UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARIANE AFFONSO MEDEIROS

OTIMIZAÇÃO DE ARQUITETURA DE SOFTWARE UTILIZANDO SISTEMA DE COLÔNIA DE FORMIGAS

MONOGRAFIA

CAMPO MOURÃO

2016

MARIANE AFFONSO MEDEIROS

OTIMIZAÇÃO DE ARQUITETURA DE SOFTWARE UTILIZANDO SISTEMA DE COLÔNIA DE FORMIGAS

Trabalho de Conclusão de Curso de graduação apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso de Bacharelado em Ciência da Computação do Departamento Acadêmico de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marco Aurélio Graciotto Silva

Coorientador: Prof. Me. Filipe Roseiro Côgo

CAMPO MOURÃO

2016

Resumo

Affonso Medeiros, Mariane. Otimização de arquitetura de software utilizando Sistema de Colônia de Formigas. 2016. 58. f. Monografia (Curso de Bacharelado em Ciência da Computação), Universidade Tecnológica Federal do Paraná. Campo Mourão, 2016.

Contexto: O design arquitetural é uma fase crítica no desenvolvimento do software, pois decisões tomadas nesta fase têm um impacto significativo no custo e qualidade do sistema final. Um dos grandes problemas enfrentados pela arquitetura de software é o alto índice de mudança que esta sofre ao longo do desenvolvimento e o fato de ser uma tarefa dependente do engenheiro de software. Para resolver esse problema métodos de otimização arquitetural vêm sendo utilizados para propor diretrizes e recomendações a fim de identificar elementos arquiteturais, recuperar e otimizar arquiteturas. **Objetivo:** A fim de evitar mudanças arquiteturais ao longo do desenvolvimento e tornar a atividade menos dependente do arquiteto, este trabalho investiga a utilização da metaheurística colônia de formigas (ACO) para otimização arquitetural quanto a manutenibilidade de sistemas baseado em componentes. **Método:** Para utilizar o *Ant Colony Optimization* (ACO) para problemas de otimização arquitetural, primeiro definimos o tipo da arquitetura de entrada aceita pelo algoritmo, arquitetura baseada em componentes. Em seguida definimos o modelo de representação da arquitetura. Somente então definimos a forma de representação da arquitetura para o algoritmo. Após isso, determinamos a métrica de qualidade arquitetural usada para avaliar uma solução e a métrica para analisar o estilo arquitetural. **Resultados:** O método proposto foi avaliado a partir de experimentos, aplicando o algoritmo em um sistema real e observando se a otimização é satisfatória. Considerando a métrica de qualidade arquitetural adotada (MQ), o ACO obteve resultados satisfatórios, gerando soluções com valores MQ maiores que a arquitetura original. **Conclusões:** O método proposto mostrou-se eficiente para encontrar soluções boas a partir da métrica adotada.

Palavras-chaves: Engenharia de Software Baseada Em Busca, otimização arquitetural, otimização por colônia de formigas, métricas de qualidade arquitetural, estilo arquitetural

Abstract

Affonso Medeiros, Mariane. Software architecture optimization using Ant Colony Optimization. 2016. 58. f. Monograph (Undergradute Program in Computer Science), Federal University of Technology – Paraná. Campo Mourão, PR, Brazil, 2016.

Context: Architectural desgin is a critical phase of system development, because decisions taken in this phase have an important impact over the cost and quality of final system. One of the biggest problems faced by software architecture is the high cost of changes and the high dependency on the software engineer. To solve this problem, architectural optimization methods have been used to propose guidelines and recommendations to identify architectural elements, recover and optimize architectures. **Objective:** In order to avoid architectural changes during development and to make this activity less dependent of the software architect, this work investigates the utilization of ant colony methaheuristic to optimizate the architecture. **Method:** Firstly, the use of ACO to architectural optimization problems requires use to define the type of architecture accepted by the algorithm, which was component based architecture for our study. Then, we define a representation model of architecture. Finally, we determined the architectural quality metric used to evaluate the solution and the metric to analyze the architectural style. **Results:** The proposed method was evaluated using experiments with an existent software, observing if the architecture optimization was satisfactory. Considering the adopted quality metric, Modularization Quality (MQ), ACO achieved satisfactory results, generating solutions with higher MQ values than original architecture values. **Conclusions:** The proposed method proved efficient to find good solutions considering the adopted quality metric.

Keywords: Search based software engineering, architectural optimization, ant colony optimization, architectural quality metrics, architecture style

Lista de figuras

Lista de tabelas

Sumário

Capítulo 1

Introdução

A arquitetura de software é uma descrição do sistema que auxilia na compreensão do comportamento do software, servindo como modelo de como o sistema deve ser desenvolvido (Garlan, 2014). Além disso, a arquitetura é o primeiro modelo portador de atributos de qualidade do sistema, tais como desempenho, modificabilidade e segurança, cuja averiguação sem uma visão arquitetural é prejudicada. Desta forma, ao construir uma arquitetura efetiva, pode-se identificar riscos de design e mitigá-los nos primeiros estágios de desenvolvimento (SEI, 2015).

De fato, o design arquitetural é uma fase crítica no desenvolvimento do sistema, pois decisões tomadas nesta fase têm um impacto significante no custo e qualidade finais (Clements, 2002; Grunske, 2007). Por exemplo, um dos grandes problemas enfrentados pela arquitetura de software é o alto índice de mudança que esta sofre ao longo do desenvolvimento, mudanças essas que levam à degradação da qualidade da arquitetura e, eventualmente, a sua degeneração. Mudanças arquiteturais, depois que o software está implantado, tendem a gerar altos custos (Bosch, 2004) por causa da quantidade de retrabalho necessária e podem levar a desperdício de tempo (Grunske *et al.*, 2007). Dessa forma, seria interessante utilizar ferramentas que provessem ou indicassem mudanças na arquitetura existente a fim de melhorá-la. Outro problema enfrentado pelas equipes de desenvolvimento é a falta de padronização como, falta de documentação ou má implementação das regras da arquitetura, causando confusão entre a equipe quando é necessário compreendê-la.

Com a crescente demanda industrial por sistemas de software complexos, que possuem um alto índice de mudança nos requisitos de qualidade, o design da arquitetura do software se tornou uma importante atividade no desenvolvimento e pesquisa. Uma forma de tratar tais problemas é pela adoção de padrões e técnicas para especificação de arquiteturas. Por exemplo, diversas linguagens de descrição arquitetural (ADL) foram definidas com o intuito

de padronizar a arquitetura: Acme Project[1] (Garlan *et al.*, 2000), Darwin (Magee *et al.*, 1995) e ABC ADL (Huang *et al.*, 2006), dentre outras.

Entretanto apesar destes progressos, a construção de arquitetura com qualidade é altamente dependente do ser humano. Os arquitetos de software aprendem o ofício de construção arquitetural através de experiências próprias, vividas por diversos projetos passados e acabam não conseguindo ensinar a outros profissionais o que sabem (Garlan, 2000).

A fim de evitar mudanças arquiteturais ao longo do desenvolvimento e tornar a atividade menos dependente do arquiteto, investigam-se abordagens para otimização arquitetural. É interessante que os arquitetos de sistemas sejam auxiliados por métodos que automatizam a procura por uma arquitetura adequada levando em consideração métricas de qualidade e características do sistema. Estes métodos são chamados de métodos de otimização arquitetural (Aleti *et al.*, 2013).

Otimização arquitetural pode ser categorizada em um área de pesquisa chamada Engenharia de Software Baseada em Busca (ESBB) (Aleti *et al.*, 2013), proposta originalmente por Harman e Jones (2001). De modo geral, ESBB vem se mostrando maneira interessante de auxiliar os engenheiros de software nas suas tarefas (Harman *et al.*, 2012, 2014). A ideia central desta abordagem é a conversão de problemas da Engenharia de Software em problemas de busca. Especificamente quanto a arquiteturas, o problema de otimização arquitetural pode ser modelado como um problema de busca, considerando que a abordagem utiliza critérios pré-estabelecidos e características do sistema para alterar a arquitetura até que esta esteja em condição mais aceitável possível. Esta condição é medida através de métricas arquiteturais que avaliam a qualidade da arquitetura.

Métodos de otimização arquitetural vêm sendo utilizados para propor diretrizes e recomendações a fim de identificar elementos arquiteturais, recuperar arquiteturas de software e otimizar arquiteturas em relação a requisitos não funcionais, como desempenho, confiabilidade e custo, em tempo de projeto e de execução (Aleti *et al.*, 2013). Neste sentido, ferramentas que otimizam arquiteturas de software podem ser uma boa solução para auxiliar na manutenção e na melhora da qualidade de modelos arquiteturais.

Diversos trabalhos aplicam metaheurísticas no problema da otimização da arquitetura de software. Aleti *et al.* (2013) verificaram que as metaheurísticas mais usadas na literatura são Algoritmos Evolucionários, (Ramírez *et al.*, 2015; Colanzi; Vergilio, 2014; Vathsavayi *et al.*, 2013). Outras abordagens também foram encontradas na literatura, tal como Subida de Encosta (*Hill-climbing*) (Barros *et al.*, 2015), Busca Tabu (Mirandola *et al.*, 2014) e Enxame de Partículas (Hussain *et al.*, 2015).

Baseado na revisão bibliográfica feita, averiguamos que há poucos trabalhos que utilizam

[1] http://www.cs.cmu.edu/ acme/

Colônia de Formigas para abordar o problema de otimização arquitetural. Visto isto, gostaríamos de verificar como a metaheurística de colônia das formigas (ACO) se comporta para este tipo de problema, qual seu desempenho e a qualidade das soluções que ele fornece, considerando a métrica escolhida.

Para resolver um problema, algoritmos ACO utilizam uma colônia de formigas artificiais. Estas formigas se movimentam em um espaço de busca (que pode ser representado por um grafo) em várias direções a fim de construir uma solução para o problema em questão. Como as formigas reais, as formigas artificiais depositam feromônio pelo caminho que passam, com o objetivo de informar a outros membros da colônia que aquele caminho em questão já foi utilizado (testado). O feromônio depositado nos caminhos é utilizado pelas formigas durante o processo de construção da solução (muito feromônio influencia mais a formiga por determinado caminho do que pouco feromônio). A quantidade de feromônio depositado em um caminho é proporcional a qualidade da solução construída. Dessa forma os melhores caminhos (avaliados pela função objetivo) têm mais feromônio, aumentando a probabilidade deste caminho ser escolhido pelas próximas formigas. As formigas podem utilizar uma informação heuristica para auxiliar na construção da solução.

Este trabalho aplica a metaheuristica colônia de formigas para o problema de otimização de arquitetura de software. Dada uma arquitetura como entrada, a metaheurística deve otimizar a arquitetura a fim de obter uma solução otimizada em relação as métricas de qualidade. A motivação deste trabalho foi a construção de arquiteturas de software otimizadas e de qualidade arquitetural, de forma que não seja necessária a constante reparação da arquitetura pelo engenheiro de software, resultando em baixa manutenção arquitetural e melhor desenvolvimento do sistema. Para atingir este objetivo tivemos que definir os seguintes pontos:

- Tipo da arquitetura de software que a metaheurística considera;
- Forma de definição da arquitetura de software no ACO, ou seja, a estrutura de dados utilizada pelo algoritmo para representar a arquitetura;
- Métricas de arquitetura para avaliar a qualidade de cada solução.

O restante deste trabalho esta organizado da seguinte forma. O Capítulo 2 apresenta um referencial teórico da área de pesquisa, definindo arquitetura de software, engenharia de software baseada em busca e arquitetura de software com utilização de esforços da ESBB. O Capítulo 3 apresenta o algoritmo desenvolvido neste trabalho, que utiliza o algoritmo da colônia de formigas no contexto de otimização arquitetural. O Capítulo 4 apresenta os experimentos executados para avaliar a abordagem proposta, os resultados obtidos e uma discussão sobre os resultados. O Capítulo 5 apresenta as conclusões obtidas por este trabalho e trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo apresenta um referencial teórico a fim de embasar a construção deste trabalho. A Seção 2.1 trata sobre conceitos e características de arquitetura de software. Na Seção 2.2 é abordado o conceito de engenharia de software baseada em busca, apresentando definições desta área e aplicações na engenharia de software. Por fim a Seção 2.3 aborda a aplicação da engenharia de software baseada em busca para otimização de arquitetura de software.

2.1. Arquitetura de Software

A arquitetura de software apresenta um conjunto de decisões sobre a organização do sistema, como seleção de elementos estruturais e interfaces que irão compor o software. Essas informações compreendem parte dos requisitos funcionais que a arquitetura visa compreender. A arquitetura também deve determinar o comportamento e colaboração entre estes elementos, sendo estes os requisitos não funcionais (Booch *et al.*, 1999).

O conceito de arquitetura de software teve sua origem em uma pesquisa feita por Edsger Dijkstra em 1968 e David Parnas em 1971. Eles enfatizaram que a estrutura do sistema de software é importante e que determinar a estrutura correta é um processo crítico (Dijkstra, 1968; Parnas, 1971).

Nas primeiras décadas da engenharia de software, a arquitetura era considerada um artefato com resultados provisórios, isto pelo fato de ter um alto índice de mudanças ao longo do projeto. As descrições arquiteturais eram normalmente diagramas de linhas e raramente eram mantidas ao longo do projeto (Garlan, 2014). Escolhas arquiteturais eram feitas basicamente adaptando designs anteriores ao escopo atual. Bons arquitetos de software aprendiam seu ofício através de experiências com projetos passados e normalmente não eram

capazes de transmitir seus conhecimentos aos novos arquitetos. Era geralmente impossível analisar uma descrição da arquitetura do ponto de vista de coerência ou inferir propriedades não-triviais sobre esta (Garlan, 2014). Não havia maneira de verificar se uma determinada implementação respeitava os padrões arquiteturais.

