷ Caso sim, verifica se a troca não infringe nenhuma restrição do problema
15:                     if checkSwap(actualSolution[i], actualSolution[j]) = true then
16:                         ▷ Verifica se esta troca é o melhor retorno encontrado até então
17:                         if newValue > bestValue then
18:                             bestValue ← newValue;
19:                             bestAllocation ← j;
20:                         end-if
21:             end-for
22:             ▷ troca as atribuições que resultaram no melhor retorno
23:             if bestValue > 0 then
24:                 changeAllocation(i, bestAllocation);
25:                 continueLocalSearch ← true;
26:             end-if
27:         end-for
28:     end-while
29: end-procedure
```

No caso desta implementação, a determinação da vizinhança e as modificações feitas na solução estão relacionadas às equipes e seus esforços na execução das atividades. Esclarecendo a escolha desta estratégia, o tratamento entre as diferenças de tempo de execução das atividades, prazo de *sprints* e disponibilidade das equipes mostrou-se como um processo não trivial para a determinação de vizinhanças expressivas, dificultando o encontro de ótimos locais relevantes. Como a determinação da vizinhança poderia ficar comprometida, a procura por esforços menores para as atribuições da

solução inicial se mostrou uma estratégia muito mais eficiente na busca local, podendo ser comparada e atualizada mais vezes, sem extrapolar a complexidade do algoritmo.

Na prática, serão procuradas trocas entre equipes de forma a diminuir o esforço da execução das atividades, de forma com que o prazo das *sprints* e a disponibilidade das equipes estejam de acordo com seus limites (linhas 7 a 21). Assim, considere que durante a Busca Local o ganho de implementação dos requisitos não será alterado, somente o seu esforço, que conforme a formulação, é deduzido do ganho dos requisitos.

Existem diversas políticas de movimento que podem ser adotadas durante a busca, dentre as duas principais o **Primeiro Aprimorante** e o **Melhor Aprimorante**. Respectivamente, na primeira política, a busca local avança no momento em que for encontrada uma solução melhor do que a anterior, não buscando necessariamente todas as possibilidades viáveis para aquela situação. Embora isso possa afetar o resultado final da solução, há um ganho computacional que permite que novas soluções sejam criadas e aprimoradas. Já na segunda política ocorre o contrário, onde todos os vizinhos são analisados previamente, movendo-se para o melhor [Talbi, 2009]. Ambas políticas foram implementadas e experimentadas previamente, com o Melhor Aprimorante demonstrando resultados um pouco melhores para instâncias grandes, sendo este representado no Algoritmo 4 (linhas 8 e 9, 17 a 20 e 23 a 26).

Resumindo os passos desta etapa, após a construção da solução inicial, ela é submetida à heurística de Busca Local, a qual irá encontrar o ótimo local para a vizinhança daquela solução. Se o resultado for melhor que qualquer solução encontrada anteriormente, a solução final, os pesos e parâmetro α são atualizados, e uma nova iteração do GRASP ocorrerá enquanto o critério de parada não for atingido.

4.2.6 Apresentação de Resultados

Como o propósito desta implementação foi a experimentação da formulação e heurísticas utilizadas, ela apresenta de forma textual somente as informações necessárias para a análise de resultados, como a melhor solução encontrada, estatísticas, variação de parâmetros e tempo de execução.

A melhor solução encontrada é apresentada em formato de lista, com a ordem das atribuições que devem ser aplicadas, expressando o ganho total dos requisitos implementados e o esforço gasto para tal. O resultado final é determinado pela diferença entre o ganho total pelo esforço total. Também é apresentado a quantidade de atribuições realizadas, o número de requisitos implementados e a ocupação das equipes e das *sprints* na solução (tempo utilizado da disponibilidade de cada equipe em cada *sprint*).

Como estatísticas, são apresentados a quantidade de soluções diferentes encon-

tradas, a performance da busca local, a média do parâmetro α durante a execução e o número de iterações. Estas estatísticas serão melhor analisadas no capítulo de Experimentos.

4.3 Análise de Complexidade

Esta seção discute a complexidade dos algoritmos implementados para este trabalho. Os dois procedimentos mais exigentes são as heurísticas de construção e refinamento (respectivamente Algoritmos 3 e 4), utilizadas repetidamente em função do critério de parada do algoritmo.

Durante a formulação do problema, foram apresentados diversos aspectos do problema que influenciam a complexidade do algoritmo, como as atividades, os requisitos, as equipes e as *sprints*. Desta forma, por motivos de simplicidade, é mais adequado estudar a complexidade dos algoritmos em função da estrutura criada para tal, ou seja, as atribuições. Considere como *"n"* o número de operações para ler todas as atribuições fornecidas na entrada do problema, em função de O(a+r+e+p).

Começando pela etapa de inicialização (Algoritmo 2), destaca-se os procedimentos de inicialização das variáveis, da leitura de parâmetros e dos dados de entrada, do cálculo do *greedy value*, da ordenação das atribuições e da pesagem inicial das atribuições. Com exceção da ordenação, todos esses procedimentos são triviais, com complexidade temporal constante ou linear em função do número de atribuições da entrada, ou seja, *O(1)* e *O(n)* respectivamente.

Algoritmos de ordenação presentes na literatura podem ser facilmente implementados em complexidade *O(n log n)* e $O(n^2)$. Por esclarecimento, o cálculo do *greedy value* ocorre de forma linear, em *O(n)*, uma vez que todos os valores necessários para seu cálculo são obtidos durante a leitura dos dados de entrada. Desta forma, este procedimento consiste somente na atribuição do valor em cada atribuição do problema, resultando em um total de *"n"* operações.

A etapa de Heurística Construtiva (Algoritmo 3) consiste de muitos procedimentos simples executados em *loops* sequenciais, que elevam a complexidade temporal desta etapa. Estes *loops* remetem às escolhas gulosas realizadas pelo GRASP e a construção da lista de candidatos, que ocorre antes de cada escolha realizada.

As escolhas gulosas do GRASP continuam enquanto houver candidatos, ou atribuições viáveis, para serem inseridas na solução inicial. A lista de candidatos é construída sob um *loop* que dura enquanto a lista não for preenchida totalmente por candidatos ou até o algoritmo não encontrar mais atribuições viáveis. O tamanho da lista de

candidatos é proporcional ao parâmetro α sorteado para aquela iteração do GRASP.

Desta forma, tendo em mente que $0 < \alpha \leq 100$, o pior caso ocorre quando α é máximo, possibilitando que todas as atribuições entrem na lista de candidatos, resultando em um tamanho máximo de $"n"$. Neste caso, como as atribuições estão ordenadas, elas seriam verificadas apenas uma vez para entrar na lista de candidatos, resultando em *O(n)* operações. Note que isso apenas ocorreria se todas as atribuições fossem viáveis, uma vez que o *loop* continua apenas enquanto acha candidatos. Pela formulação e restrições do problema isso jamais ocorreria, uma vez que isso infringiria obrigatoriamente a restrição da sequência de atividades. Assim, teríamos no pior caso uma complexidade de $O(re)$, uma vez que a lista de candidatos terá, em prática, no máximo $(r \times e)$ elementos.

O procedimento do parágrafo anterior é executado enquanto for inserido candidatos na solução. No pior caso, este *loop* seria executado a quantidade máxima de candidatos possíveis, ou seja, equivalente à solução que possuir a maior quantidade de atribuições viáveis. Isso ainda ocorreria somente na situação onde todos os requisitos conseguissem ser implementados na instância especificada. Como isso representa que todas as atividades de todos os requisitos foram executados, tem-se $(a \times r)$ atribuições presentes na solução.

Levando em consideração essas duas análises, conclui-se que a complexidade da heurística construtiva é, no pior caso, $O(ar^2e)$, ou de forma relaxada, $O(n^2)$. No entanto, perceba que para essa situação, tais configurações estáticas são anormais para a resolução do problema, uma vez que as restrições e os parâmetros controlam dinâmicamente o fluxo da heurística construtiva. Em outras palavras, alcançar esta situação requer uma instância com valores irrealísticos, restrições relaxadas e parâmetros fixos com aleatoriedade máxima constante. As funções que limpam a lista, configuram seu tamanho, inserem atribuições e verificam os candidatos são constantes ou lineares.

A etapa de Busca Local (Algoritmo 4) consiste na busca contínua de soluções melhores, onde a cada ciclo todas as atribuições da solução inicial são comparadas entre si, para que seja averiguado se uma suposta troca resulta em uma solução melhor. Como a implementação utiliza a estratégia do melhor aprimorante, toda atribuição da solução inicial será comparada com todas as outras atribuições da mesma solução, realizando a troca ao final de todas as comparações, de forma similar à um algoritmo de ordenação simples de Seleção. Como analisado anteriomente, a solução poderá ter no máximo $(a \times r)$ atribuições, resultando em até $\frac{ar(ar-1)}{2}$ operações. Assim, no pior caso, esse procedimento tem complexidade de $O(a^2r^2)$, ou de forma relaxada, $O(n^2)$.