No entanto, apesar desta informalidade, a descrição arquitetural se tornou um elemento importante no desenvolvimento do sistema. As pessoas começaram a notar o papel crítico que o *design* de arquitetura representava no sucesso do projeto. Passou-se então a adotar abordagens mais disciplinadas para determiná-las.

Com o crescente interesse da comunidade em arquitetura, pesquisadores começaram a observar certos princípios comuns dos projetos arquitetônicos (Rechtin, 1991) e passaram a estabelecer um vocabulário de trabalho para arquitetos de software (Garlan; Shaw, 1994) a fim de formalizar estas partes em comum e padronizar a forma de definição da arquitetura. Fornecedores de ferramentas começaram a dar suporte explícito para o projeto arquitetônico e projetos de linguagens começaram a considerar notações para representação arquitetônica (Garlan, 2000). A identificação e documentação de estilos arquiteturais permitem que outros engenheiros envolvidos no projeto ou ainda pessoas de fora da equipe de desenvolvimento possam adotar as estruturas arquitetônicas de outros projetos como ponto de partida (Garlan, 2014). Para tanto, a comunidade de arquitetura tem um paralelo com outras comunidades para reconhecer padrões de projeto bem documentados (Garlan, 2000). Um estilo arquitetônico especifica um vocabulário de *design*, restrições sobre a forma de uso do vocabulário e suposições sobre semânticas que o vocabulário usa (Garlan, 2000). Por exemplo, um estilo tubo-e-filtro (*piper-and-filter*) pode especificar vocabulário em que os componentes de processamento de dados são transformadores (filtros) e as interações são feitas através de fluxos de preservação da ordem (tubos). Restrições podem incluir a proibição de ciclos. Pressupostos semânticos podem incluir o fato de que os tubos são sem perdas e preservam a ordem dos dados escritos neles (Garlan, 2000).

Outros estilos comuns incluem arquiteturas de quadro-negro, cliente-servidor, centrada no repositório, baseada em evento, N-camadas, orientada a serviço, baseada em componentes, entre outros estilos. Cada estilo é apropriado para determinado fim. Por exemplo, um modelo de tubo-e-filtro provavelmente seria apropriado para uma aplicação de processamento de sinal, mas não para uma aplicação em que existe uma necessidade significativa de acesso simultâneo aos dados partilhados (Garlan, 2014). Há também a possibilidade da combinação de vários estilos para suprir a necessidade de uma arquitetura que precise de várias características de diversos estilos arquiteturais.

Devido à larga escala dos sistemas modernos, eles se tornam cada vez mais complexos e difíceis de serem mantidos, resultando em alto custo de desenvolvimento, baixa produtividade e qualidade duvidosa (Pour, 1998). Consequentemente há uma crescente demanda por

um paradigma de desenvolvimento eficiente. Um dos paradigmas de desenvolvimento mais promissor e vigorosamente investigado é o desenvolvimento baseado componentes (Cai *et al.*, 2000).

O desenvolvimento baseado em componentes visa a decomposição do sistema em componentes funcionais e lógicos com interfaces bem definidas, usadas para comunicação entre os componentes (Jifeng *et al.*, 2005). Componentes estão em um nível de abstração mais alto que objetos, portanto não compartilham estados e comunicam-se por troca de mensagens. As principais características do estilo baseado em componentes são (Microsoft, 2015):

- Reusabilidade: os componentes são desenvolvidos para que possam ser reutilizados em diferentes cenários e diferentes aplicações.
- Substituível: os componentes devem ser facilmente substituídos por outros similares sem causar grandes impactos no sistema.
- Nenhum contexto específico: os componentes são desenvolvidos para operar em diferentes ambientes e contextos. Informações específicas, como estado dos dados, devem ser fornecidas para os componentes ao invés de serem parte dele.
- Extensibilidade: um componente deve poder ser extensível para outros componentes a fim de prover novas funcionalidades.
- Encapsulamento: os componentes devem expor interfaces que permitem o uso de suas funcionalidades de maneira que não sejam revelados detalhes internos.
- Independência: os componentes são desenvolvidos a fim de ter o mínimo possível de dependência entre si. Portanto podem ser implantados em qualquer ambiente sem afetar outros componentes ou sistemas.

Uma arquitetura pode ser especificada em termos de UML (*Unified Modeling Language*), que é uma linguagem padrão de modelagem de software. A UML é uma linguagem gráfica para visualização, especificação e construção de documentação de artefatos de um sistema (Rumbaugh *et al.*, 2004) e permite a modelagem de arquiteturas baseadas em componentes em XML.

A Figura 2.1 mostra um exemplo de dois componentes modelados em UML 2.0. O componente *Checkout* é responsável por facilitar o pedido do cliente e requer do componente *CardProcessing* o preço gasto pelo cliente para que possa debitar ou creditar da conta do cliente. Os símbolos que ligam os dois componentes são os conectores que representam uma interface de comunicação entre os componentes. As interfaces de comunicação são duas: uma provida pelo componente *CardProcessing* e outra requerida pelo componente *Checkout*.

Apesar das vantagens da arquitetura baseada em componentes, existem limitações desta abordagem que não podem ser ignoradas. Por exemplo, evolução do sistema, migração e

compatibilidade são alguns do desafios encontrados quando se constrói uma arquitetura baseada em componentes (Vitharana, 2003).

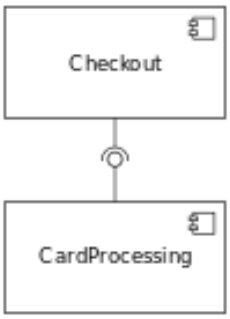

Figura 2.1. Exemplo de uma arquitetura baseada em componentes.
Fonte: Wikipedia `https://en.wikipedia.org/wiki/Component-based_software_engineering`

Independente da abordagem arquitetural que se use, seja baseada em componente ou não, é comum que engenheiros de software tenham que lidar com um grande número de requisitos não funcionais e de qualidade tais como: segurança, disponibilidade, confiabilidade, facilidade de manutenção e requisitos temporais de correção. Uma das principais dificuldades são os requisitos não funcionais que podem ter conflito um com o outro, ou seja, aumentar a qualidade de um requisito na arquitetura acaba diminuindo outro (Grunske, 2006).

Construir um sistema que cumpra todos os requisitos de qualidade necessários muitas vezes é impraticável. Como consequência, os engenheiros precisam considerar várias alternativas de arquitetura e identificar uma solução que satisfaça a maioria dos requisitos de qualidade. Este processo é chamado de análise de *trade-off*.

Para encontrar boas alternativas de *design*, que satisfaçam os requisitos de qualidade e auxiliem os arquitetos de sistemas na busca pela arquitetura mais adequada, surgiu a área de pesquisa de otimização arquitetural. A intenção básica dessa tarefa é encontrar um conjunto de especificações de arquiteturas que atendam a um conjunto de requisitos ou métricas de qualidade. Estas medidas de qualidade visam avaliar a solução sob alguma perspectiva. Por exemplo, caso o projeto espere obter alta coesão e baixo acoplamento, suas métricas de qualidade seriam coesão e acoplamento. Estas seriam as medidas avaliadas para determinar a qualidade da arquitetura. Desta forma, as métricas garantem que a otimização arquitetural melhorará características boas do sistema.

Diversas medidas de qualidade de arquitetura de software vêm sendo adotadas ao longo do tempo na engenharia de software. Estas técnicas determinam como medir características que podem ser utilizadas para comparar diferentes opções de arquiteturas e prover indicações de qual arquitetura satisfaz melhor os requisitos de qualidade, Seção 2.3.2 e Seção 3.5 discorrem sobre estas medidas.

2.2. Engenharia de Software Baseada em Busca (ESBB)

Assim como outras disciplinas de engenharia, a engenharia de software está concentrada em obter soluções ótimas ou o mais próximo do ótimo possível. Esta característica faz com que metaheurísticas de otimização baseada em busca sejam adequadas para serem aplicadas a engenharia de software (Harman; Jones, 2001).

Sendo assim, Harman e Jones (2001) propõem a Engenharia de Software Baseada em Busca. Esta abordagem aplica metaheurísticas a fim de propor melhores soluções em problemas de engenharia de software. Porém, para utilizar metaheurísticas na engenharia de software, é necessária a reformulação dos problemas para problemas de busca. Para fazer esta reformulação é necessário:

- Uma representação do problema que seja passível de manipulação simbólica. A representação do candidato é um processo crítico dos problemas de busca. As representações frequentemente usadas são números de ponto flutuante e código binário.
- Determinar uma função objetivo (definida em termos da representação). Esta função é a caracterização do que será considerada uma boa solução.
- Conjunto de operadores para manipulação dos elementos em análise. Os operadores fazem a modificação de um candidato a fim de gerar soluções diferentes do candidato atual.

Como pode ser constatado no trabalho de Harman *et al.* (2012), existe uma ampla variedade de áreas na engenharia de software onde técnicas de otimização baseada em busca podem ser utilizadas. As mais utilizadas são busca local, *simulated annealing* e algoritmos genéticos. Conforme apresentado na Figura 2.2 de Harman *et al.* (2012), observa-se os domínios de aplicação de ESBB, que consiste em 59% da literatura de ESBB concentrada em teste de software (Figura 2.2). A área de pesquisa de *Design* de ferramentas e técnicas consta só 10% da literatura de ESBB (Figura 2.2). Nesta área que se enquadra a otimização arquitetural abordada neste trabalho. Segundo Harman *et al.* (2012), *Design* de ferramentas e técnicas faz uso de técnicas de busca a fim de desenvolver melhores técnicas de *design* baseado em métricas.

2.3. ESBB para otimização de arquitetura de software

Neste trabalho foi realizado um mapeamento sistemático a fim de capturar os estudos da literatura que utilizam metaheurísticas para otimização de arquitetura. Para isso foram

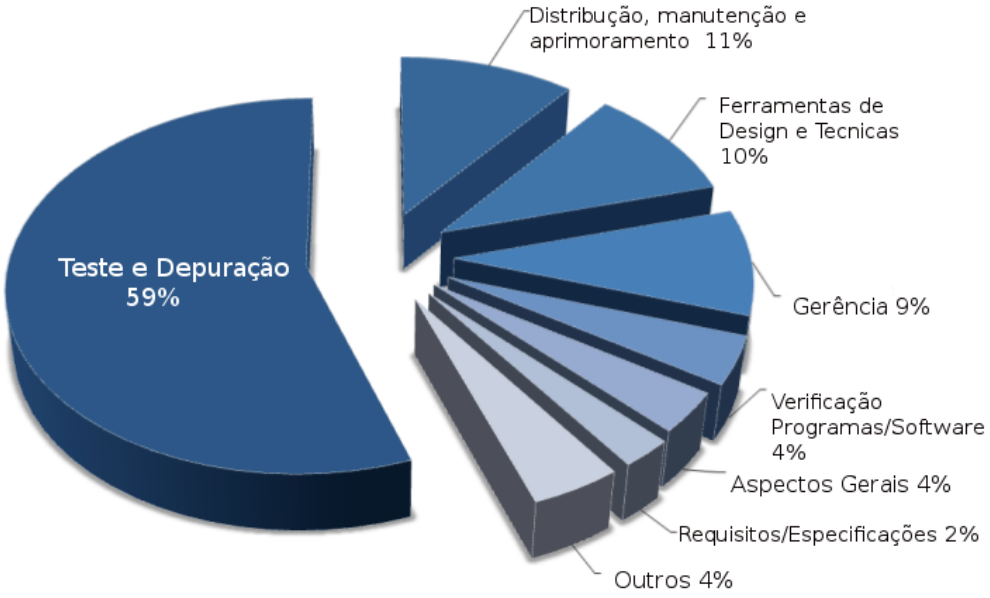

Figura 2.2. Áreas de pesquisa em ESBB.
Fonte: Harman *et al.* (2012)

utilizadas as bibliotecas digitais: *ACM library*, *IEEE Xplore*, *ScienceDirect* e *Scopus*. As buscas nas bibliotecas digitais foram feitas através de expressões de busca. Este mapeamento sistemático identificou os problemas arquiteturais, as metaheurísticas, as métricas e os métodos de avaliação, buscando responder as seguintes questões:

1. Como os problemas relacionados a arquitetura de software são descritos como problemas de busca?
2. Quais métricas relacionadas a arquitetura de software são consideradas nas intervenções propostas?
3. Como são avaliadas as soluções geradas pelas intervenções?

No total, foram encontrados 31 trabalhos e selecionados 12 artigos, o critério de seleção adotado foi responder as questões de pesquisa definidas acima. A maioria dos artigos utilizam algoritmo genético como metaheurística. Alguns estudos utilizando Busca Tabu, Subida de Encosta (*Hill-climbing*) e Enxame de Partículas. As métricas mais comuns foram coesão e acoplamento. A principal forma utilizada pelos autores para avaliar o desempenho e a qualidade das soluções sugeridas pela metaheuristica proposta foi através da comparação com outras abordagens. A Tabela 2.1 lista os trabalhos que serão relatados nas próximas seções, o algoritmo utilizado, a métrica e a forma de avaliação da abordagem proposta.

As seções seguintes discorrem sobre as metaheuristicas que foram encontradas na literatura pelo mapeamento sistemático, as métricas usadas pelas abordagens e a forma de avaliação que os artigos usaram para validar as propostas.

Tabela 2.1. Trabalhos encontrados pelo mapeamento sistemático.

Autor	Algoritmo	Métrica	Avaliação
Mirandola *et al.* (2014)	Busca Tabu	desempenho, segurança e custo da arquitetura	compara sua abordagem com métodos de busca aleatória
Barros *et al.* (2015)	Hill Climbing	número de pacotes, classes, atributos, métodos, método públicos e dependências,coesão e acoplamento	Um sistema
Martens *et al.* (2009), Martens *et al.* (2010)	Genético	desempenho, segurança e custo da arquitetura	Um sistema
Li *et al.* (2011)	Genético	utilização de processador, latência do fluxo de dados e o custo da arquitetura	Dois sistemas
Vathsavayi *et al.* (2013)	Genético	modificabilidade, eficiência e compreensividade da arquitetura	Um sistema
Ramírez *et al.* (2015)	Genético	coesão e acoplamento	Seis sistemas
Chhabra *et al.* (2015)	Genético	coesão e acoplamento	Oito sistemas
Hussain *et al.* (2015)	Enxame de Partículas	coesão e acoplamento	Três sistemas
Tawosi *et al.* (2015)	ACO	acoplamento método-atributo, acoplamento método-método, coesão de uma classe e coesão comportamental de uma classe	Quatro projetos
Simons e Smith (2013)	ACO	acoplamento do objeto e elegância do modelo arquitetural	compara com o desempenho do algoritmo genético

2.3.1. Metaheurísticas

Metaheuristicas são métodos de otimização que orquestram uma interação entre procedimentos de melhoria locais e estratégias de alto nível para criar processos capazes de escapar de ótimos locais e realizar uma busca robusta em um espaço de solução. (Glover; Kochenberger, 2003). Uma metaheuristica trabalha especialmente com informações incompletas, imperfeitas, com limitações computacionais (Bianchi *et al.*, 2008) ou com conjuntos de soluções muito grandes para serem totalmente amostrados. Uma metaheuristica pode fazer algumas suposições sobre o problema de otimização a ser resolvido e pode ser utilizada para diversos problemas (Blum; Roli, 2003).

Nas seções a seguir, são apresentadas as metaheuristicas encontradas na literatura para resolver o problema de otimização arquitetural. Veremos como essas metaheuristicas foram

usadas e adaptadas para o problema em questão.

2.3.1.1. Busca Local

Algoritmos de busca local iniciam sua execução a partir de uma solução inicial e iterativamente tentam trocar a solução atual por uma solução melhor, solução que está presente na vizinhança da solução atual (Blum; Roli, 2003). Dois algoritmos deste tipo foram encontrados no mapeamento sistemático, Busca Tabu e *Hill Climbing*.

Busca tabu é uma estratégia para resolver problemas de otimização combinatória (Glover, 1989). O objetivo é identificar a melhor decisão ou ação, a fim de maximizar ou minimizar alguma medida especificada. A principal característica da busca tabu é marcada pela implementação de uma memória adaptativa, onde são designadas estratégias especiais para explorar memória adaptativa (Glover; Laguna, 2013). Partindo de uma solução inicial, que pode ser determinada da forma desejada pelo programador, a busca se move a cada iteração para a melhor solução na vizinhança, nunca voltando para uma solução já visitada, pelo fato de armazenar em uma lista tabu todas as soluções já geradas. Esta lista permanece na memória por um determinado espaço de tempo ou por um determinado número de iterações.

Mirandola *et al.* (2014) utilizam Busca Tabu para otimização de uma arquitetura, tendo por objetivo adaptar uma arquitetura orientada a serviços, fazendo a escolha correta entre requisitos funcionais e não funcionais. A metaheurística seleciona o candidato que promove maior segurança e minimiza o custo de adaptação. O algoritmo recebe uma solução inicial, o qual é representado utilizando SCA-ASM *assemblies*, que consistem na definição de componentes, suas implementações e os nós onde são executados. SCA é uma técnica para modelagem formal de aspectos arquiteturais e comportamentais de aplicações. Os requisitos funcionais e não funcionais são avaliados por uma ferramenta externa. O algoritmo é baseado em uma lista tabu que evita que soluções já geradas sejam novamente visitadas. A cada iteração a vizinhança da solução atual é restringida somente às soluções que não estejam contidas na lista tabu. Então um conjunto de novos candidatos é obtido através de mudanças no candidato atual. Há dois métodos para gerar os novos candidatos. O primeiro método explora o espaço de busca através da aplicação de planos de adaptação do usuário, seleção de serviços e re-implantação de serviços. O segundo método usa um conhecimento específico dos padrões de *design* arquiteturais e técnicas para melhorar aspectos extras funcionais nos candidatos. O candidato selecionado é o que obtém o menor valor para a função objetivo, que é baseada em segurança e minimização dos custos. O candidato se torna então a base para a próxima geração, juntamente com as outras soluções contidas na população. Cada solução obtida é adicionada na lista tabu e uma das soluções que sempre permaneceu na lista é removida. O processo continua iterativamente até que o critério de parada, predefinido

como um número máximo de iterações, seja atingido e a melhor solução encontrada.

Outro algoritmo de busca local encontrado foi *Hill Climbing*. Esta metaheuristica tenta melhorar iterativamente o estado atual da solução através da função objetivo (Russell; Norvig, 2003). Esta metaheuristica possui uma solução inicial do problema abordado e procura melhorar esta solução, modificando sempre um elemento da solução atual. Se a mudança produzir uma solução melhor (segundo a função objetivo), uma mudança incremental é feita para a nova solução. Esse processo se repete até que o critério de parada seja atingido.

Barros *et al.* (2015) realizam um estudo de caso sobre o software *Apache Ant* com o objetivo de aplicar um algoritmo de otimização sobre a arquitetura do projeto para reduzir o número de dependências entre os pacotes e trazer á arquitetura a simplicidade original de quando o projeto se iniciou. Para isso, os autores utilizam a metaheurística *Hill Climbing*. O autor usou duas funções objetivo para resolver o problema: *Modularization Quality* (MQ), que busca reduzir o acoplamento e otimizar a coesão do projeto; e a função EVM, que verifica a robustez do grafo, que representa a arquitetura. No algoritmo proposto, dado um sistema com N classes, a busca inicia criando N agrupamentos e associando de forma aleatória cada classe a um agrupamento. A solução é representada por um vetor com uma entrada por classe. Cada entrada contém um valor inteiro no intervalo de $[0, N-1]$ que indica a qual agrupamento a classe está associada. Os operadores do algoritmo movem uma classe a um agrupamento distinto e calculam a função objetivo. Uma classe é selecionada e movida para qualquer agrupamento já avaliado. Depois que todos os agrupamentos tenham sido avaliados para a primeira classe, a busca segue com o procedimento considerando outras classes. Sempre que encontrar uma solução com resultado da função objetivo maior, a nova solução substitui a solução anterior. Se nenhum movimento gerar uma solução nova, então uma solução aleatória é gerada e a busca reinicia a partir desta. O critério de parada do algoritmo é um valor máximo predefinido para função objetivo.

2.3.1.2. Algoritmo Genético

Algoritmos genéticos compõem uma família de modelos computacionais inspirados em evolução. Este tipo de algoritmo codifica uma possível solução de um determinado problema em um cromossomo (representado por estruturas de dados) e aplica operadores de recombinações sobre estes cromossomos para otimizar uma informação (Whitley, 1994).

Os trabalhos de Martens *et al.* (2009, 2010) propõem um algoritmo genético multi-objetivo para arquitetura de software modelada com *Palladio Component Model* (PCM), uma linguagem de modelagem utilizada para previsão de desempenho e segurança de sistemas orientados a serviço e baseados em componentes. Para medir a segurança, transforma-se a modelagem do projeto em uma cadeia de Markov, que representa todos os caminhos possíveis

de execução considerando determinada arquitetura, juntamente com a probabilidade de cada caminho ser seguido. Há três estados especiais na cadeia de Markov: estado inicial, falha e sucesso. Analisando a cadeia de Markov é possível calcular a probabilidade do estado de sucesso ser encontrado a partir do estado inicial. A representação da arquitetura é dada por um vetor de números, onde cada opção de *design* é codificada como um gene. O algoritmo inicia com uma população inicial especificada pelo usuário. A evolução da população dá-se através da mutação e do cruzamento dos candidatos ou são aleatoriamente gerados. Em seguida os candidatos evoluídos são avaliados pelas métricas de desempenho e segurança. Após a fase de evolução, a população cresce, precisando passar pela fase de seleção para ser reduzida, restando apenas os elementos que foram mais bem avaliados pelas métricas. O critério de parada adotado é um número predefinido de iterações.

A metaheurística proposta por Li *et al.* (2011) utiliza como entrada uma arquitetura de software baseada em serviços. Com a arquitetura inicial, representada por um vetor, o algoritmo gera diversas alternativas de modelos arquiteturais por meio de operadores genéticos como cruzamento, mutação e seleção. Estes novos modelos gerados são avaliados pelos critérios de qualidade relacionados a utilização de processador, latência do fluxo de dados e custo.

O algoritmo genético proposto por Vathsavayi *et al.* (2013) converte a arquitetura inicial em um cromossomo. Quando a arquitetura é codificada para um cromossomo, algumas informações básicas são necessárias para cada operação, como: as operações que dependem da operação em questão, nome, tipo, frequência de uso, quantidade de parâmetros e sensibilidade a variação. Outra informação é sobre o local da operação na arquitetura, as classes que o elemento pertence, as interfaces que implementa, mensagens que despacha, as operações que chamam e os padrões de *design* que faz parte. Baseado na arquitetura inicial o algoritmo gera uma população inicial de arquiteturas com padrões de *design*. Então o algoritmo genético aplica os operadores de cruzamento e mutação para gerar novas arquiteturas. Por fim escolhe a melhor solução baseado nas métricas consideradas e a substitui na população. O processo continua até que a ultima geração seja obtida.

O algoritmo proposto por Ramírez *et al.* (2015) inicia com uma população aleatória extraída do modelo. O modelo de representação usado pelos autores para representar uma arquitetura baseada em componentes é descrita como um conjunto de componentes, interfaces e conectores, que são representados por uma estrutura de árvore. Em cada geração, características dos pais são selecionadas e sofrem mutação. Os novos candidatos são avaliados quanto às métricas impostas (coesão e acoplamento). Então uma estratégia de troca escolhe quais indivíduos irão sair da população para a entrada dos novos indivíduos. O critério de parada adotado é um valor preestabelecido da quantidade máxima de avaliações que o algoritmo efetuará.

Chhabra *et al.* (2015) propõem um método de otimização evolucionária multi-objetivo para agrupamento de módulos (pacotes) de software a fim de preservar componentes (classes) de núcleo de um pacote. Os autores utilizam duas funções objetivo para avaliar a qualidade de uma solução: MCA (*Maximizing Group Approach*) e ECA (*Equal Group-size Approach*). A representação da arquitetura de software é dado por um grafo de dependências, representado em forma de vetor, onde os componentes (classes) estão associados com o índice do vetor e os agrupamentos (pacotes) estão associados com os valores no vetor. Os autores estabelecem uma quantia máxima de execuções do algoritmo com critério de parada.

2.3.1.3. *Swarm Intelligence*

Algoritmos *Swarm Intelligence* (SI) são inspirados em uma população de agentes que interagem uns com os outros e com o ambiente. A inspiração deste tipo de abordagem veio da natureza, especialmente de sistemas biológicos (Blum; Li, 2008).

Hussain *et al.* (2015) propõem uma abordagem baseada na técnica de Otimização por Enxame de Partículas - OEP (*Particle Swarm Optimization*), uma metaheurística de busca baseada no comportamento biológico e social de espécies como pássaros e peixes. No OEP as possíveis soluções são partículas. Estas partículas sobrevoam o espaço do problema em busca da melhor posição encontrada por ela mesma ou pelas partículas vizinhas. Cada partícula possui uma posição e velocidade, inicializadas aleatoriamente. A cada iteração, a velocidade e posição das partículas são atualizadas. A OEP é composta por três passos: atualização da velocidade; atualização da posição das partículas e cálculo da função objetivo, que é obtida através das métricas de coesão e acoplamento, até que a solução desejada seja encontrada. A arquitetura de software é representada por uma matriz $k \times m$ na qual cada entrada da matriz pode ter o valor de 0 ou 1. Para uma partícula especifica P_i, se $P_i[r, c] = 1$ significa que a classe c está associada ao agrupamento r. Os autores estabelecem como critério de parada um valor máximo para a função objetivo ou um número máximo de iterações do algoritmo.

Mueller (2014) propõe um sistema automatizado para otimização de arquiteturas usando a abordagem de colônia das formigas. Mueller (2014) afirma que o espaço de busca por um *design* arquitetural cresce conforme n (número de componentes) aumenta, pois o particionamento da arquitetura de software em um conjunto de unidades é generalizado como o problema de particionamento (Harris *et al.*, 2008). Uma partição de um conjunto S é uma coleção de subconjuntos disjuntos de S do qual a união é S. Por exemplo, temos cinco alternativas para particionar o conjunto {a, b, c}. As alternativas são: {{a}, {b}, {c}}, {{a}, {b,c}}, {{b}, {a, c}}, {{c}, {a, b}}, {{a, b, c}} (Mueller, 2014). Levando estas considerações para o escopo do problema abordado neste trabalho, temos que os conjuntos são os componentes da arquitetura e dentro deles são combinadas diversas classes de forma a otimizar as métricas

consideradas.

Há diversas maneiras para particionar um conjunto de n elementos em k subconjuntos não vazios. Este particionamento pode ser definido pelo número de partição Stirling (*Stirling partition number*) $S(n, k)$, calculado com a seguinte fórmula:

$$S(n, k) = k \times S(n - 1, k) + S(n - 1, k - 1), n \leq 1 \tag{2.1}$$

As formas de particionar um conjunto com n elementos não vazios é definido por *Bell number*, como segue:

$$B_n = \sum_{k=1}^{n} S(n, k) \tag{2.2}$$

B_n é o número total de partições de uma arquitetura de software com n componentes. B_n cresce exponencialmente com n. Por exemplo, $B_1 = 1$, $B_3 = 5$, $B_5 = 52$, $B_7 = 877$. Portanto, o espaço de busca por um *design* arquitetural cresce conforme n (número de componentes) aumenta (Mueller, 2014).