Por fim, ressaltando a complexidade espacial, o algoritmo armazena as informações de acordo com a estrutura criada paras as atribuições e listas, sendo elas dinâmicas

ou estáticas. No pior caso, serão necessários somente a criação das variáveis, parâmetros e armazenamento das atribuições, resultando em complexidade linear $O(arep)$, ou relaxando, $O(n)$.

Como a complexidade das heurísticas prevalecem sobre os demais procedimentos, tem-se como complexidade temporal total do algoritmo $O(ar^2e + a^2r^2)$, ou de forma relaxada, $O(n^2)$, que demonstra a característica polinomial quadrática do algoritmo, em função das atribuições, viabilizando seu uso para instâncias ou projetos de grande porte.

Capítulo 5

Experimentos e Análises

A experimentação em Engenharia de Software é um tópico em desenvolvimento nos últimos 20 anos. Wohlin et al. [2000] é uma das principais referências nesta área, apresentando boas diretrizes para a elaboração de experimentos, análises e validações de estudos e pesquisas empíricas. A importância e necessidade da realização de experimentos se deve à aproximação desta área aos métodos científicos quanto à elaboração de modelos e teorias. Além disso, naturalmente a área de Engenharia de Software tem muitas similaridades com ciências sociais, onde ambas dependem da análise do comportamento humano [Wohlin et al., 2000]. Esse e outros trabalhos do período deram berço à área de Engenharia de Software Experimental.

Deve-se destacar, portanto, a necessidade destas práticas para averiguar quaisquer contribuições formuladas e propostas para a comunidade científica. No caso deste trabalho, deseja-se avaliar a formulação do Problema do Próximo *Release* em contexto ágil de desenvolvimento de software, no âmbito de estudos de otimização, e as heurísticas utilizadas para solucionar o problema em tempo polinomial, no âmbito de estudos de computação. Ambas diretrizes justificam os pilares da área de *Search Based Software Engineering*, a qual o trabalho propõe-se a contribuir.

Além de Wohlin, existem na literatura diversos trabalhos de outros autores envolvendo práticas de experimentação nesta área. Por exemplo, Armbrust [2003], baseado no modelo de Kellner et al. [1999], propõe a condução de simulações previamente às pesquisas empíricas tradicionais, sendo esse procedimento indicado para estudos da área de processos de software, gerenciamento ou operacional. Esta será a prática utilizada, sendo detalhada e justificada na seção seguinte.

As seções seguintes apresentam a elaboração, execução e resultados dos experimentos. Primeiramente, a seção 5.1 explica a prática de simulações na Engenharia de Software. A seção 5.2 descreve o planejamento, as configurações e as instâncias do expe-

rimento. A seção 5.3 demonstra os resultados obtidos e realiza uma análise dissertativa sobre eles. A seção 5.4 apresenta as limitações e ameaças à validade do experimento executado. Por fim, a seção 5.5 debate sobre a relação do problema apresentado com outros similares da área de Engenharia de Software, analisando pontos comuns entre eles e a possibilidade de integração entre modelos, de forma que os estudos das áreas se complementem.

5.1 Simulações em Pesquisas Empíricas

Antes de dissertar sobre a prática de simulações, é necessário ter conhecimento das principais pesquisas empíricas utilizadas na Engenharia de Software, que também possuem participação na metodologia apresentada por Armbrust [2003]. Wohlin et al. [2000] apresentam os seguintes tipos de estratégia de pesquisa:

- **Survey**: uma pesquisa de obtenção de dados quantitativos e qualitativos através de questionamentos à indivíduos da população alvo do objeto de estudo, normalmente aplicado através de entrevistas e questionários.

- **Estudo de Caso**: uma pesquisa de caráter qualitativo aplicada em ambiente real de uso da abordagem estudada, onde os indivíduos são observados com o mínimo controle ou interferência dos observadores.

- **Experimento**: uma pesquisa em ambiente controlado, normalmente em laboratório, onde os indivíduos seguem regras específicas, passando por altos níveis de controle e manipulação das variáveis envolvidas no processo.

Estes tipos de pesquisa são aplicadas mais facilmente em problemas com participação ativa de usuários, como uso de ferramentas e metodologias, independente do caráter qualitativo ou quantitativo de pesquisa, ou dos custos para execução do experimento. São os objetivos de pesquisa que irão determinar os tipos de experimentação mais indicados para um estudo [Wohlin et al., 2000].

Contudo, quando o contexto da avaliação refere-se a um modelo desenvolvido, desempenho operacional, treinamento ou gerenciamento estratégico, especialmente em pesquisas de caráter quantitativo, **Simulações** ou ambientes virtuais são alternativas válidas de experimentação [Armbrust, 2003].

Segundo Armbrust [2003], três benefícios podem se identificados através do uso de simulações: custo, tempo e conhecimento.

O benefício de custo ocorre pelo fato de experimentos convencionais da área de Engenharia de Software serem caros, necessitando de espaço, materiais e indíviduos (normalmente profissionais da área, o que eleva ainda mais o custo pelo fato das empresas oferecerem o seu tempo de trabalho). A condução de simulações diminui sensivelmente este custo.

O benefício de tempo é esperado pelo fato das simulações poderem ser conduzidas em qualquer tempo e velocidade esperados. Um experimento de gerenciamento de projeto poderia levar meses, mas uma simulação pode arbitrariamente durar o tempo necessário ou seguir qualquer ritmo desejado, com pausas ou retrocessos durante a condução.

O benefício de conhecimento complementa duas áreas: propósito de treinamentos e replicação em diferentes contextos. A simulação pode ser usada como ferramenta de treinamento para engenheiros de softwares para se adequarem a uma nova ferramenta ou tecnologia. Enquanto experimentos convencionais exigem muito mais tempo e custo para o treinamento, a simulação conseguirá fornecer *feedbacks* muito mais rápidos e sem nenhum risco de afetar o ambiente de experimentação, especialmente caso ele não seja controlado. A replicação de experimentos é muito mais simples quando se trata de uma simulação. Alterar as variáveis e contexto do experimento são mais baratos e rápidos de serem realizados desta forma. Aprender essas variações através das replicações simuladas pode economizar tempo e dinheiro comparado a experimentos em ambiente real.

Entretanto, as simulações não substituem os pápeis cumpridos pelos experimentos reais. Armbrust [2003] elabora um modelo de integração entre os dois tipos de pesquisa, onde as simulações são o primeiro passo para validação e direcionamento necessário para que os experimentos em ambiente real sejam mais consistentes. Em outras palavras, simulações são métodos práticos e de baixo custo que podem ser utilizadas para validar e obter dados fundamentais para auxiliar a condução de experimentos em laboratórios ou estudos de caso.

A figura 5.1 demonstra esse modelo, onde o papel das simulações de modelos é apoiar os estudos do objeto de pesquisa nos experimentos empíricos, e vice-versa. Isto é, experimentos tornando futuros modelos e simulações mais precisos.

Desta forma, antes da aplicação dessa formulação em experimentos controlados ou práticos de uso, foi idealizado a sua experimentação com instâncias virtuais e simuladas, a fim de certificar a veracidade do modelo, o desempenho dos algoritmos e qualidade das heurísticas implementadas.

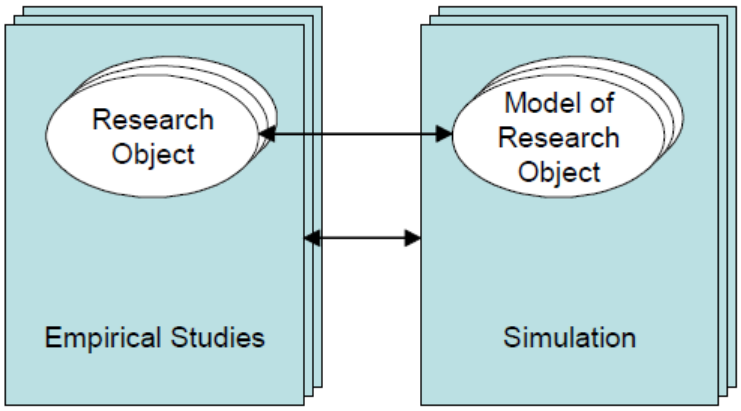

Figura 5.1. Relações entre experimentos virtuais e reais [Armbrust, 2003]

5.2 Configurações e Instâncias

Existe uma base de dados gratuita disponibilizada online referente ao trabalho de Xuan et al. [2012] que já foi utilizada em outros trabalhos do problema na literatura. No entanto, a adaptação dessas entradas à formulação proposta desconfiguravam complementamente suas características, não justificando seu uso como referência à essa nova formulação.

Dada esta situação, dedicou-se o enfoque na construção de instâncias com três características distintas: dados totalmente randômicos, parcialmente randômicos e sem randomização, sendo as últimas duas *ad-hoc*, a fim de analisar casos específicos do problema. As instâncias foram elaboradas em tabelas programadas para tais randomizações, sendo posteriormente adaptadas para o formato em que o algoritmo recebe as entradas e os parâmetros. A configuração dessas tabelas programadas e das instâncias não randomizadas foram realizadas por profissionais que utilizam ou já utilizaram a metodologia *Scrum* em ambiente de desenvolvimento. Primeiramente foi codificado um algoritmo para a geração dos casos de testes. Em acordo unânime, quatro membros decidiam os parâmetros de randomização quanto à duração e esforços das atividades e equipes. Foram também decididos previamente pelos membros os casos *ad-hoc* a serem explorados por instâncias não randomizadas.

Como descrito no capítulo anterior, o algoritmo trabalha os dados de entrada sob forma de atribuições, resultantes da combinação entre as atividades, requisitos, equipes e *sprints* do projeto. As entradas foram elaboradas entre um alcance de 100 até 60000 atribuições, com diversas composições dos aspectos citados a pouco. O prazo das *sprints*, a disponibilidade das equipes, os ganhos dos requisitos e os tempos/esforços de execução das atividades foram estipulados de forma que a randomização não permita

Tabela 5.1. Configurações das instâncias da simulação.

ID	*sprints*	*Equipes*	*Ativ.*	*Req.*	*Atribuições*	*ID*	*sprints*	*Equipes*	*Ativ.*	*Req.*	*Atribuições*
NRP01	1	2	5	10	100	NRP21	8	5	5	50	10000
NRP02	2	2	5	10	200	NRP22	5	5	5	90	11250
NRP03	2	2	5	15	300	NRP23	5	5	5	100	12500
NRP04	2	3	5	15	450	NRP24	6	5	5	90	13500
NRP05	2	3	5	20	600	NRP25	6	4	5	120	14400
NRP06	3	2	5	25	750	NRP26	6	5	5	100	15000
NRP07	3	3	5	20	900	NRP27	7	4	5	120	16800
NRP08	2	4	5	25	1000	NRP28	8	5	5	90	18000
NRP09	4	3	5	20	1200	NRP29	7	5	5	110	19250
NRP10	3	3	5	30	1350	NRP30	8	4	5	140	22400
NRP11	4	4	5	20	1600	NRP31	8	5	5	120	24000
NRP12	4	3	5	30	1800	NRP32	10	4	5	130	26000
NRP13	4	4	5	25	2000	NRP33	10	4	5	140	28000
NRP14	4	4	5	30	2400	NRP34	10	4	5	150	30000
NRP15	5	4	5	30	3000	NRP35	10	5	5	140	35000
NRP16	5	4	5	40	4000	NRP36	12	4	5	170	40800
NRP17	5	5	5	40	5000	NRP37	12	4	5	190	45600
NRP18	5	5	5	50	6250	NRP38	10	5	5	200	50000
NRP19	6	5	5	50	7500	NRP39	12	5	5	180	54000
NRP20	5	5	5	70	8750	NRP40	12	5	5	200	60000

algum comportamento irrealístico, como valores negativos, esforços superiores à ganhos de requisito e atribuições muito discrepantes. Foi considerado como processo de desenvolvimento de software das instâncias as seguintes atividades: especificação de requisitos, projeto/modelagem, implementação, validação/testes e implantação/manutenção.

As instâncias *ad-hoc* procuraram atender algumas situações comuns em ambiente real de uso da abordagem. Por exemplo, embora as equipes sejam multifuncionais, é provável que em contexto real uma equipe seja mais competente do que outra para executar determinado tipo de atividade. Desta forma, espera-se que o tempo de execução desta atividade seja menor para esta equipe. A mesma variação pode ocorrer com o esforço. Uma equipe mais competente pode executar as atividades em tempo menor, porém a um custo financeiro mais caro, ou seja, no contexto do problema, um esforço (gasto de recursos) maior. Outras situações comuns podem acontecer como troca de membros durante as *sprints*, o que afetaria em devida proporção a disponibilidade e eficiência da equipe. Cada instância *ad-hoc* se propôs a explorar uma destas situações.

O experimento foi aplicado em 40 instâncias diferentes, sendo metade destas totalmente randômicas e a outra metade *ad-hoc*, com randomização parcial ou nula. A Tabela 5.1 mostra algumas características de cada instância da simulação realizada. As 20 primeiras instâncias foram classificadas como *"básicas"*, apresentando menor complexidade e mais instâncias *ad-hoc*. As outras 20 instâncias foram classificadas como *"avançadas"* apresentando um número elevado de atribuições e exigindo mais desempenho por parte dos algoritmos. Outros dados estarão disponíveis na tabela de resultados do experimento.

O experimento foi executado em um computador com sistema operacional Windows 10 - 64 bits, com processador Intel Core i7 e 8GB de memória RAM. A formulação foi testada e solucionada através do *Solver* ILOG CPLEX 12.6.3, enquanto as heurísticas foram implementadas utilizando a linguagem C, compiladas via GCC/GNU, utilizando somente sua biblioteca padrão e sem interferência de recursos de IDE's. O tempo de execução tolerado para as instâncias foi de uma hora para ambos *Solver* e heurísticas.

5.3 Análise de Resultados

Serão apresentados nessa seção os resultados obtidos pelos experimentos através de gráficos e tabelas, bem com uma dissertação analisando e discutindo esses resultados. Por motivos de simplicidade e visibilidade, tome como referência as seguintes abreviações para os seguintes critérios gulosos apresentadas no Capítulo 4: **CBR**, representando a estratégia gulosa de *"Custo-benefício do requisito"* na construção da solução inicial pelo GRASP, e **CBA**, representando a estratégia gulosa de *"Custo-benefício da atribuição"* para o mesmo fim.

Antes de iniciar, alguns termos estatísticos devem ser definidos para análise dos gráficos e tabelas. O *gap* é um indicador que demonstra o distanciamento da solução do *Solver* em relação à solução ótima da relaxação linear do problema original, definido como: [gap = (limite superior - limite inferior) $\div$ limite superior $\times$ 100]. Similarmente, o *desvio* é o indicador que demonstra o distanciamento da solução da heurística em relação à solução do *Solver*. Logo, um desvio de 0% indica que a heurística encontrou a mesma solução do *Solver*, enquanto um desvio negativo indica que a heurística retornou uma solução melhor (logo, o *Solver* não encontrou a solução ótima).

Iniciando a discussão dos resultados, primeiramente será analisado a qualidade das soluções encontradas pelas heurísticas entre si. A figura 5.2 apresenta o desempenho dos critérios do GRASP quanto às soluções encontradas por ele referentes à solução do *Solver* em cada instância. A figura 5.3 representa a dispersão dos resultados de ambos critérios, em função do tamanho das instâncias testadas.