Mueller (2014) não avalia sua proposta, nem define métricas para avaliar a arquitetura. O objetivo do autor é formalizar o problema de otimização de arquitetura, considerando múltiplas funções objetivo, como um problema de otimização utilizando sistema de formigas.

Tawosi *et al.* (2015) utilizam o ACO para propor um design otimizado de diagrama de classes de um software de acordo com algumas métricas. A representação da arquitetura é um grafo direcionado acíclico, onde os nós do grafo são organizados em duas dimensões. Colunas e linhas representam a responsabilidade e as classes. Dessa maneira a seleção de um determinado nó representa a possível atribuição de uma responsabilidade a uma classe. Em um primeiro estágio o algoritmo preprocessa o material de entrada (arquitetura) para as próximas etapas. Em seguida inicia a busca, gerando designs candidatos para encontrar a solução mais próxima de ótima possível. As formigas procuram caminhos no grafo acíclico para gerar suas soluções e depositam feromônio nestes caminhos percorridos de acordo com o valor da função objetivo. Uma vez que as formigas têm suas soluções construídas, a qualidade da solução é calculada pela função objetivo. Esta função avalia as soluções de acordo com as métricas de design de software utilizadas pelos autores. Como os autores utilizam várias métricas para avaliar a arquitetura, as formigas possuem multi-objetivos (multi-critérios) para determinar a qualidade de uma solução. Para isso utilizam múltiplas matrizes de feromônio, uma por objetivo. O algoritmo utiliza o valor da função objetivo para determinar a quantidade de feromônio que será depositado na matriz pelas formigas. As formigas continuam este processo de construir uma solução através da combinação dos

elementos arquiteturais até que seja atingido o valor máximo de iterações determinado pelos autores como critério de parada.

Simons e Smith (2013) comparam três métodos para otimização de design de software. Os autores comparam os seguintes algoritmos: otimização por colônia de formigas, algoritmo genético e busca gulosa. O modelo de representação de um design arquitetural (considerando classes, métodos e atributos) é um conjunto com uma sequência de inteiros, onde cada valor do conjunto representa um elemento do design do software, portanto $g_i = j$, significa que a solução g possui o elemento i inserido na classe j. O ACO implementado pelos autores, inicializa a matriz de feromônio com 1.0 em todas as posições de matriz e nas iterações subsequentes este valor vai sendo incrementado ou decrementado. Cada formiga percorre o espaço de busca (combinando os elementos atributos, métodos e classes) para construir sua solução, escolhendo cada combinação através do feromônio depositado na posição a matriz que indica aquela combinação. Após uma formiga finalizar a construção de sua solução, esta é avaliada pela função objetivo e a matriz de feromônio é atualizada. O critério de parada do algortimo é o número de iterações.

2.3.2. Métricas

Métricas são utilizadas para avaliar a qualidade de uma arquitetura. Na Figura 2.3 pode-se observar a relação entre a quantidade de artigos encontrados no mapeamento sistemático e as métricas utilizadas pelas abordagens encontradas. Pode-se notar que as métricas de coesão, acoplamento, segurança e eficiência são as mais utilizadas pelos autores na literatura.

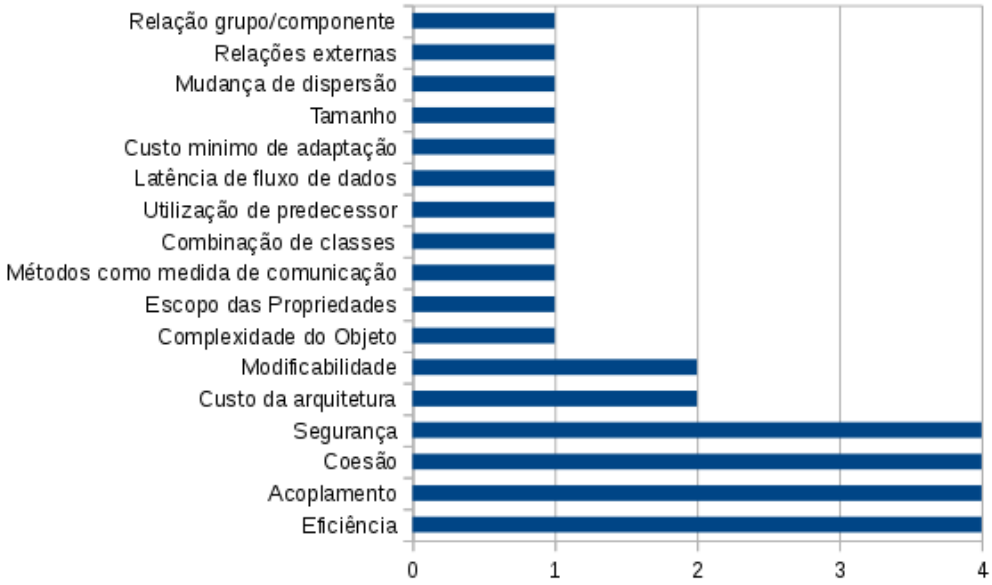

Figura 2.3. Métricas de qualidade da arquitetura de software identificadas no mapeamento sistemático

As métricas encontradas na literatura para medir a qualidade de arquiteturas de software baseadas em serviços foram: desempenho, segurança, custo da arquitetura, utilização de processador e latência do fluxo de dados. Martens *et al.* (2009, 2010) e Mirandola *et al.* (2014) utilizam métricas de desempenho, segurança e custo da arquitetura. Para medir o desempenho os autores utilizam técnicas para verificação do tempo de resposta do sistema: LQN (*Layered Queueing Networks*)(Koziolek; Reussner, 2008) e SimuCom (Becker *et al.*, 2009). Li *et al.* (2011) verificam a qualidade de uma arquitetura através da utilização de processador, latência do fluxo de dados e o custo da arquitetura. A plataforma proposta pelos autores permite que outros critérios de qualidade possam ser utilizados caso o arquiteto de software deseje.

Vathsavayi *et al.* (2013) propõem um algoritmo genético que avalia modificabilidade, eficiência e compreensividade da arquitetura. Para isso, utiliza as métricas definidas por Chidamber e Kemerer (1994), acoplamento, coesão, complexidade do objeto (*i.e.*, a cardinalidade de seu conjunto de propriedades), escopo de propriedades (uma classe é um conjunto de objetos que têm propriedade em comum, *i.e.*, métodos e variáveis), quantidade possível de troca de mensagens entre os objetos e por fim, combinação de classes (a combinação de duas classes resulta em outra classe onde as propriedades são a união das classes pais). Ramírez *et al.* (2013) e Hasheminejad e Jalili (2014) utilizam coesão e acoplamento para medir a qualidade de uma arquitetura baseada em componentes.

Barros *et al.* (2015) consideram três métricas para avaliar a metaheurística proposta: tamanho (número de pacotes, classes, atributos, métodos, método públicos e dependências), coesão e acoplamento. Ramírez *et al.* (2015) utilizam coesão e acoplamento como métrica e aplicam uma penalidade por relações externas (i.e., aplica-se uma penalidade caso haja alguma relação que não seja especificada por meio de uma interface) e relação grupo/componente (para a qual verifica-se a relação entre o número de operações, classes ou módulos e o número de componentes de toda a arquitetura do software). Chhabra *et al.* (2015) e Hussain *et al.* (2015) utilizam a métrica Modularização de Qualidade (MQ) para medir a qualidade da arquitetura. A MQ mede o equilíbrio entre coesão e acoplamento de um componente de software.

Tawosi *et al.* (2015) utilizam métricas de design de sistemas orientados a objetos para avaliar uma solução. Estas métricas são: acoplamento método-atributo, acoplamento método-método, coesão de uma classe e coesão comportamental de uma classe, métricas estas propostas por Chidamber e Kemerer (1994). Os autores também avaliam a complexidade do design e complexidade de uma classe, conforme as métricas propostas por Harabagiu *et al.* (1999). Os autores propoem ainda uma métrica chamada *meaningfulness metric* (MM), utilizada para medir quanto o diagrama de classes é compreensível para um humano.

As métricas consideradas por Simons e Smith (2013) para verificar a qualidade de uma

solução são: acoplamento do objeto e elegância do modelo arquitetural, esta sendo medida através do número de atributos e métodos de cada classe e da quantidade de atributos em cada método de uma classe do modelo.

2.3.3. Avaliação das soluções propostas

Martens *et al.* (2009, 2010) validam o algoritmo genético multiobjetivo utilizado aplicando-o em um sistema de relatório de negócios. A metaheuristica encontrou a solução após 4 iterações, retornando seis possíveis arquiteturas para esse sistema. Assim, o arquiteto de software deve analisar entre seis arquiteturas propostas a fim de obter a mais adequada. Porém, concluiu-se que o algoritmo é capaz de encontrar soluções de melhor qualidade, em contrapartida havendo uma redução do seu desempenho.

Ramírez *et al.* (2015) validam seu algoritmo testando-o em seis sistemas, onde cada um destes possui uma variedade de complexidade quanto a quantidade de classes e interfaces. Hussain *et al.* (2015) utilizam três sistemas no experimento a fim de validar sua proposta: um sistema de visualização de análise estatística; um conversor para linguagem de impressão e um sistema de despacho econômico. Para cada um dos sistemas, aplica-se o algoritmo de otimização por enxame de partículas e um algoritmo genético a fim de comparar qual possui o melhor desempenho. As comparações são baseadas no valor da função objetivo da melhor solução, na taxa de convergência e no tempo computacional.

Vathsavayi *et al.* (2013) utilizam um sistema chamado *E-home* a fim de validarem a proposta estabelecida. A arquitetura de software proposta pelo algoritmo foi comparada com a arquitetura feita manualmente por alunos, que levaram cerca de 40 minutos para construí-la, enquanto o algoritmo levou 5 segundos para encontrar a solução.

A ferramenta AQOSA, proposta por Li *et al.* (2011), é utilizada sobre dois *benchmarks*, um sistema para navegação de rádio de carro e um sistema de relato de negócios, a fim de avaliar seu desempenho.

Chhabra *et al.* (2015) conduziram quatro estudos empíricos para avaliar o desempenho do método proposto, onde cada estudo empírico era composto por sete problemas de instâncias reais e uma instância de problema aleatório e utilizou duas abordagens multiobjetivas para avaliação. Para os dois primeiros experimentos, a média e o desvio padrão da função objetivo MQ foram calculadas para as duas abordagens (MCA e ECA), sem a preservação dos componentes de núcleo. Para os outros dois experimentos considerou-se a preservação dos componentes de núcleo e também foram avaliadas a média e o desvio padrão da métrica MQ.

Para analisar a desempenho e a exatidão de sua proposta, Ramírez *et al.* (2013) testaram 30 execuções sobre três especificações de arquiteturas diferentes. Os autores compararam

o algoritmo proposto com uma busca aleatória. O algoritmo genético obtém um melhor desempenho em todas as instâncias do problema. Os autores medem em sua análise empírica o valor da função objetivo, a coesão, o acoplamento e o coeficiente de variação.

Mirandola *et al.* (2014) comparam o desempenho da Busca Tabu proposta em seu trabalho com os métodos de busca aleatória e uma avaliação numérica da escalabilidade da abordagem.

O algoritmo proposto por Barros *et al.* (2015) foi aplicado a última versão do software *Apache Ant* e percebeu-se que, enquanto os valores das métricas ao longo do processo de otimização iam aumentando, o *design* das versões otimizadas ia ficando mais complexas, isto é, concluiu-se que, apesar do algoritmo retornar uma boa arquitetura, essa era demasiadamente complexa, o que tornava-a difícil de ser adotada pelos desenvolvedores.

Tawosi *et al.* (2015) validam sua abordagem aplicando seu algoritmo em quatro projetos. Os autores buscam averiguar o poder do método ACO para encontrar soluções mais próximas do ideal possível. Dado os valores das métricas abordadas para o processo de otimização os autores verificaram que o método proposto é capaz de encontrar valores próximos do ideal para cada métrica considerada. Os autores também comparam a qualidade da arquitetura obtida pelo algoritmo com o design de uma arquitetura desenvolvida por um especialista.

2.4. Considerações Finais

O objetivo deste capítulo foi identificar as metaheurísticas mais utilizadas na literatura, as métricas mais abordadas e a forma como estas propostas estão sendo avaliadas. Subsidiando a decisão de quais métricas serão utilizadas e como iremos avaliar nossa abordagem.

Baseada nas informações recuperadas na seções anteriores, foi possível observar que os autores normalmente consideram como entrada do algoritmo modelos arquiteturais desenvolvidos por linguagens de modelagem apropriada (SCA-ASM *assemblies* Mirandola *et al.* (2014), Palladio Component Model - PCM Martens *et al.* (2009, 2010)). Dada a forma de entrada da arquitetura, esta precisa ser representada para o algoritmo. Diversas formas de representação da arquitetura são utilizadas na literatura, as mais comuns são: representação dos relacionamentos através de vetores ((Barros *et al.*, 2015), (Martens *et al.*, 2009, 2010)), árvores (Ramírez *et al.*, 2015), grafos (Chhabra *et al.*, 2015) e matrizes (Hussain *et al.*, 2015). Verificamos que o algoritmo genético é a metaheurística mais usada para otimização arquitetural e que as métricas de qualidade mais comuns são coesão e acoplamento. O estilo arquitetural mais abordado é o estilo baseado em componentes. As formas de avaliação encontradas mais utilizadas foram busca aleatória, comparação da proposta com outro algoritmo e comparação da arquitetura retornada pela proposta com a arquitetura desenvolvida por um arquiteto.

Capítulo 3

ACO2SoftwareArchitecture

Este capítulo descreve todas as etapas para o desenvolvimento do algoritmo de otimização baseado em colônia de formigas feito neste trabalho, *ACO2SoftwareAchitecture*. A Seção 3.1 descreve o tipo de arquitetura considerada neste trabalho. A Seção 3.3 descreve como o modelo arquitetural baseado em componentes é representado, como ele é descrito e a maneira como a ferramenta processa este modelo. Na Seção 3.4 é descrito como a arquitetura é representada para o algoritmo. A Seção 3.5 discorre sobre a métrica de qualidade arquitetural utilizada para avaliar as soluções geradas. A Seção 3.6 descreve a forma que o algoritmo baseado em colônia de formigas foi utilizado no contexto de arquitetura de software.

3.1. Arquitetura Baseada em Componentes

A arquitetura utilizada como entrada do algoritmo desenvolvido neste trabalho é uma arquitetura baseada em componentes. Esta arquitetura deve ser representada por um modelo UML, escrito em um arquivo XMI, onde devem estar presentes elementos da arquitetura como: relacionamentos, classes e componentes.

3.2. Recuperação da arquitetura

Caso o projeto considerado não possua um arquivo XMI (*XML Metadata Interchange* é um padrão defino pela *Object Management Group* (OMG) para troca de informações de matadados via Extensible Markup Language (XML)) representando sua arquitetura, o modelo UML pode ser extraído a partir do código fonte. Para esta situação, o modelo UML é extraído

através de um plugin da plataforma Eclipse chamado Modisco [1]. Dado um código fonte de entrada este plugin analisa o código extraindo informações para a criação de um modelo de conhecimento. Normalmente, tal extração considera todos os dados do código fonte: classes, atributos, métodos e corpo dos métodos. Tal modo de operação é chamado de *deep analysis*, porém é possível omitir a análise do corpo dos métodos. Isso permite que o desempenho da análise seja melhorado em termos de tempo e memória. Considerando que a representação adotada não considera informações do corpo dos métodos, neste trabalho não consideramos *deep analysis*, pois como o projeto utilizado para os experimentos é relativamente grande, quando tentávamos fazer uma análise profunda do projeto o plugin dava estouro de memória. Para as arquiteturas recuperadas a partir de código fonte e que não possuem componentes definidos explicitamente, adotou-se a heuristica de considerar os pacotes do projeto como componentes.

3.3. Representação do Modelo

O algoritmo parte do modelo UML para extrair as informações da arquitetura. Para fazer esta extração programaticamente, utilizamos a API UML2 que permite o processamento de um modelo UML como um objeto *model*. Dessa forma, a partir deste modelo há como identificar os elementos do modelo UML e extrair informações como: quantidade de classes, pacotes (consideradas como os componentes), relacionamentos entre classes e pacotes e os relacionamentos entre as classes da arquitetura. Estas informações são então inseridas em estruturas de dados, que armazenam os relacionamentos do modelo, estas estruturas são utilizadas pelo algoritmo para armazenar a disposição dos relacionados dos elementos do modelo. Foi implementado um módulo no algoritmo chamado *LoadModel* que é responsável por fazer a extração da arquitetura de entrada para as estruturas de dados (matrizes e mapas) utilizadas pelo algoritmo que representam a arquitetura.

É possível dar dois tipos de arquitetura de entrada para o algoritmo, uma é os componentes e classes soltos, sem nenhum relacionamento entre si, o algoritmo tentará combinar os elementos da melhor maneira possível baseado na métrica. A segunda opção, é dar de entrada uma arquitetura já estruturada e o algoritmo otimizar esta arquitetura dada.

3.3.1. Representação do Estilo

O algoritmo permite a entrada de projetos com o estilo arquitetural definido no modelo UML. Este estilo arquitetural irá auxiliar o ACO na busca pela solução mais adequada. Para que

[1] https://eclipse.org/MoDisco/

isso fosse possível utilizamos técnicas UML que permitem a anotação do estilo arquitetural no modelo UML. Estas técnicas são Perfil (*Profiles*) e Estereótipos (*Stereotypes*) UML.

Perfil UML provê uma forma genérica para customizar ou personalizar modelos UML para domínios e plataformas específicas (Rumbaugh *et al.*, 2004). Os perfis UML são definidos usando estereótipos, *tagged values* ou restrições que são aplicadas em determinados elementos do modelo.

Este mecanismo UML foi utilizado para poder representar os estilos arquiteturais de um modelo, os estilo arquiteturais representados foram: em camadas e cliente/servidor. Foi criado um Perfil denotado por *ArchitectureStyle*. Este perfil representa o estilo arquitetural. Ele contém três estereótipos: *Layered*, *Server* e *Client*. Os estereótipos representam os tipos de estilos arquiteturais considerados (Figura 3.1, Figura 3.2). O estereótipo *Layered* possui uma variável *id*, que contém o nome da camada que o componente pertence (Figura 3.1). O elemento marcado por um estereótipo no estilo arquitetural em camadas foi o pacote, que representa o componente, como mostra a Figura 3.3. Para o segundo estilo arquitetural considerado, cliente/servidor, foi criado dois estereótipos: *Client* e *Server*, para estes estereótipos não foi definido variáveis. Para o estilo cliente/servidor, a marcação do estereótipo foi feita na classe.

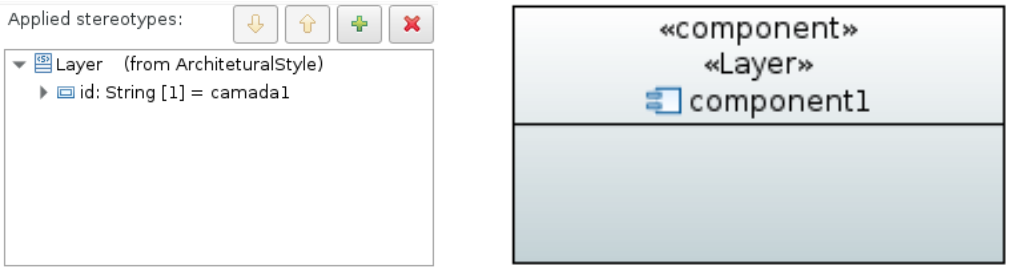

Figura 3.1. Exemplo de arquitetura em camadas representada com Perfil UML.

Figura 3.2. Exemplo de arquitetura cliente/servidor representada com Perfil UML.

A Figura 3.3 apresenta um exemplo da utilização de Perfil e Estereótipo em um modelo arquitetural baseado em componentes. O estilo representado é o em camadas e cada componente que possui a denotação *«Layer»* é uma camada do sistema. A Figura 3.1 mostra a atribuição de valor à variável que este estereótipo possui o nome da camada que o

componente está inserido. O algoritmo ACO utiliza a informação do estilo arquitetural para verificar se as soluções geradas quebram o estilo imposto no modelo arquitetural. Portanto, os elementos denotados no modelo serão armazenados em estruturas de dados para que o algoritmo possa fazer esta averiguação e levar em consideração a informação do estilo arquitetural para construir uma solução.

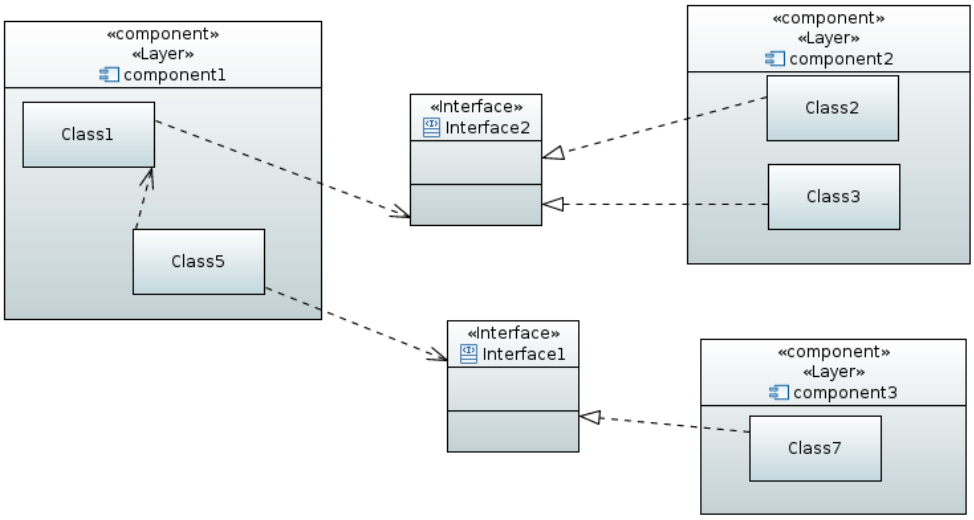

Figura 3.3. Modelo representativo de um modelo arquitetural com a utilização de estilo em camadas

3.4. Representação do problema

O ACO desenvolvido neste trabalho possui duas matrizes de feromônio: uma representando as classes e componentes (Tabela 3.1) e outra representando os relacionamentos entre as classes (Tabela 3.2). Essas matrizes informam a quantidade de feromônio depositada em determinada posição, de forma que durante o processo de construção da solução, a formiga seja influenciada pelas combinações mais atrativas (que possuem maior valor de feromônio). Sendo assim, os valores armazenados nestas matrizes estão diretamente relacionados com os valores das métricas de qualidade. Inicialmente todas as posições da matriz de feromônio possuem a mesma atratividade de 0.5 (Tabela 3.1, Tabela 3.2). Conforme as formigas vão construindo suas soluções e as melhores vão sendo selecionadas, esta matriz vai sendo atualizada. Na Tabela 3.2 há uma coluna N, a qual representa a possibilidade de não combinar a classe C_x outras classes, ou seja, a classe C_x não tem relacionamentos.

Tabela 3.1. Matriz Classe por Componente

	Comp1	Comp2	Comp3	Comp4
C1	0.5	0.5	0.5	0.5
C2	0.5	0.5	0.5	0.5
C3	0.5	0.5	0.5	0.5

Tabela 3.2. Matriz Classe por Classe

	C1	C2	C3	N
C1	0.5	0.5	0.5	0.5
C2	0.5	0.5	0.5	0.5
C3	0.5	0.5	0.5	0.5

3.5. Métricas e função objetivo

Dada uma solução proposta pela metaheuristica, esta precisa ser avaliada. Para tal avaliação é utilizada métricas de qualidade que verificam se uma solução é boa dada uma medida de avaliação. As métricas podem ser representadas em termos de equações para que se possa utilizá-las como função objetivo da metaheuristica. Esta seção apresenta a métrica utilizada para medir a qualidade de uma arquitetura e a define em termos da função objetivo utilizada pelo ACO.

A qualidade da arquitetura pode ser avaliada em termos de coesão e acoplamento. Normalmente é desejável ter uma alta coesão e um baixo acoplamento para uma arquitetura. Estas características asseguram que a arquitetura será compreensível, manutenível e reutilizável (Saed; Kadir, 2011). Coesão mede quanto as classes de um componente estão intra-relacionadas (intra-conectividade) e o acoplamento mede quão inter-dependentes (inter-conectividade) são dois componentes (Saed; Kadir, 2011). Inter-conectividade representa dependência entre componentes, ou seja, relacionamentos entre classes que pertencem a componentes diferentes. A intra-conectividade representa relacionamentos entre classes que estão em um mesmo componente.

A inter-conectividade $E_{i,j}$ entre um componente i e j com N_i e N_j classes (quantidade de classes presentes nos componentes i e j) e $\epsilon_{i,j}$ inter-arestas (arestas que começam no componente i e terminam no componente j) (Saed; Kadir, 2011; Mancoridis *et al.*, 1999)é calculada pela Equação 3.1:

$$E_{i,j} = \begin{cases} 0 & \text{i = j} \\ \frac{\epsilon_{i,j}}{2 \cdot N_i \cdot N_j} & \text{se } i \neq j \end{cases} \tag{3.1}$$

A intra-conectividade (coesão) A_i de um componente i com N_i classes e μ_i intra-arestas (arestas e começam e terminam no mesmo componente) (Saed; Kadir, 2011; Mancoridis *et al.*, 1999) é dada pela Equação 3.2:

$$A_i = \frac{\mu_i}{{N_i}^2} \tag{3.2}$$

Ambas as métricas, coesão e acoplamento (intra e inter-conectividade), são encapsuladas em uma métrica chamada *Modularization Quality* (MQ) que determina a qualidade de um grafo de dependência de módulos (*Module Dependency Graph MDG*). Um MDG é formalmente representado por $MDG = (M, R)$, onde M é o conjunto de componentes do software e R é um conjuntos de pares de classes $< u, v >$ que representam os relacionamentos entre as classes (Mancoridis *et al.*, 1999). A métrica MQ será utilizada como função objetivo do algoritmo ACO. A Equação 3.3 representa a métrica MQ, considerando inter e intra conectividade. A variável k representa a quantidade de componentes do sistema.

$$MQ = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k \cdot (k-1)}{2}} & \text{se k} > 1 \\ A_1 & \text{se k} = 1 \end{cases} \tag{3.3}$$

Os resultados da Equação 3.3 variam entre -1 e 1, sendo que -1 significa um componente sem coesão interna e 1 significa um componente sem acoplamento externo (Mancoridis *et al.*, 1999). Portanto a metaheuristica irá buscar pelo maior valor de MQ possível.

3.6. ACO para o problema de otimização de arquitetura de software

O algoritmo foi implementado na linguagem de programação Java. O processo completo para otimização da arquitetura é mostrada na Figura 3.4. A partir do código fonte de um projeto Java o algoritmo faz a extração do modelo arquitetural. Após esta etapa inicia-se a análise dos elementos do modelo UML: classes, componentes e relacionamentos. Finalizada esta etapa o ACO inicia a otimização do modelo arquitetural. Após o fim de todas as iterações o algoritmo retorna um arquivo texto que representa a solução com o melhor valor da função objetivo obtida.