É perceptível na figura 5.2 a volatilidade do critério *"CBA"*, cujo propósito é priorizar as atribuições de melhor custo-benefício individual. Como discutido na subseção 4.2.3, este critério apresenta a clara desvantagem de não considerar a implementação total de um requisito. Assim, é provável que muitas atribuições escolhidas não levaram a um desfecho de um requisito, pelo simples motivo de existir uma ou mais atividades deste requisito com atribuições não boas o suficiente para serem priorizadas. A melhor

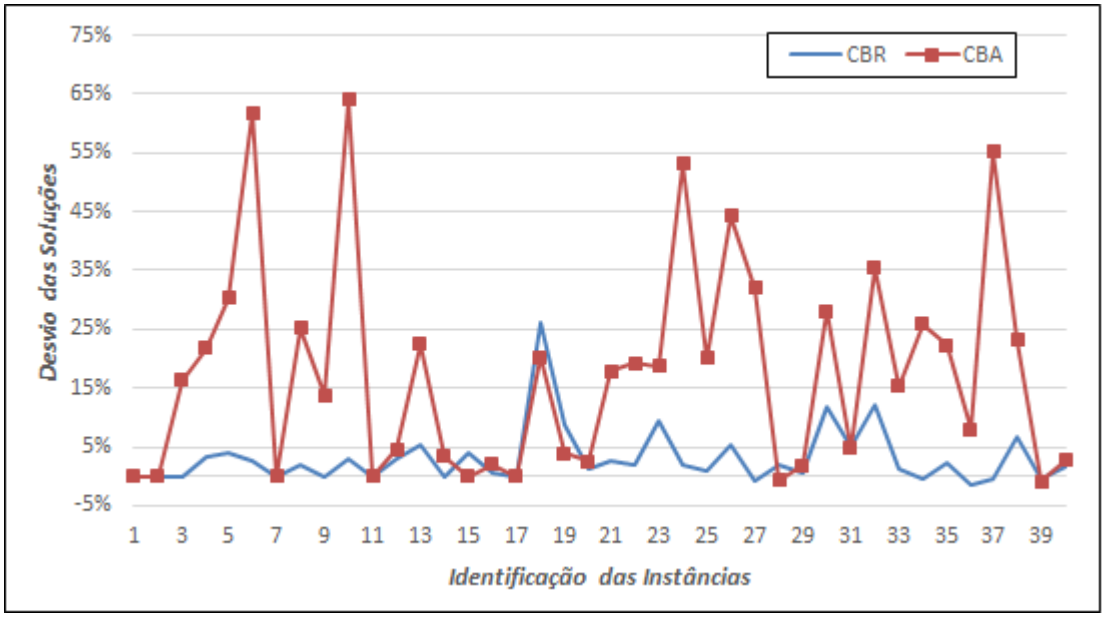

Figura 5.2. Resultado das soluções encontradas pelas heurísticas

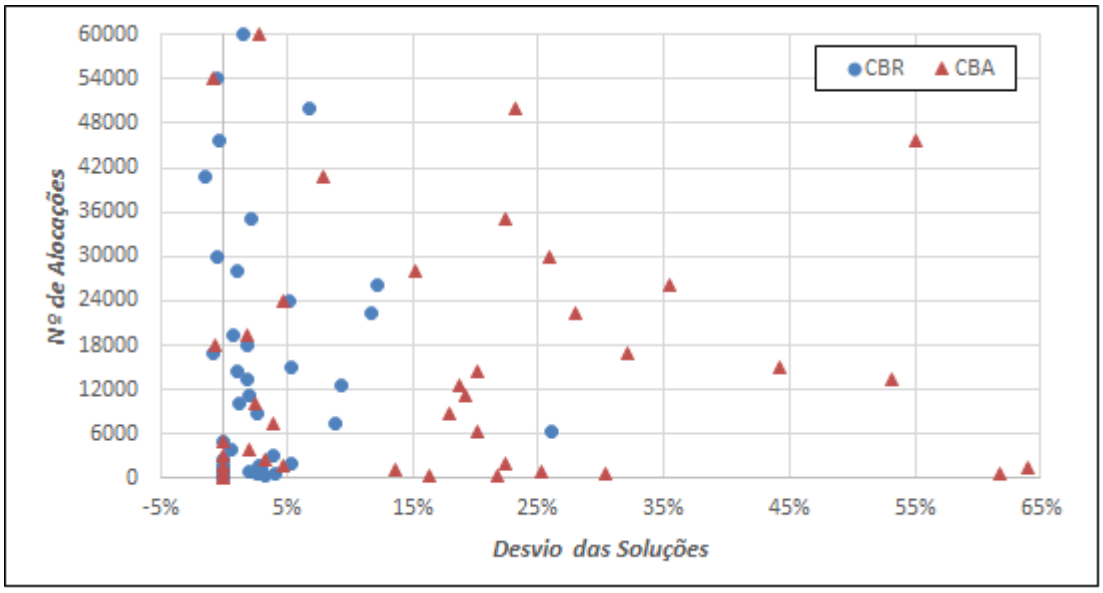

Figura 5.3. Dispersão das soluções encontradas pelas heurísticas

indicação deste critério mostrou-se para instâncias com extensos prazos para as *sprints* e disponilidade para as equipes. Nesta situação, praticamente todos os requisitos são implementados. Este critério, por priorizar individualmente as atribuições, conseguirá escolher as atribuições com melhor custo-benefício em relação ao tempo de execução e esforço das equipes (o que o *"CBR"* não consegue fazer, coincidentemente).

O critério *"CBA"* apresentou melhores resultados em somente 6 instâncias, com

pequena margem no desvio. Em todas as instâncias que ela encontrou o ótimo global, o critério *"CBR"* conseguiu fazer o mesmo. Entretanto, também observou-se que nestas instâncias básicas onde o ótimo global foi encontrado, o *"CBA"* conseguiu fazê-lo antes do que o *"CBR"* na maioria dos casos.

Já o critério *"CBR"* apresentou uma consistência notavelmente maior do que o critério *"CBA"*. Com um desvio menos volátil e abaixo do que o outro critério no conjunto destas 40 instâncias, é sensato dizer que priorizar os requisitos mais valiosos pela média de suas atribuições mostrou-se mais efetivo. A figura 5.3 deixa claro também o maior volume de soluções boas encontradas por este critério.

A clara vantagem do *"CBR"* discutida na subseção 4.2.3 é o fato dele conseguir priorizar o desfecho dos requisitos implementados, garantindo mais soluções candidatas a uma busca local relevante do que o *"CBA"*, independente da configuração da instância. A desvantagem é que esse critério não consegue priorizar as atribuições mais eficientes em termos de equipe, uma vez que utiliza-se como métrica de prioridade a média dos tempos e esforços de todas as equipes. Portanto, é provável que muitos requisitos bons sejam implementados, mas não da maneira mais eficiente. Outra desvantagem é que possíveis requisitos valiosos, mas com médias ruins, podem ser ignorados dependendo da gulosidade.

A média do parâmetro α também suporta essa vantagem do critério *"CBR"*. Em seu caso, o α médio das iterações do GRASP variavam entre 0,2% e 3% para instâncias pequenas e entre 0,005% e 0,02% para instâncias maiores. Isso representa que as melhores soluções eram construídas com listas de candidatos pequenas que, portanto, continham somente as melhores atribuições segundo critério guloso. De fato, o critério *"CBA"* apresentou um percentual maior do α médio, com variações entre 0,5% e 12% para instâncias pequenas e entre 0,05% e 0,8% para instâncias maiores, indicando que era necessário um maior grau de aleatoriedade na escolha das atribuições. Ressalta-se que houve algumas exceções para ambos critérios, onde algumas instâncias apresentaram um α médio fora dos intervalos estipulados há pouco.

É importante salientar resultados insatisfatórios ou comportamento errôneo durante a condução do experimento. Dentre as instâncias, destaca-se a *[NRP18]* que apresentou um *desvio* elevado para ambos critérios. Analisando detalhes da solução encontrada com a solução ótima da instância, percebe-se uma discrepância ainda maior entre os ganhos e os esforços ideais que explica algumas características da heurística.

Na solução ótima, a instância *[NRP18]* possui *ganho=1267*, *esforço=772* e *resultado=495*, enquanto a solução da heurística *"CBA"*, por exemplo, é composta por *ganho=1538*, *esforço=1143* e *resultado=395*. Logo, embora a heurística retorne uma

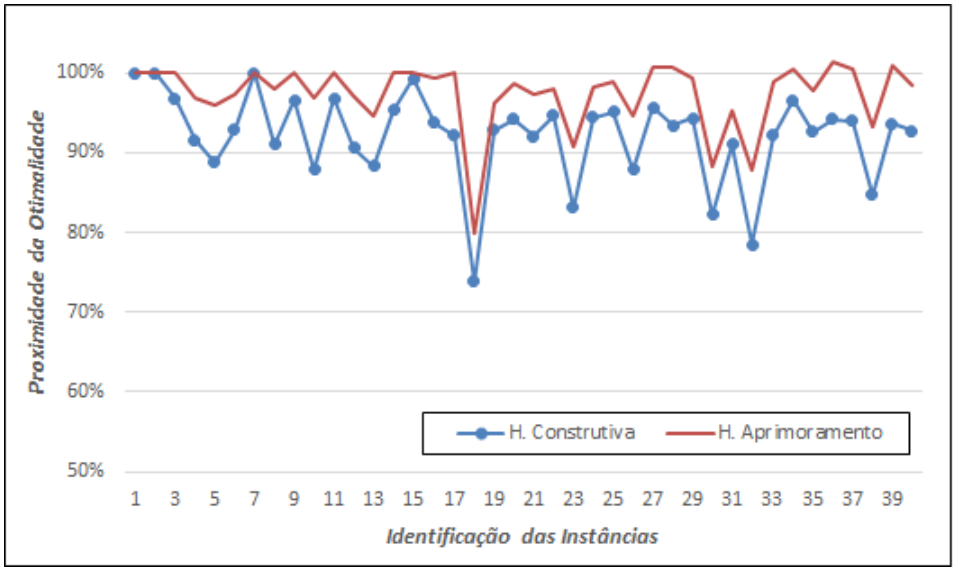

Figura 5.4. Proximidade do ótimo obtido pelas heurísticas construtivas e de aprimoramento

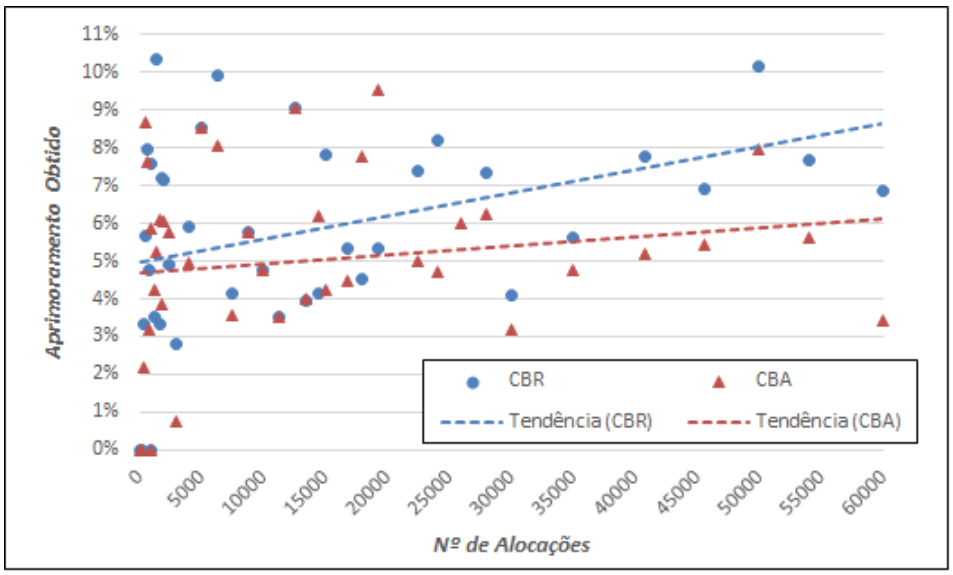

Figura 5.5. Dispersão dos aprimoramentos obtidos pela busca local

solução de maior ganho, ela vem ao preço de um esforço ainda maior. Dada observação, nota-se que a heurística credita mais um requisito pelo seu ganho do que o possível esforço gasto para sua implementação. Estes números deixam claro que a maior deficiência da heurística é a minimização dos esforços enquanto mantêm os ganhos de requisitos elevados e as restrições de equipes e *sprints* respeitadas.