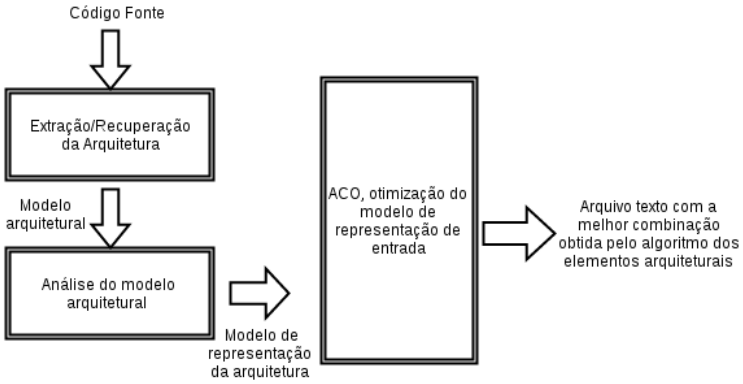

Figura 3.4. Processo de otimização arquitetural.

Durante o processo de construção de uma solução, o algoritmo ACO faz a combinação de uma classe i dentro de um componente j e calcula a probabilidade desta combinação. Esta probabilidade é dada pela Fórmula 3.4, que representa a probabilidade da formiga k inserir a classe i no componente j na iteração t.

$$P_{i,j}^{k}(t) = \frac{\tau_{i,j}^{\alpha}(t) \cdot \eta_{i,j}^{\beta}(t)}{\sum_{m=0}^{|\{Comp\}|} \tau_{i,m}^{\alpha}(t) \cdot \eta_{i,m}^{\beta}(t)} \tag{3.4}$$

- $\tau_{i,j}$ é o feromônio depositado na posição i, j da matriz de feromônio;
- $\eta_{i,j}$ informação heurística. Usada neste trabalho como penalizações, as arquiteturas que infringem seu estilo arquitetural serão desvalorizadas e tendem a não ser escolhidas, o cálculo desta penalidade é definida na Seção 3.6.1;
- α e β são parâmetros que salientam a importância da informação definida pelo feromônio (τ) e da informação heurística definida (η).
- $Comp$ é o conjunto de componentes da arquitetura, representado no algoritmo por uma matriz de componentes.

Outra combinação feita é de classe com classe, que calcula a probabilidade de haver um relacionamento entre duas classes, representada na Fórmula 3.5.

$$P_{i,j}^{k}(t) = \frac{\tau_{i,j}^{\alpha}(t) \cdot \eta_{i,j}^{\beta}(t)}{\sum_{m=0}^{|\{Classes\}|} \tau_{i,m}^{\alpha}(t) \cdot \eta_{i,m}^{\beta}(t)} \tag{3.5}$$

- $\tau_{i,j}$ é o feromônio depositado na posição i, j da matriz de feromônio;
- $\eta_{i,j}$ informação heurística, é abordada neste trabalho como penalizações, as arquiteturas que infringem seu estilo arquitetural serão desvalorizadas e tendem a não ser escolhidas;
- α e β são parâmetros que salientam a importância da informação definida pelo feromônio (τ) e da informação heurística definida (η).
- $Classes$ é o conjunto de classes da arquitetura, representado no algoritmo por uma matriz de classes.

A Figura 3.5 mostra o fluxograma de funcionamento do algoritmo ACO. O procedimento de construção de uma solução é descrito pelo Algoritmo 1. Para construir a solução, cada formiga utiliza as Funções 3.4 e 3.5 para decidir adicionar uma classe em um componente e se irá haver relacionamento entre duas classes.

O procedimento de construção da solução dá-se em duas partes. Na primeira etapa a formiga k percorre cada linha da matriz de feromônio de classe $\times$ componente (Tabela 3.1), combinando a classe i no componente j e calculando a probabilidade desta escolha (Equação 3.4). Cada probabilidade calculada é inserida em um vetor de probabilidades. Ao final

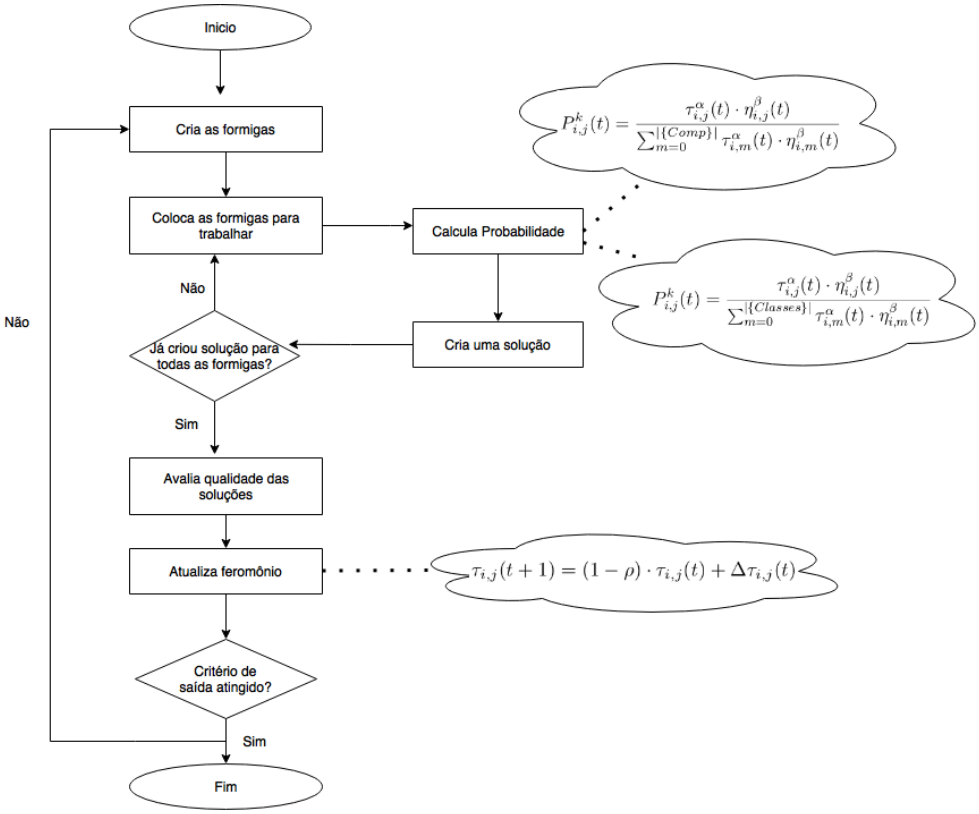

Figura 3.5. Fluxograma do funcionamento do ACO.

da linha é utilizada uma roleta a fim de escolher uma das probabilidades. A escolhida irá representar a combinação de classe dentro de componente. Esta solução parcial é armazenada e a formiga continua construindo sua solução.

Algoritmo 1 Contrói uma solução

```
1:  function CONTROISOLUÇÃO
2:      for i ← 1 até qtd_classes do
3:          for j ← 1 até qtd_componentes do
4:              calculaProbabilidade();
5:          escolheProbabilidadeAleatoriamente();
6:      if há uma arquitetura inicial then
7:          calculaPenalidade();
8:      for i ← 1 até qtd_classes do
9:          for j ← 1 até qtd_classes do
10:             calculaProbabilidade();
11:         escolheProbabilidadeAleatoriamente();
12: end function
```

Na sequência vem a função calculaPenalidade(). Esta função irá acontecer se houver

uma arquitetura inicial (arquitetura estruturada dada como entrada ao algoritmo) e tem por objetivo verificar qual o estilo arquitetural da arquitetura de entrada e analisar quais combinações de classes quebram a regra do estilo. Esta informação servirá para a informação heurística no momento do cálculo da probabilidade de combinação entre as classes.

A terceira etapa irá combinar pares de classe buscando criar relacionamento entre elas. Para isso o algoritmo percorre a matriz classe × classe (Tabela 3.2), calculando a probabilidade de combinar a classe i e a classe j (Equação 3.5). Ao final de cada linha uma roleta é executada a fim de escolher uma das combinações. A combinação escolhida é armazenada e o algoritmo continua o processo até o fim da matriz. Ao final destes procedimentos a formiga k terá sua solução construída.

O próximo passo é avaliar a qualidade dessa solução por uma função objetivo baseada nas métricas de qualidade. Após todas as formigas construírem suas soluções, é verificado qual possui o melhor valor da função objetivo. Escolhida a melhor solução, são atualizados os valores de feromônio da matriz de classe por componente e na matriz de classes (apenas as classes e componentes presentes na solução são atualizadas nas matrizes de feromônio). A atualização do feromônio é feita baseada na Função 3.6.

$$\tau_{i,j}(t+1) = (1-\rho) \cdot \tau_{i,j}(t) + \Delta\tau_{i,j}(t), \tag{3.6}$$

em que $\Delta\tau_{i,j}(t)$ representa o incremento de feromônio no tempo $t+1$ na combinação (i,j). $\Delta\tau_{i,j}(t)$ é diretamente proporcional a função objetivo, pois a quantidade de depósito de feromônio está atrelada ao valor da função objetivo. A constante ρ representa a taxa de evaporação de ferômonio. Este valor tem como objetivo tirar uma quantia de feromônio das matrizes a fim de que, soluções com valores ruims da funções objetivo acabem sendo desvalorizadas e não sejam mais selecionadas. Este fator é importante pois ele acaba apagando, conforme a evolução das iterações, da matriz os caminhos que não produzem bons resultados.

3.6.1. Informação Heurística

A informação heurística do algoritmo é abordada neste trabalho como penalizações. As penalizações são dadas verificando o estilo arquitetural da arquitetura de entrada. Se a arquitetura estruturada dada de entrada define um estilo arquitetural, a otimização será feita levando em consideração este estilo. Caso não haja, então a informação heurística será desconsiderada.

Os estilos arquiteturais considerados para este trabalho são: estilo arquitetural em camada e estilo arquitetural cliente/servidor. Segundo Garlan e Shaw (1993), o estilo arquitetural

em camadas é organizado hierarquicamente. Cada elemento de uma camada provê serviços para a camada acima e serve como cliente para a camada abaixo. As camadas são limitadas por regras que delimitam a comunicação entre camadas: elementos em cada camada podem acessar somente elementos em sua própria camada, ou elementos na camada diretamente abaixo. Mariani (2015) propõe em seu trabalho algumas regras que os estilos arquiteturais devem seguir. O estilo em camada possui a seguinte regra: um elemento de uma camada x só pode se relacionar com um elemento de uma camada $x + 1$ ou um elemento da camada x. A Figura 3.6 mostra um exemplo de relacionamento que quebra a regra do estilo arquitetural em camadas.

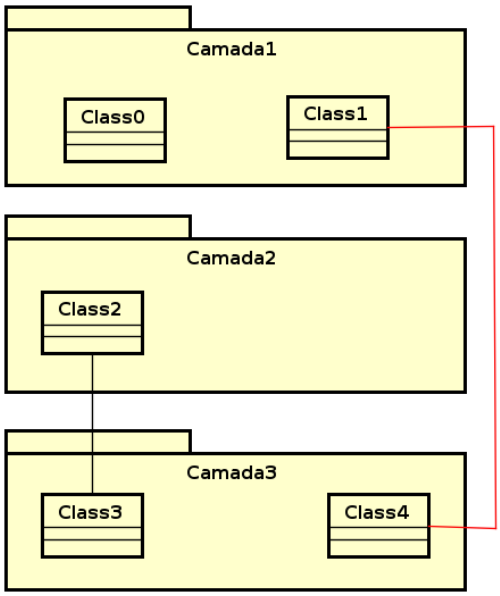

Figura 3.6. Exemplo do estilo arquitetural em camadas.

No estilo cliente/servidor, um servidor representa um elemento que fornece serviços a outros elementos da arquitetura e um cliente é um elemento que consome serviços oferecidos por um servidor. Mariani (2015) propõe as seguintes regras para o estilo cliente/servidor:

- um elemento arquitetural a presente em um cliente poderá utilizar um elemento arquitetural b somente quando este estiver presente no mesmo cliente que a ou em qualquer servidor da arquitetura;
- um elemento arquitetural a presente em um servidor poderá utilizar um elemento arquitetural b quando este estiver presente em qualquer servidor da arquitetura.

Dessa maneira, um relacionamento bidirecional pode existir somente entre elementos de um mesmo cliente, de um mesmo servidor ou entre servidores. Um relacionamento unidirecional pode existir nos mesmos casos que um relacionamento bidirecional e também entre um cliente e um servidor desde que o elemento que fornece serviços pertença ao servidor e o elemento que os usa pertença ao cliente. A Figura 3.7 ilustra uma arquitetura com estilo

cliente/servidor e os relacionamentos que quebram a regra.

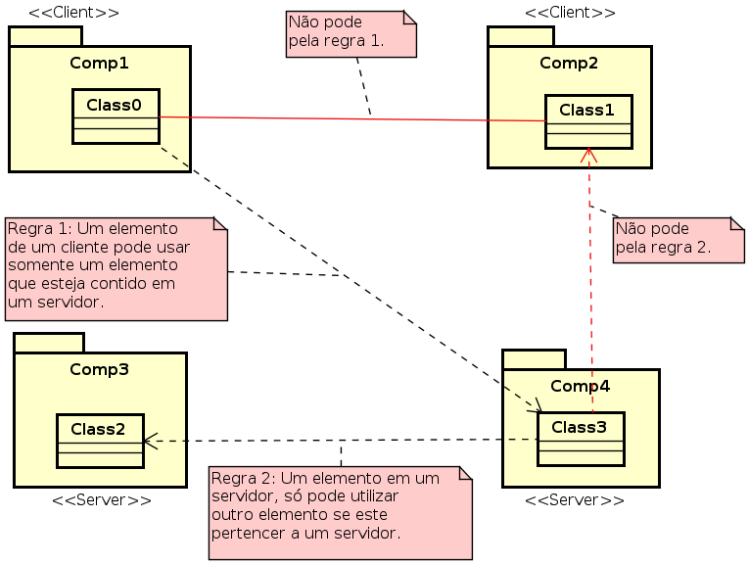

Figura 3.7. Exemplo do estilo arquitetural cliente/servidor.