Essa característica foi encontrada em outras instâncias durante a fase construtiva do algoritmo, criando soluções de alto ganho, mas com esforços elevados. Para

minimizar esse problema, a heurística de Busca Local vem a corrigir parcialmente esta deficiência. Discutida na subseção 4.2.4, a estratégia da Busca Local consiste na troca entre as atribuições da solução de forma que o esforço utilizado pelas equipes diminua. Em outras palavras, a Busca Local consegue ajustar as atribuições feitas pela heurística construtiva de forma que o esforço gasto seja menor, aumentando o seu custo-benefício.

Entenda que esta prática não trata este problema totalmente. A Busca Local otimiza os *"esforços"* das equipes, mas não o *"tempo de execução"* das atividades, que também faz parte do cálculo de custo-benefício. Otimizar o tempo de execução das atividades representaria uma possível maior acomodação de atribuições em função do prazo das *sprints* e disponibilidade das equipes. Porém, é improvável que a lacuna encontrada na solução consiga ser aproveitada por outros requisitos sem extrapolar a complexidade do algoritmo, dado que possivelmente seria necessário a reconstrução total da solução.

Os gráficos das figuras 5.4 e 5.5 demonstram o desempenho das heurísticas construtivas e de aprimoramento utilizadas. A figura 5.4 mostra a proximidade das soluções encontradas por ambas em relação ao ótimo global ou limite inferior encontrado. Este gráfico é útil para medir o impacto da busca local nas soluções iniciais encontradas. Já a figura 5.5 mostra o aprimoramento obtido na solução pelo uso da Busca Local quando usados os critérios *"CBR"* e *"CBA"* na construção da solução inicial, em função do tamanho das instâncias. Este gráfico é útil para identificar em qual critério a Busca Local foi mais impactante, traçando uma linha de tendência para avaliar esse aprimoramento.

Pelo simples fato do *"CBA"* priorizar individualmente uma atribuição, é mais provável que as soluções construídas sejam constituidas por atribuições eficientes, mas não necessariamente com os melhores requisitos. Nesta configuração, o ganho da Busca Local é visivelmente menor. Já no *"CBR"*, percebe-se um aprimoramento considerável nas soluções iniciais, com uma tendência crescente acompanhando o tamanho das instâncias.

Os resultados sumarizados nos gráficos anteriores estão disponibilizados nas Tabelas 5.2 e 5.3, permitindo uma base detalhada para possíveis replicações da simulação ou uma diferente análise dos valores absolutos das soluções obtidas. Nesta tabela estão as soluções encontradas pelas heurísticas utilizando a busca local e as soluções encontradas pelo *Solver*. O desvio permite avaliar o quão próximo a melhor solução da heurística está da solução encontrada pelo *Solver*.

Como descrito anteriormente, as 20 primeiras instâncias foram classificadas como *básicas* enquanto as demais foram classificadas como *avançadas*. Dentre as básicas, no período de uma hora, o *Solver* conseguiu encontrar a solução exata em

Tabela 5.2. Resultados da simulação nas instâncias básicas.

ID	*Relax*	*Solver*	*T(s)*	*Gap*	*CBR*	*CBA*	*T(s)*	*Desvio(Ag)*	*CBR*	*CBA*	*T(s)*	*Desvio(Pa)*
NRP01	26	26	2	0,00%	24	**26**	1	0,00%	**26**	**26**	55	0,00%
NRP02	77	77	2	0,00%	73	**77**	1	0,00%	**77**	**77**	321	0,00%
NRP03	61	61	2	0,00%	**57**	46	1	6,56%	**61**	51	401	0,00%
NRP04	124	124	3	0,00%	**113**	85	4	8,87%	**120**	97	295	3,23%
NRP05	145	145	3	0,00%	**131**	78	11	9,66%	**139**	101	306	4,14%
NRP06	149	149	3	0,00%	**136**	46	5	8,72%	**145**	57	130	2,68%
NRP07	170	170	2	0,00%	157	**170**	2	0,00%	**170**	**170**	5	0,00%
NRP08	194	194	6	0,00%	**167**	109	2	13,92%	**183**	145	211	5,67%
NRP09	226	226	3	0,00%	**218**	179	13	3,54%	**226**	195	378	0,00%
NRP10	197	197	38	0,00%	**174**	38	8	11,68%	**191**	71	417	3,05%
NRP11	286	286	2	0,00%	**275**	258	15	3,85%	**286**	**286**	509	0,00%
NRP12	279	279	148	0,00%	**257**	212	12	7,89%	**271**	266	296	2,87%
NRP13	316	316	442	0,00%	**284**	223	27	10,13%	**299**	245	756	5,38%
NRP14	330	330	6	0,00%	**303**	299	10	8,18%	**330**	319	645	0,00%
NRP15	390	390	6	0,00%	364	**380**	15	2,56%	375	**390**	195	0,00%
NRP16	614	614	405	0,00%	**570**	556	32	7,17%	**610**	602	574	0,65%
NRP17	741	741	3	0,00%	**734**	719	51	0,94%	**741**	**741**	191	0,00%
NRP18	495	495	4	0,00%	322	**367**	59	25,86%	366	**395**	1885	20,20%
NRP19	550	546	606	0,85%	469	**504**	81	7,69%	498	**525**	679	3,85%
NRP20	1041	1018	672	2,28%	**920**	887	54	9,63%	**1005**	992	814	1,28%

Tabela 5.3. Resultados da simulação nas instâncias avançadas.

ID	*Relax*	*Solver*	*T(s)*	*Gap*	*CBR*	*CBA*	*T(s)*	*Desvio(Ag)*	*CBR*	*CBA*	*T(s)*	*Desvio(Pa)*
NRP21	1017	960	772	5,66%	**878**	692	61	8,54%	**935**	788	568	2,60%
NRP22	1131	1072	635	5,29%	**941**	698	33	12,22%	**1051**	866	1021	1,96%
NRP23	1166	1123	481	3,76%	**985**	781	177	12,29%	**1018**	912	988	9,35%
NRP24	709	685	1855	3,44%	**622**	271	256	9,20%	**672**	321	1344	1,90%
NRP25	1459	1366	523	6,38%	**1194**	603	69	12,59%	**1352**	1091	1568	1,02%
NRP26	1144	1051	1788	8,14%	**890**	355	42	15,32%	**995**	586	1895	5,33%
NRP27	1094	1001	397	8,53%	**943**	391	90	5,79%	**1005**	680	2569	-0,40%
NRP28	2604	2569	461	1,37%	2352	**2551**	136	0,70%	2521	**2585**	2321	-0,62%
NRP29	2968	2879	510	3,02%	**2705**	2645	338	6,04%	**2860**	2825	1411	0,66%
NRP30	2773	2638	772	4,90%	**2198**	1698	45	16,68%	**2328**	1899	802	11,75%
NRP31	2983	2849	608	4,51%	**2525**	2475	125	11,37%	2702	**2715**	953	4,70%
NRP32	2612	2458	1212	5,91%	**2054**	1274	211	16,44%	**2158**	1588	633	12,21%
NRP33	2873	2716	515	5,47%	**2568**	2114	192	5,45%	**2686**	2301	1065	1,10%
NRP34	2710	2503	899	7,65%	**2272**	1633	95	9,23%	**2515**	1854	1789	-0,48%
NRP35	2802	2646	862	5,60%	**2287**	1715	55	13,57%	**2588**	2055	2129	2,19%
NRP36	1074	954	711	11,18%	**865**	597	65	9,33%	**968**	878	2432	-1,47%
NRP37	1559	1436	658	7,91%	**1329**	378	56	7,45%	**1442**	645	2780	-0,42%
NRP38	2459	2329	731	5,31%	**1938**	1301	201	16,79%	**2172**	1788	3298	6,74%
NRP39	1715	1607	1066	6,30%	**1421**	1325	92	11,57%	1615	**1621**	1985	-0,87%
NRP40	2354	2254	1285	4,28%	**2094**	1755	485	7,10%	**2218**	2192	3055	1,60%

18 ocasiões. Já nas instâncias avançadas, o *Solver* não conseguiu encontrar a solução exata em nenhuma ocasião nesse mesmo prazo. Retomando o que foi dito no início da seção, o *gap* irá indicar nessas instâncias o quão distante a solução ficou do limite da relaxação linear, enquanto o *desvio* irá indicar a distância entre as soluções heurísticas e a solução do *Solver*.

O experimento das heurísticas foi executado sob duas configurações principais: *agressiva* e *passiva* (identificadas nas tabelas como *"Ag"* e *"Pa"* respectivamente). A configuração **agressiva** é mais ousada quanto à distribuição de pesos (probabilida-

des) nas soluções criadas, favorecendo drasticamente uma região gulosa do espaço de soluções, ou seja, uma região que possui um conjunto de vizinhanças onde se gera gulosamente soluções melhores do que a atual. Ela consegue encontrar soluções boas ou razoáveis muito rapidamente, mas fica presa em regiões específicas do espaço de soluções. Isto é, como ela se restringe a uma região gulosa, ela não consegue explorar devidamente outras regiões com ótimos locais. Em suma, essa configuração é ineficaz no experimento de uma hora. Em nenhum experimento ela retornou soluções melhores após 10 minutos. No entanto, em intervalos curtos de tempo ela encontra soluções boas mais rapidamente do que o *Solver*, que por sua vez a supera em termos de qualidade de soluções ao longo do tempo.

Na configuração **passiva** ocorre o contrário, tornando o algoritmo mais independente e prezando mais a randomização. Nessa configuração, os pesos impactam menos nas escolhas e o algoritmo começa a procurar por outras regiões no espaço de soluções ao notar que os parâmetros atuais não estão gerando soluções melhores. Com isso, no intervalo de uma hora ele consegue gerar soluções melhores que a configuração agressiva, porém o *Solver* passa a encontrar soluções boas mais rapidamente. Para as instâncias pequenas, o algoritmo conseguiu encontrar soluções ótimas ou bem próximo do ótimo em uma hora enquanto o *Solver* o fez em poucos minutos. Para as instâncias maiores os resultados foram diversificados. Em termos de eficácia, nenhuma particularidade específica se tornou evidente, uma vez que algumas instâncias se aproximaram do ótimo enquanto em outras apresentaram um desvio considerável.

Os resultados das heurísticas foram variados, apresentando alguns aspectos ruins e outros bons. Em somente 6 instâncias as heurísticas conseguiram encontrar soluções melhores que o *Solver*, sendo todas elas avançadas e de maior complexidade. No entanto, os resultados do experimento não são suficientes para garantir que existe uma curva onde à medida que a complexidade aumenta o desempenho da heurística melhora sobre o *Solver*. Sugere-se que, embora a dimensão da instância influencie a qualidade das soluções, as características da instância possivelmente são o principal fator de impacto na eficácia das heurísticas, dado que as instâncias *ad-hoc* apresentaram resultados melhores para ambas heurísticas e *Solver*. Isto também leva a crer que o critério guloso pode não ser bom o suficiente para este problema, ou que o GRASP não é a heurística mais indicada para o mesmo.

Em suma, conclui-se pelos resultados obtidos, e demonstrados nas figuras anteriores, que a heurística usando o *"CBR"* possui uma melhor consistência na construção de soluções, possuindo uma sinergia relevante com a heurística de Busca Local. Ressalta-se, no entanto, que o critério *"CBA"* conseguiu encontrar soluções melhores em algumas situações, incluindo ótimos globais. Quando comparadas ao *Solver*, as

heurísticas apresentaram resultados variados, com desvios positivos e negativos nas instâncias de maior complexidade. Nas instâncias básicas o *Solver* foi evidentemente mais efetivo. Enquanto a configuração agressiva gerava inicialmente soluções melhores, somente a configuração passiva conseguia gerar soluções próximas ou equivalentes às soluções do *Solver*, ao custo de tempo notavelmente maior.

A execução da simulação com essas configurações resultou na construção de soluções com *gap* médio de 2,94% por parte do *Solver* e desvio médio de 2,79% (referente ao *Solver*) por parte das heurísticas. O parâmetro α com valores médios pequenos, principalmente durante o *"CBR"*, mostrou que o grau de aleatoriedade para alcançar soluções boas deve ser baixo, indicando que a gulosidade consegue construir soluções boas, mas não necessariamente próximas do ótimo.

5.4 Ameaças à Validade

Nenhum trabalho científico está assegurado de perfeição em sua formulação ou experimentação. Por envolver variáveis não programadas, estar sujeito à subjetividades, e por natureza ser suscetível à erros humanos ou deficiências técnicas ou tecnológicas, é esperado que todo estudo procure assegurar ao leitor as limitações encontradas. Essa seção busca esclarecer essas e possíveis erros de experimentação que possam ter ocorrido durante a condução das simulações e das análises realizadas.

A primeira ameaça à validade do trabalho parte da própria formulação do problema, cujas limitações já foram discutidas na subseção 3.5.4. Retomando o que foi discutido, essas limitações, que envolvem possíveis casos e variáveis não devidamente tratadas, podem compremeter a representação mais fidedigna do problema ou generalizá-lo em um nível maior do que o esperado.

Uma segunda ameaça à validade da experimentação é quanto a aplicação das simulações. Esta prática não substitui a experimentação real do problema, mais indicada para retratar os comportamentos e situações de difícil previsibilidade, mas serve como propósito de validar modelos e medir desempenhos de ferramentas e algoritmos. As simulações são práticas para fornecer dados úteis aos experimentos reais. Em outras palavras, uma análise do problema pode tornar-se incompleta sem a devida experimentação em ambiente real.

Uma terceira ameaça que pode comprometer a formulação e algoritmos idealizados é a ausência de outras meta-heurísticas clássicas na experimentação, para devida comparação de efetividade das soluções. Embora a literatura do problema já aborde diversos experimentos comparando-as, essa nova formulação pode permitir que

meta-heurísticas anteriores possam mostrar-se novamente eficientes para resolver esse problema.

Por fim, ameaças conclusivas também podem ocorrer quanto à análise e interpretação dos resultados, uma vez que não foi possível cumprir com exímia segurança as propriedades estatísticas que permitiriam a análise quantitativa mais apropriada dos dados coletados. Em outras palavras, dadas outras configurações ou instâncias os resultados podem ser diferentes.

Na discussão de trabalhos futuros serão sugeridos meios de tratar as devidas ameaças bem como possibilitar a expansão dos estudos realizados. O primeiro passo para essa expansão é a tentativa de integração deste problema com outros similares da área de Engenharia de Software, como discutido na próxima seção.

5.5 Integração a Outros Problemas de Engenharia de Software

Esta seção propõe estender o estudo do Problema do Próximo *Release* relacionando-o com outros problemas similares da Engenharia de Software, buscando encontrar pontos comuns entre eles e abrir uma discussão para integração entre modelos e técnicas que sejam mutualmente benéficas. Como a formulação proposta é de caráter modular e genérico para o contexto ágil, é ideal que haja estudos complementares para estendê-la, alterando restrições e variáveis para suprir particularidades específicas de aplicação. Nesta seção, serão previamente analisadas algumas possíveis extensões à formulação que tendem a corrigir possíveis limitações e desvantagens encontradas durante os experimentos do capítulo anterior.

Como dissertado na seção 2.1, *Search Based Software Engineering* é composto por diversas áreas de conhecimento envolvendo o desenvolvimento de software, como testes, manutenção e gerenciamento. De certa forma, muitos problemas estão interligados por fazerem parte de um determinado modelo ou processo de software. O Problema do Próximo *Release*, pertencente à área de gerenciamento de projetos, pode ser beneficiado com técnicas e formulações de outros problemas gerenciais e de requisitos, apresentados nas subseções seguintes.

Encontrados pontos de interesse ao problema, eles poderão ser futuramente integrados à formulação proposta ou uso dos algoritmos, ajudando a sua aplicação em contexto real.

5.5.1 Engenharia de Requisitos

A Especificação de Requisitos é uma atividade comum dentro de diversos processos de desenvolvimento de software. Sua prática é tão importante que a área de Engenharia de Requisitos existe com o propósito de estudar técnicas e modelos para seu exercício, identificando, analizando, documentando e validando requisitos para o desenvolvimento de software [Paetsch et al., 2003].

Ainda segundo Paetsch et al., a Engenharia de Requisitos é normalmente vista como incompatível aos métodos ágeis. A primeira é fortemente dependente de documentação para compartilhamento de conhecimento enquanto a segunda foca em colaboração face a face entre clientes e desenvolvedores para atingir tal objetivo. Ainda, a Engenharia de Requisitos é normalmente aplicada por práticas tradicionais como entrevistas, casos de uso e grupos focais para atingir esses objetivos de elicitação e análise.

No entanto, essas etapas podem ser encontradas nos processos ágeis através de diferentes técnicas dos diversos métodos discutidos na seção 2.4. Um exemplo citado neste trabalho é a prática do *Planning Poker* no Capítulo 3, utilizado comumente em *Scrum* e XP para auxiliar a elicitação e análise de requisitos. O Backlog também exerce um papel importante neste sentido. Como ele possui uma lista priorizada das características, funções, aprimoramentos e *bugs* a serem tratados, o Backlog pode ser comparado a uma documentação de requisitos incompleta, mas em evolução corrente [Paetsch et al., 2003].