O Algoritmo 2 representa como é feito o cálculo da penalidade. Percorre-se a matriz de relacionamentos classe×classe e qual estilo arquitetural esta imposto na arquitetura. Caso o estilo arquitetura seja em camadas (*LAYER*) verifica-se o relacionamento entre a classe i e j pertencentes as camadas $layer_i$ e $layer_j$, quebra as regras do estilo em questão. Caso o estilo definido na arquitetura seja cliente/servidor, verifica se o relacionamento entre as classe i e j que são dos tipos $tipo_i$ e $tipo_j$ (uma classe poder ser cliente ou servidor) quebra as regras.

Algoritmo 2 Calcula Penalidade

```
1: function CALCULAPENALIDADE
2:     for i ← 1 até qtd_classes do
3:         for j ← 1 até qtd_classes do
4:             if estilo == LAYER then
5:                 verificaPenalidadeEstiloCamada(i,j,layer_i, layer_j)
6:             else if estilo == CLIENT/SERVER then
7:                 verificaPenalidadeEstiloClienteServidor(i,j, tipo_i, tipo_j)
8: end function
```

Se o relacionamento agredir a regra, este é adicionado a uma lista de vetores, que armazena os relacionamentos que quebraram a regra, e em um mapa que armazena a classe e a quantidade de vezes que esta quebrou a regra. Dessa forma, teremos todos os relacionamentos que quebraram a regra e quantas vezes cada classe quebrou a regra.

Este processo é feito para que, no momento da combinação das classes para gerar os relacionamentos (*for* da linha 8 do Algoritmo 1), possamos informar para a função de cálculo

da probabilidade o total de vezes que cada classe envolvida no relacionamento quebra a regra.Este total é dividido pelo total de quebras de regra que a arquitetura possui. O valor passado para a função de probabilidade é o seguinte:

$$penalidade_{i,j} = 1 - \frac{total_i + total_j}{totalGeralDeQuebrasDaRegra} \tag{3.7}$$

A Equação 3.7, que caracteriza quebras de regras do estilo arquitetural para um relacionamento entre as classes i, j, é a informação heuristica ($\eta_{i,j}$) utilizada para auxiliar na busca pela melhor solução.

3.7. Considerações Finais

Neste capítulo apresentamos como desenvolvemos o algoritmo ACO para otimização arquitetural. Explicamos como se deu a representação da arquitetura, passando pela recuperação caso não haja um arquivo XMI representando o modelo arquitetural. Mostramos como se deu a representação do estilo arquitetural no modelo UML para que possamos utilizar esta informação durante a busca pela arquitetura de melhor valor MQ. Foi mostrado ainda, a métrica utilizada para avaliar a qualidade de uma solução e a definição desta em termos da função objetivo para ser aplicada no ACO. Explicamos também como formulamos a informação do estilo arquitetural como métrica para utilizar como informação heuristica no ACO. A próxima seção irá mostrar os experimentos feitos e os resultados obtidos pela abordagem proposta nesta seção.

Capítulo 4

Experimento, Resultados e Discussão

Este capítulo descreve os passos feitos para obter os resultados da abordagem proposta neste trabalho. Na Seção 4.1 mostramos como foram executados os experimentos do algoritmo, os projetos usados e as configurações utilizadas para calibrar o ACO. Na Seção 4.2 mostramos os resultados obtidos pelo algoritmo e, por fim, na Seção 4.3 foram analisados e discutidos os resultados obtidos pela abordagem.

4.1. Experimento

Os experimentos feitos neste trabalho foram executados sobre as versões 1.1.0 e 1.3.0 do software *Apache Ant*, um projeto *open-source* usado para construção e implantação de projetos Java (Apache Software Foundation, 2000). A Tabela 4.1 mostra a quantidade de componentes, classes e o valor de MQ para cada uma das versões analisadas. Como o software *Apache Ant* não possui componentes definidos em sua estrutura, consideramos os pacotes como componentes do sistema, seguindo a heuristica adotada em outros trabalhos.

Tabela 4.1. Apache-Ant

Projeto	Nº de classes	Nº de componentes	MQ
Apache Ant 1.1.0	97	7	0.55534
Apache Ant 1.3.0	274	26	0.51541

Para cada versão do software fizemos uma versão com o estilo arquitetural em camadas, aplicando-se os Perfis e Estereótipos explicados na Seção 3.3.1. Para definir uma nova camada do sistema consideramos a seguinte heuristica: cada pacote da raiz principal do projeto é uma camada, os subpacotes pertencem a mesma camada que seu pacote pai, desde que dentro deste subpacote tenha apenas classes, caso um subpacote possua mais subpacotes dentro

então ele será uma nova camada. Este processo foi feito para que possamos avaliar a métrica de estilo arquitetural proposta neste trabalho. A Tabela 4.2 mostra a quantidade de camadas definida para cada versão.

Tabela 4.2. Apache-Ant Com Estilo Arquitetural em Camadas

Projeto	Nº de classes	Nº de componentes	Nº de Camadas	MQ
Apache Ant 1.1.0	97	7	6	0.55534
Apache Ant 1.3.0	274	26	20	0.51541

Para cada teste feito, aplicamos várias combinações dos parâmetros de configuração para verificar qual obtinha melhor resultado do valor MQ. Além disso, cada configuração foi executada 10 vezes. Essas 10 execuções foram feitas por causa da caraterística probabilística do algoritmo. Os parâmetros que obtiveram melhores valores foram armazenados para testes posteriores e as configurações que não produziram valores satisfatórios foram descartadas. Lembrando que valores satisfatórios de MQ são os valores mais próximos de 1, portanto o algoritmo busca maximizar o valor MQ em busca da solução mais próxima da ideal possível.

Tabela 4.3. Parâmetros de configuração do ACO para Apache-Ant 1.1.0.

	ACO
Formigas	{5, 10, **20, 30, 50**}
Iterações	{10, **20, 50, 100**}
ρ	{0.1, **0.2**, 0.3, 0.4, 0.5, 0.6, **0.7, 0.8**, 0.9, 1.0}
α	{0.1, **0.2**, 0.3, **0.4, 0.5, 0.6, 0.7, 0.8, 0.9**, 1.0}
β	{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}

As Tabelas 4.3 e 4.4 mostram as configurações executadas para as versão 1.1.0 (Tabela 4.3) e 1.3.0 (Tabela 4.4) do software Apache Ant. Os valores destacados em negrito foram os que obtiveram melhores resultados nos experimentos feitos. Alguns destes valores foram selecionados e são reportados na Seção 4.2.

4.2. Resultados

Os resultados obtidos pelo algoritmo foram comparados com o modelo original da arquitetura, ou seja, dada uma arquitetura de entrada, extraímos dela o valor da métrica de qualidade, MQ (coesão e acoplamento). Depois da finalização do algoritmo, comparamos os resultados de MQ da solução indicada pelo ACO com o valor MQ da solução original. Também analisamos a forma de distribuição das classes dentro dos componentes e dos relacionamentos entre as classes da solução original (arquitetural de entrada) e da solução proposta pelo ACO, com o objetivo de verificar qual a diferença da distribuição para MQ's com valores altos e baixos e analisar a diferença da distribuição entre as soluções propostas e a arquitetura inicial.

Tabela 4.4. Parâmetros de configuração do ACO para Apache-Ant 1.3.0.

	ACO
Formigas	{**50**, 100, 200, **500**}
Iterações	{**20**, 50, **100**, 200}
ρ	{0.6, **0.7**, 0.8, 0.9, 1.0}
α	{**0.4**, 0.5, 0.6, 0.7, 0.8, **0.9**}
β	{0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}

4.2.1. Apache 1.1.0

A partir da análise dos resultados obtidos pelo ACO, é possível averiguar que, partindo da métrica MQ, o algoritmo consegue encontrar soluções satisfatórias em relação à arquitetura de entrada, como mostra os valores da coluna MQ da Tabela 4.5, que mostra algumas configurações usadas e os resultados obtidos para a versão 1.1.0 do software Apache Ant, ordenados em ordem decrescente dos valores da coluna MQ.

Podemos verificar que mesmo a pior solução encontrada pelo algoritmo é melhor que a solução de entrada que possui valor MQ de 0.55534 enquanto o da pior solução é 0.5629. A coluna ρ é a taxa de evaporação de feromônio utilizada pelo ACO, α é a constante que ressalta a importância da métrica MQ e β é a constante que realça a importância da informação heuristica utilizada. A coluna *Id* representa um identificador para a configuração, para que esta possa ser referenciada mais adiante no texto. Podemos observar que considerando o tamanho da arquitetura e a quantidade de iterações 100, valores de ρ baixo conseguem obter bons resultados, isso acontece pois, como a arquitetura é relativamente pequena 100 iterações é um valor alto. Sendo assim mesmo com uma baixa taxa de evaporação, por ter muitas iterações ainda assim a configuração alcança bom resultado.

Tabela 4.5. Resultados obtidos pelo ACO considerando a versão 1.1.0 do software Apache Ant.

Id	Iterações	Formigas	ρ	α	β	MQ
1	100	15	0.2	0.4	0.2	0.61802
2	100	5	0.4	0.2	0.2	0.61765
3	100	15	0.4	0.2	0.4	0.61033
4	100	5	0.1	0.9	0.8	0.59660
5	50	20	0.7	0.4	0.9	0.59499
6	30	20	0.4	0.6	0.9	0.59274
7	20	20	0.6	0.4	0.1	0.59023

A Tabela 4.6 mostra como ficou a distribuição das classes dentro dos componentes, ou seja, quantidade de classes presentes dentro de cada componente. A coluna *Id do Componente* representa o valor de identificação do componente, a coluna Id: 1 representa a quantidade de classes dentro de cada componente obtido pela configuração: iteração = 100, formigas = 15, $\rho = 0.2$, $\alpha = 0.4$, $\beta = 0.2$, que obteve MQ = 0.61802 (ver Tabela 4.5). Procedeu-se de forma semelhante para a coluna Id : 7 que obteve MQ = 0.59023. A coluna *Distribuição*

Original ilustra a distribuição das classes dentro dos componentes da arquitetura original, a arquitetura dada de entrada ao algoritmo. Com base na Tabela 4.6 podemos verificar que a distribuição das duas configurações diferem de forma considerável da distribuição original. Observa-se que ambas as soluções distribuíram melhor o número de classes dentro de cada componente do que a arquitetura inicial. No entanto, podemos notar uma diferença entre a forma de distribuição das duas soluções. A configuração Id : 1 que obteve valor de MQ mais alto, gerou componentes com um maior número de classes do que a configuração Id : 2 com valor MQ menor.

Tabela 4.6. Distribuição da quantidade de classes dentro de um componente.

	Id: 1 Iterações: 100 Formigas: 15 ρ: 0.2, α: 0.4, β: 0.2	Id: 7 Iterações: 20 Formigas: 20 ρ: 0.6, α: 0.4, β: 0.1	Distribuição Original
Id do Componente	Quantidade de Classes	Quantidade de Classes	Quantidade de Classes
0	2	17	2
1	2	2	23
2	17	11	51
3	15	2	5
4	2	13	8
5	28	27	2
6	30	24	5

A Tabela 4.7 mostra a quantidade de classes e a quantidade de relacionamentos internos (relacionamentos entre classes de um mesmo componentes) e externos (relacionamentos entre classes de componentes diferentes). A coluna *Arquitetura original* mostra como são distribuídos os relacionamentos internos e externos da arquitetura de entrada. Dessa maneira a leitura da tabela fica da seguinte forma, a configuração Id: 1 (iteração: 100, formiga: 15, ρ: 0.2, α: 0.4, β: 0.2) possui 1 classe com 35 relacionamentos internos e 43 relacionamentos externos. A configuração Id: 2 (iteração: 20, formiga: 20, ρ: 0.6, α: 0.4, β: 0.1) possui 1 classe com 31 relacionamentos internos e 44 relacionamentos externos enquanto a arquitetura original possui 1 classe com 13 relacionamentos internos e 8 relacionamentos externos.

Podemos observar na Tabela 4.7 que ambas as configurações redistribuíram de forma similar a quantidade de relacionamentos internos e externos em comparação com a distribuição original. A primeira configuração, que obteve resultado MQ mais alto, tem menos relacionamentos externos do que a segunda configuração (primeira linha da tabela mostra que 31 relacionamentos externos para 0 quantidade de classes, enquanto a segunda configuração possui 15 relacionamentos externos para 0 quantidade de classes). Isso significa que, quanto menor o número de relacionamentos externos melhor tende a ser o valor da métrica MQ.

As tabelas a seguir reportam os resultados obtidos a partir de uma arquitetura de entrada

Tabela 4.7. Distribuição da quantidade de classes com relacionamentos internos e externo.

	Id: 1 Iterações: 100 Formigas: 15 ρ: 0.2, α: 0.4, β: 0.2		Id: 7 Iterações: 20 Formigas: 20 ρ: 0.6, α: 0.4, β: 0.1		Arquitetura original	
Quantidade de Classes	Rel. Internos	Rel. Externos	Rel. Internos	Rel. Externos	Rel. Internos	Rel. Externos
0	0	31	0	15	75	83
1	35	43	31	44	13	8
2	36	16	41	26	3	0
3	18	4	17	8	1	2
4	3	1	4	1	1	1
5	2	0	2	0	2	0
10	0	0	0	0	0	1

com estilo arquitetural em camadas da versão 1.1.0 do software Apache-Ant. A ordenação dos dados da Tabela 4.8 se dão de forma decrescente considerando a coluna MQ. Podemos analisar pela coluna de Relacionamentos Penalizados (Rel. Penalizados) que as soluções com maiores valores de MQ e iterações infringiram menos as regras do estilo arquitetural (as regras estão definidas na Seção 3.6.1) do que as soluções com valores baixos de MQ e iterações. É possível notar que com a taxa de evaporação (ρ) baixa os resultados obtidos de MQ não são satisfatórios e o número de relacionamentos que violam a regra do estilo é mais alto. Como a taxa de evaporação entre uma iteração e outra é baixa, há uma chance maior das formigas ficarem muitas iterações percorrendo caminhos que não produzem bons resultados, resultando assim em valores MQ's baixos.