Paetsch et al. ainda apresentam outras práticas utilizadas em métodos ágeis que se alinham às práticas da Engenharia de Requisitos como revisões de *sprints*, *brainstorms*, modelagens ágeis e envolvimento de clientes. No entanto, quanto ao foco de otimização, a área de *Search Based Software Engineering* tem como um dos campos mais promissores os estudos dos problemas de estimação de custos, que também podem ser aplicados à Especificação de Requisitos. Esses estudos tem potencialmente mais pareabilidade com o trabalho tratado.

5.5.2 Estimação de Custos

A área de Estimação de Custos tem boa representabilidade na literatura de *Search Based Software Engineering*, com centenas de artigos publicados em dezenas de periódicos nos últimos anos [Jorgensen & Shepperd, 2007]. No entanto, o início desses estudos tem origem bem anterior à consolidação do SBSE, entre a década de 1970 e 1980, com a proposta dos modelos SLIM, *Checkpoint*, PRICE-S e SEER, citados por Boehm et al. [2000a], precursores desta área. Neste trabalho, Boehm et al. apresentaram uma pes-

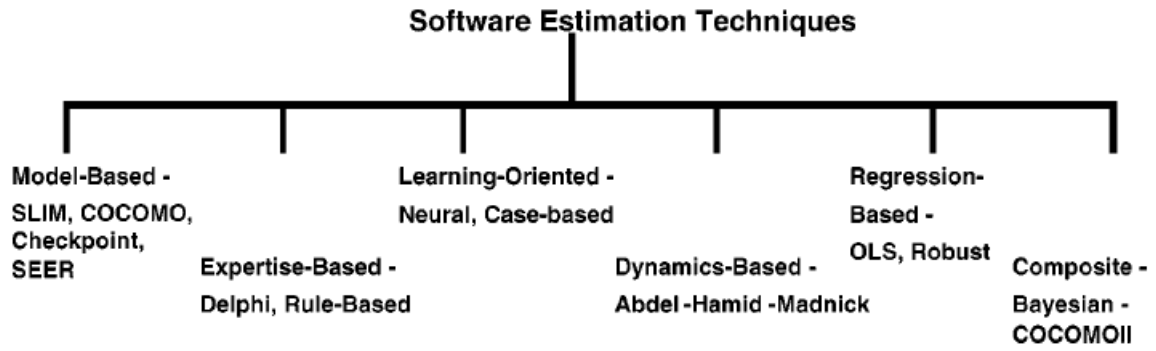

Figura 5.6. Categorias de métodos de estimação de custos [Boehm et al., 2000a]

quisa e revisão da literatura sobre diversas técnicas utilizadas na estimação de custos, muitas destas utilizadas até os dias atuais na formulação e modelagem de problemas.

Segundo Boehm et al. [2000a], os métodos de estimação são baseados e categorizados nos seguintes tipos: modelado, *expertise*, aprendizado, dinâmico, regressão e composto, sendo o último a categoria de métodos populares como o Bayesiano e COCOMO-II, proposto pelo próprio Boehm. A figura 5.6 demonstra a classificação desses métodos.

Praticamente qualquer um destes pode ser utilizado como ferramenta para a estimação de custos no problema deste trabalho. Porém, a popularidade do COCOMO-II e sua utilização em diversos trabalhos de otimização da literatura o tornam candidato ideal para análise. Embora esses métodos sejam indicados para métodos tradicionais de desenvolvimento, eles podem ser aplicados aos ágeis porque, retomando as declarações de Paetsch et al. [2003], em essência, os elementos tradicionais estão representandos sob alguma forma nas metodologias ágeis.

O método COCOMO-II (referente à *Constructive Cost Model*) é um modelo originado na década de 80 por Boehm e aprimorado em meados dos anos 90, com o propósito de visar as questões de desenvolvimento não sequencial, processos ágeis, reengenharia e reuso de software, e orientação à objeto [Boehm et al., 2000b]. O modelo possui três submodelos: Composição de Aplicações, *Design* Inicial e Pós-Arquitetura, que podem ser combinados de diversas maneiras para lidar com práticas atuais e futuras de desenvolvimento de software. O grande atrativo dos modelos COCOMO é a utilização independente de equações e parâmetros para estimação de valores, tornando-se um modelo adequado para ser aplicado aos problemas de otimização de Engenharia de Software.

Na prática, parte do modelo COCOMO poderia ser utilizado na formulação apresentada para tratar a estimação de custos dos requisitos e das atividades. Tomando

a ideia que a formulação deste trabalho utiliza as atribuições como estrutura principal (uma composição de atividade, requisito, equipe e *sprint*), seria adequado que o método fosse utilizado de forma a suprir a construção/definição das atribuições para a entrada do problema.

5.5.3 Alocação de Equipes

Um problema similar ao Problema do Próximo *Release*, e que de certa forma esta inserido no mesmo contexto, é o problema de Alocação de Equipes, outro problema clássico da área de gerenciamento e projetos de *Search Based Software Engineering*. A similaridade ocorre pelo fato que ambos os problemas têm como objetivo contextual a busca do melhor cronograma dado um projeto e prazo.

Como analisado na seção 5.3, uma deficiência perceptível na heurística de melhores resultados refere-se às equipes selecionadas para as atribuições. O critério de priorizar as atribuições de melhores requisitos não garante que elas serão alocadas às melhores equipes dados os custos e as disponibilidades. Técnicas de alocação podem beneficiar a formulação e os algoritmos implementados, em termos de eficiência e representação da realidade.

Torres [2010] realizou um trabalho onde formulou o problema de Alocação de Equipes e Cronogramas em Projetos utilizando o mesmo método COCOMO-II mencionado na subseção anterior para a estimação de custos e recursos das alocações realizadas. Em sua formulação, Torres utiliza os conceitos de habilidades (*skills*), experiências e *overhead* de comunicação para cálculo de produtividade. Quanto aos aspectos utilizados para buscar as melhores alocações, são considerados as tarefas (equivalentes às atividades), recursos (equivalentes às pessoas, ou equipes) e custos (equivalente ao esforço). São considerados também ao final os custos de atrasos e horas extras das equipes. No modelo, o método COCOMO utiliza os fatores pessoais como multiplicadores de esforço, os quais ajustam a estimação dos custos de acordo com parâmetros previamente conhecidos. Dessa forma, o cálculo de produtividade impactaria diretamente o tempo de execução do projeto.

No problema apresentado por Torres, deseja-se minimizar o tempo e o custo total do projeto, tornando-se um problema multiobjetivo. Adaptando-o à formulação deste trabalho, seria desejável somente a minimização do custo total do projeto, enquanto mantem-se a restrição de que o tempo de execução das atividades não supere o prazo das *sprints* e a disponibilidade das equipes.

Assim, em futuras adaptações à formulação apresentada neste trabalho, poderiam ser levadas em consideração as habilidades e experiências das equipes para o cálculo

de sua produtividade e adaptá-las às atribuições do problema, estimando o tempo e o esforço para suas execuções. Isso tornaria a formulação mais fundamentada e adaptável às diversas soluções apresentadas ao Problema de Alocação de Equipes na literatura.

Capítulo 6

Conclusão e Trabalhos Futuros

Este trabalho propôs contribuir à literatura com estudos referentes à problemas gerenciais da crescente área de *Search Based Software Engineering*, que busca alinhar otimização aos problemas de Engenharia de Software. O principal artefato desenvolvido neste trabalho foi a formulação do Problema do Próximo *Release* para o contexto ágil de desenvolvimento de software, em especial o método *Scrum*. A formulação elaborada é estruturada sob base de atribuições compostas por atividades, requisitos, equipes e *sprints*, com o objetivo de encontrar o cronograma de melhor valor dados o prazo das *sprints* e a disponibilidade das equipes. Uma outra contribuição do trabalho foram os experimentos realizados, comparando soluções obtidas via *Solvers* e heurísticas. O GRASP Reativo, que mostrou-se como meta-heurística construtiva mais eficiente para modelos anteriores do problema, foi utilizado no experimento base para a formulação proposta. Os experimentos mostraram uma eficácia razoável da heurística neste problema. As análises identificaram que a formulação e o algoritmo possuem pontos a serem aprimorados, como maior fundamentação na estimação de valores e melhores critérios gulosos nas atribuições. Já o *Solver* apresentou regularidade e bons resultados para todos os tipos de instância testados.

Ao fim de todo o trabalho realizado, dadas as referências literárias, formulações, modelos, experimentos e análises, pode-se concluir os seguintes questionamentos apresentados na introdução do documento:

R1: Os modelos de Ruhe & Saliu [2005], van den Akker et al. [2008] e Moura [2015] possuem elementos importantes para o contexto estudado, como conhecimentos e experiências, ganhos e custos, equipes, histórias e *backlog*. No entanto, nenhum deles apresenta os elementos das sucessivas iterações comuns aos métodos ágeis. Desta forma, representada por *sprints*, tal elemento foi inserido no modelo proposto, assim como a subdivisão de requisitos em atividades ou tarefas, pertencentes à um processo de

desenvolvimento de software. Esse novo modelo caracteriza o problema como Múltiplos *Releases*.

R2: Nos experimentos realizados, o *Solver* demonstrou um comportamento muito mais regular que as heurísticas, resultando em soluções com *gaps* médios menores às soluções das heurísticas. Embora o GRASP Reativo tenha conseguido soluções melhores em algumas instâncias, é certo dizer que, ao menos nesses experimentos, as heurísticas não foram suficientemente boas para contexto real de aplicação. Isto pode ser um indício que meta-heurísticas não sejam muito eficientes para este tipo de problema. Tais resultados podem ser justificados, em hipótese, pelo fato dos *Solvers* poderem tirar proveito das características estruturais do NRP, em relação aos diversos avanços bem sucedidos para solucionar o Problema da Mochila [Bagnall et al., 2001]. Em contrapartida, pode-se considerar como positivo o bom desempenho das técnicas exatas, o que favorece a replicação do problema em outros experimentos e em contexto real de aplicação.

6.1 Trabalhos Futuros

Dadas as limitações do trabalho, é de interesse a investigação de alternativas para encontrar melhorias aos artefatos apresentados, em especial o modelo e as heurísticas. Primeiramente, espera-se aprimorar a formulação apresentada com métodos e equações de estimação para melhor fundamentação, como o discutido método COCOMO-II. Espera-se também aproximar a formulação a aspectos encontrados em outros problemas similares, como o de Alocação de Equipes ou Cronogramas de Projetos.

Em segundo, espera-se trazer os artefatos elaborados para ambiente real de aplicação, para avaliação de gerentes e desenvolvedores de software. Estudos de caso e experimentos controlados em laboratórios são alternativas válidas após as observações obtidas via simulações. Esses especialistas também poderão opinar sobre a fidelidade da formulação ao contexto em que trabalham.

Por fim, novas meta-heurísticas podem ser analizadas para o problema, comparando-as de forma à encontrar melhores resultados em tempos aceitáveis. Embora o GRASP Reativo se mostrou eficiente para modelos anteriores, isto não foi comprovado na mesma magnitude para a formulação apresentada. Algoritmos baseados em meta-heurísticas evolucionárias, têmpera simulada e colônia de formigas mostraram-se também eficientes em modelos anteriores, podendo ser futuramente aplicadas a esta abordagem.

Referências Bibliográficas

Ambler, S. (2002). *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons.

Anderson, D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press Sequim, WA.

Armbrust, O. (2003). *Using empirical knowledge for software process simulation: A practical example*. Tese de doutorado, Diploma Thesis, University of Kaiserslautern, Germany.

Bagnall, A. J.; Rayward-Smith, V. J. & Whittley, I. M. (2001). The next release problem. *Information and Software Technology*, 43(14):883--890.

Beck, K. & Anders, C. (1999). Extreme programming explained: Embrace change addison-wesley. *Reading, MA*.

Beck, K.; Beedle, M.; Van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R. et al. (2001). Manifesto for agile software development.

Beedle, M.; Devos, M.; Sharon, Y.; Schwaber, K. & Sutherland, J. (1999). Scrum: An extension pattern language for hyperproductive software development. *Pattern Languages of Program Design*, 4:637--651.

Blum, C. & Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268--308.

Boehm, B.; Abts, C. & Chulani, S. (2000a). Software development cost estimation approaches - a survey. *Annals of Software Engineering*, 10(1-4):177--205.

Boehm, B. W.; Madachy, R.; Steece, B. et al. (2000b). *Software cost estimation with Cocomo II*. Prentice Hall PTR.

Cockburn, A. (2004). *Crystal Clear: A human-powered methodology for small teams*. Pearson Education.

Cohen, D.; Lindvall, M. & Costa, P. (2003). Agile software development. *DACS SOAR Report*, 11.

del Sagrado, J.; del Aguila, I. M. & Orellana, F. J. (2010a). Ant colony optimization for the next release problem: A comparative study. Em *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pp. 67--76. IEEE.

del Sagrado, J.; del Águila, I. M.; Orellana, F. J. & Túnez, S. (2010b). Requirements selection: Knowledge based optimization techniques for solving the next release problem. Em *6th Workshop on Knowledge Engineering and Software Engineering (KESE 2010)*, pp. 40--51.

Dorigo, M. (1992). Optimization, learning and natural algorithms. *Ph. D. Thesis, Politecnico di Milano, Italy*.

Edeki, C. (2013). Agile unified process. *International Journal of Computer Science*, 1(3):13--17.

Feo, T. A. & Resende, M. G. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67--71.

Glover, F. (1989). Tabu Search - Part I. *ORSA Journal on Computing*, 1(3):190--206.

Goldberg, D. E. & Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine learning*, 3(2-3):95--99.

Greer, D. & Ruhe, G. (2004). Software release planning: An evolutionary and iterative approach. *Information and Software Technology*, 46(4):243--253.

Harman, M. & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14):833--839.

Harman, M.; Mansouri, S. A. & Zhang, Y. (2009). Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, Kings College London, Tech. Rep. TR-09-03*.

Highsmith, J. (2013). *Adaptive software development: a collaborative approach to managing complex systems*. Addison-Wesley.

Jorgensen, M. & Shepperd, M. (2007). A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53.

Kampke, E. H.; Arroyo, J. E. C. & dos Santos, A. G. (2011). Grasp reativo e iterated local search aplicado ao problema de programação de tarefas em máquinas paralelas com tempos de preparação dependentes da sequência e de recursos. Em *XXXI Congresso da Sociedade Brasileira de Computação (CSBC'2011)*.

Kellner, M. I.; Madachy, R. J. & Raffo, D. M. (1999). Software process simulation modeling: Why? what? how? *Journal of Systems and Software*, 46(2):91--105.

Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. et al. (1983). Optimization by simmulated annealing. *Science*, 220(4598):671--680.

Kniberg, H. & Skarin, M. (2010). *Kanban and Scrum-making the most of both*. Lulu.

Kruchten, P. (2004). *The rational unified process: An introduction*. Addison-Wesley Professional.

Lenstra, J. K. (1997). *Local search in combinatorial optimization*. Princeton University Press.

Li, Z.; Harman, M. & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4):225--237.

Linhares, G. R. M.; Freitas, F.; Carmo, R.; Maia, C. & Souza, J. (2010). Aplicação do algoritmo GRASP Reativo para o problema do proximo release. Em *XLII Simpósio Brasileiro de Pesquisa Operacional (SBPO'2010)*.

Miller, W. & Spooner, D. L. (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223--226.

Moura, M. A. (2015). Algoritmos de otimização para a solução do problema do próximo release. Universidade Federal Rural de Pernambuco.

Paetsch, F.; Eberlein, A. & Maurer, F. (2003). Requirements engineering and agile software development. Em *Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 308--313. IEEE.

Palmer, S. R. & Felsing, M. (2001). *A practical guide to feature-driven development*. Pearson Education.

Peixoto, D.; Mateus, G. & Resende, R. (2013). Evaluation of the search-based optimization techniques to schedule and staff software projects: a systematic literature review. *Federal University of Minas Gerais, Brazil, Technical Report*.

Pitangueira, A. M.; Maciel, R. S. P.; de Oliveira Barros, M. & Andrade, A. S. (2013). A systematic review of software requirements selection and prioritization using SBSE approaches. Em *International Symposium on Search Based Software Engineering*, pp. 188--208. Springer.

Prais, M. & Ribeiro, C. C. (2000). Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12(3):164--176.

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

Rumbaugh, J.; Jacobson, I. & Booch, G. (2004). *Unified modeling language reference manual, the*. Pearson Higher Education.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Education, 9 edição.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons.

Torres, F. C. (2010). Alocação de equipes e desenvolvimento do cronograma em projetos de software utilizando otimização. Dissertação de mestrado, Universidade Federal de Minas Gerais.

van den Akker, M.; Brinkkemper, S.; Diepen, G. & Versendaal, J. (2008). Software product release planning through optimization and what-if analysis. *Information and Software Technology*, 50(1):101--111.

Victor, R. (2003). Iterative and incremental development: A brief history. *IEEE Computer Society*, 36(6):47--56.

Wells, D. (2009). Extreme programming rules. *Acessado: Maio*. http://www.extremeprogramming.org/rules.html.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B. & Wesslén, A. (2000). Experimentation in Software Engineering: An Introduction.

Xanthakis, S.; Ellis, C.; Skourlas, C.; Le Gall, A.; Katsikas, S. & Karapoulios, K. (1992). Application of genetic algorithms to software testing. Em *Proceedings of the 5th International Conference on Software Engineering and Applications*, pp. 625--636.

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

Zhang, Y. (2012). Repository of publications on search-based software engineering. *Acessado: Abril*, pp. 265--274. http://www.sebase.org/sbse/publications.

Zhang, Y.; Harman, M. & Mansouri, S. A. (2007). The multi-objective next release problem. Em *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1129--1137. ACM.