Tabela 4.8. Resultados obtidos a partir da otimização arquitetural do software Apache Ant com estilo arquitetural em camadas.

Id	Iterações	Formigas	ρ	α	β	MQ	Rel. Penalizados
1	50	20	0.7	0.4	0.0	0.60244	0
2	100	15	0.8	0.2	0.4	0.60019	0
3	30	20	0.7	0.4	0.3	0.59660	3
4	30	20	0.7	0.4	0.7	0.59563	0
5	20	20	0.5	0.4	0.1	0.58182	8
6	20	20	0.2	0.4	0.1	0.56292	16

A Tabela 4.9 mostra a distribuição das classes dentro dos componentes considerando a melhor solução obtida, configuração Id: 1 (iteração: 50, formigas: 20, ρ: 0.7, α: 0.4, β: 0.0), a pior solução, configuração Id: 6 (iteração: 20, formigas: 20, ρ: 0.2, α: 0.4, β: 0.1) e a distribuição original dos elementos. Podemos notar que ambas as configurações em relação a distribuição original, distribuíram melhor a quantidade de classes dentro dos componentes. Deixando de existir componentes com um número muito alto de classes como na distribuição original.

Tabela 4.9. Distribuição da quantidade de classe dentro de cada componente.

	Id : 6 Iterações: 20 Formigas: 20 ρ: 0.2, α: 0.4, β: 0.1	Id: 1 Iterações: 50 Formigas: 20 ρ: 0.7, α: 0.4, β: 0.0	Distribuição Original
Id do Componente	Quantidade de Classes	Quantidade de Classes	Quantidade de Classes
0	13	19	2
1	6	2	23
2	17	26	51
3	14	2	5
4	18	4	8
5	2	22	2
6	26	21	5

A Tabela 4.10 ilustra como se dá a distribuição dos relacionamentos das classes da melhor (Id: 1) e pior (Id: 6) solução encontrada e também mostra a distribuição original dos relacionamentos da arquitetura de entrada. Os relacionamentos são divididos em relacionamentos internos e externos.

Tabela 4.10. Distribuição da quantidade de classes com relacionamentos internos e externos.

	Id: 6 Iterações: 20 Formigas: 20 ρ: 0.2, α: 0.4, β: 0.1		Id: 1 Iterações: 50 Formigas: 20 ρ: 0.7, α: 0.4, β: 0.0		Arquitetura original	
Quantidade de Classes	Rel. Internos	Rel. Externos	Rel. Internos	Rel. Externos	Rel. Internos	Rel. Externos
0	0	2	0	95	75	83
1	31	37	37	0	13	8
2	40	36	33	0	3	0
3	19	12	16	0	1	2
4	4	8	5	0	1	1
5	0	0	4	0	2	0
6	1	0	0	0	0	0

Podemos observar que ambas as configurações redistribuem os relacionamentos internos e externos de forma mais similar que a distribuição original. A configuração de Id = 1, que obteve melhor valor de MQ, não possui relacionamentos externos, o que nos leva a concluir que a métrica MQ otimiza a arquitetura buscando minimizar os relacionamentos externos pois isto diminui o acoplamento entre os componentes.

4.2.2. Apache 1.3.0

A Tabela 4.11 apresenta as configurações utilizadas para executar o ACO com a versão 1.3.0 do software Apache Ant. Podemos observar pelos valores da coluna MQ, que o ACO sugeriu

soluções com o MQ alto em relação a arquitetura de entrada que possuía MQ = 0.51541.

Tabela 4.11. Resultados obtidos pelo ACO considerando versão 1.3.0 do software Apache Ant.

Id	Iterações	Formigas	ρ	α	β	MQ
1	20	500	0.7	0.9	0.0	0.60675
2	200	100	0.7	0.4	0.0	0.59478
3	100	50	0.0	0.4	0.0	0.56587

A Tabela 4.12 mostra as configurações executadas para a versão 1.3.0 do software Apache Ant com estilo arquitetural em camadas. Podemos verificar que ambas as configurações puderam otimizaram significativamente a arquitetura em relação a arquitetura inicial que possuia MQ = 0.51541 e conseguiram também respeitar as regras do estilo em camada (coluna Rel. Penalizados está zerada).

Tabela 4.12. Resultados obtidos a partir da otimização arquitetural do software Apache Ant versão 1.3.0 com estilo arquitetural em camadas.

Id	Iterações	Formigas	ρ	α	β	MQ	Rel. Penalizados
1	20	500	0.7	0.9	0.4	0.59489	0
2	100	50	0.7	0.4	0.4	0.58079	0

4.2.3. Tempo De Execução

As Tabelas 4.14 e 4.13 mostram o tempo de execução do ACO para as configurações de cada versão do software Apache-Ant. A coluna *Versão* denota a versão do Apache-Ant, a coluna *Estilo* significa se há estilo arquitetural aplicado sobre aquela versão. As colunas *Iterações*, *Formigas*, ρ, α e β mostram a configuração executada e a coluna *Tempo de Execução* marca o tempo que o ACO demorou para executar aquela configuração para a versão marcada.

A Tabela 4.14 mostra o tempo de execução do algoritmo em uma máquina com sistema operacional Fedora, processador Intel Core i3 2.20GHz e memória de 4 Gb.

Os experimentos realizados para a versão 1.3.0 sem estilo arquitetural definido foram executados em uma máquina com as seguintes configurações: sistema operacional Ubuntu, processador Intel Dual Core 2.14 Ghz e 4 Gb de memória como mostra a Tabela 4.13.

A alteração das constantes ρ, α e β não apresentaram diferença no tempo de execução. Sendo assim constatamos ao observar a Tabela **??** que a alta quantidade de iterações ou formigas e o tamanho da arquitetura de entrada tende a aumentar o tempo de execução do algoritmo.

Tabela 4.13. Tempo de execução dos experimento em máquina com processador Dual Core.

Id	Versão	Estilo	Iterações	Formigas	ρ	α	β	Tempo De Execução
1	1.3.0	Não	20	500	0.7	0.9	0.0	13:58:58
2	1.3.0	Não	100	50	0.0	0.4	0.0	06:46:03

Tabela 4.14. Tempo de execução dos experimento em máquina com processador i3

Id	Versão	Estilo	Iterações	Formigas	ρ	α	β	Tempo De Execução
1	1.1.0	Não	100	15	0.2	0.4	0.2	00:04:32
2	1.1.0	Não	20	20	0.6	0.4	0.1	00:01:21
3	1.1.0	Sim	50	20	0.7	0.4	0.0	00:03:30
4	1.1.0	Sim	20	20	0.2	0.4	0.1	00:01:31
5	1.3.0	Sim	20	500	0.7	0.9	0.4	22:36:35
6	1.3.0	Sim	100	50	0.7	0.4	0.4	09:20:31

4.3. Discussão

Esta seção apresenta discussões sobre os resultados obtidos pelo método proposto neste trabalho. A Figura 4.1 apresenta a evolução dos valores MQ obtidos pelo algoritmo considerando estilo arquitetural em camadas. É possível verificar a evolução do MQ para três configurações diferentes, a linha azul representa a configuração de Id: 1 (iteração = 50, formigas = 20, $\rho = 0.7$, $\alpha = 0.4$ e $\beta = 0.0$) que obteve MQ = 0.60244, a linha vermelha representa a configuração Id: 3 (iteração = 30, formigas = 20, $\rho = 0.7$, $\alpha = 0.4$ e $\beta = 0.7$) com MQ = 0.59648 e a linha laranjada representa a configuração de Id : 4 (iteração = 30, formigas = 20, $\rho = 0.7$, $\alpha = 0.4$ e $\beta = 0.3$) com MQ = 0.59298, todas as configurações estão referenciadas na Tabela 4.8. Observa-se que as configurações, após convergirem para a melhor solução, ficaram algumas iterações produzindo os mesmos valores MQ's pois já haviam encontrado seu valor máximo. Esta observação nos leva a concluir que considerar como condição de parada do algoritmo um número máximo de iterações não é uma boa alternativa, por causa do tempo de execução desperdiçado produzindo soluções iguais. O ideal seria adotar como critério de parada a diferença entre os valores da função objetivo entre uma iteração e outra. Podemos observar ainda que, a variação do parâmetro β causou impacto nas soluções, fazendo com que a configuração com β maior convergisse mais rápido para o resultado bom.

A Figura 4.2, referente as mesmas configurações listadas acima, mostra a evolução das penalidades. Podemos observar o valor utilizado de β nas configurações e o que isto causou na evolução das penalidades ao longo das iterações. A linha azul que representa a configuração de Id:1 e possui $\beta = 0.0$, iniciou as iterações com soluções que infringiam muito as regras do estilo, enquanto a configurações com $\beta = 0.7$ (linha vermelha) e 0.3 (linha laranjada) produziram soluções que infringiam menos as regra do estilo ao longo das iterações. No entanto, apesar deste comportamento, ambas as configurações acabaram chegando a penalidade igual a zero,

independente dos valores de β. Constatamos ao longo dos testes feitos que os valores de β, considerando o estilo arquitetural em camadas, acabam sendo invalidados pela função MQ, isso pelo fato de que a regra do estilo arquitetural em camadas penaliza relacionamentos entre classes onde: a classe a do relacionamento está em um componente na camada x e a classe b está em um componente na camada $x + 2$. Dessa maneira, essa penalidade está condicionada a existência de um relacionamento externo entre as classes a e b. Como dito anteriormente, a função MQ busca reduzir a zero o valor de relacionamentos externos. Dessa maneira, o valor de β não influencia, pois a função objetivo já reduz os relacionamentos externos o que automaticamente fará com que não haja penalidades entre os relacionamentos.

A Figura 4.3 mostra a distribuição das classes nos componentes. A barra azul representa a configuração Id: 1 (iteração = 50, formigas = 20, $\rho = 0.7$, $\alpha = 0.4$ e $\beta = 0.0$), a barra vermelha representa a configuração de Id: 6 (iteração = 20, formigas = 20, $\rho = 0.2$, $\alpha = 0.4$ e $\beta = 0.1$) ambas as configurações listadas na Tabela 4.8. A barra laranjada mostra como era a distribuição das classes do modelo original. Para ambas as configurações podemos notar a redistribuição mais similar das classes nos componentes, enquanto a solução inicial possuía ou componentes com poucas classes ou com muitas classes. Nas soluções propostas pelo ACO podemos perceber uma distribuição mais similar da quantidade de classes dentro dos componentes, ou seja, não há componentes com uma quantidade muito alta de classes. Podemos notar ainda que a configuração que obteve resultado de MQ maior, representado pela barra azul, produziu componentes com mais classes que a solução com MQ inferior.

A Figura 4.4 mostra como se dá a distribuição de relacionamentos internos e externos obtidas pelas configurações já citadas anteriormente e comparando com a distribuição da arquitetura original.

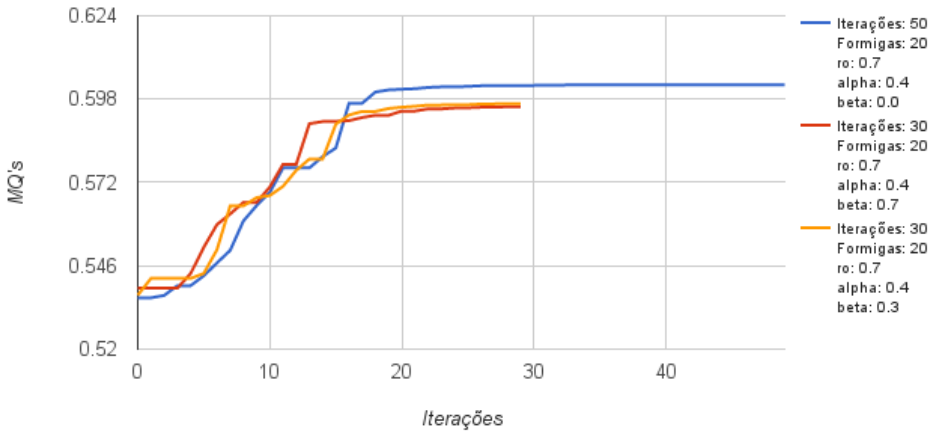

Figura 4.1. Evolução do valor da função objetivo MQ para Apache Ant 1.1.0 com estilo em camadas.

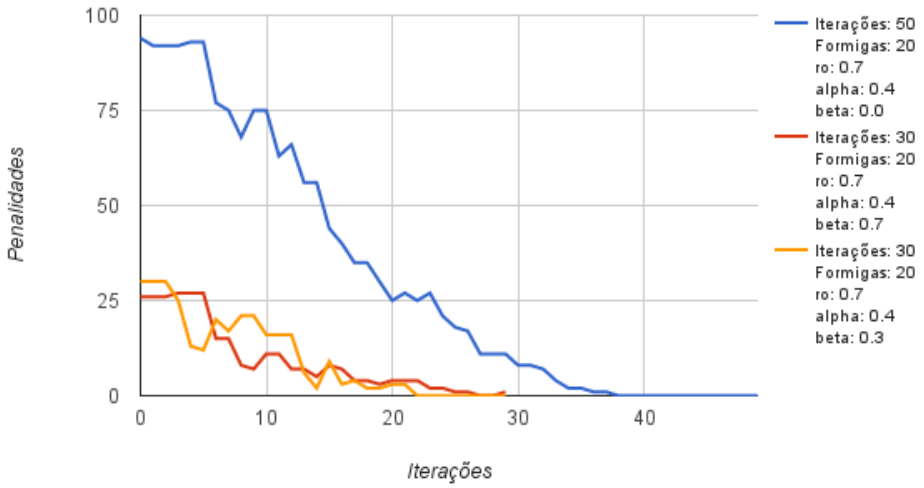

Figura 4.2. Evolução das penalidades atribuídas as soluções geradas ao longo das iterações.

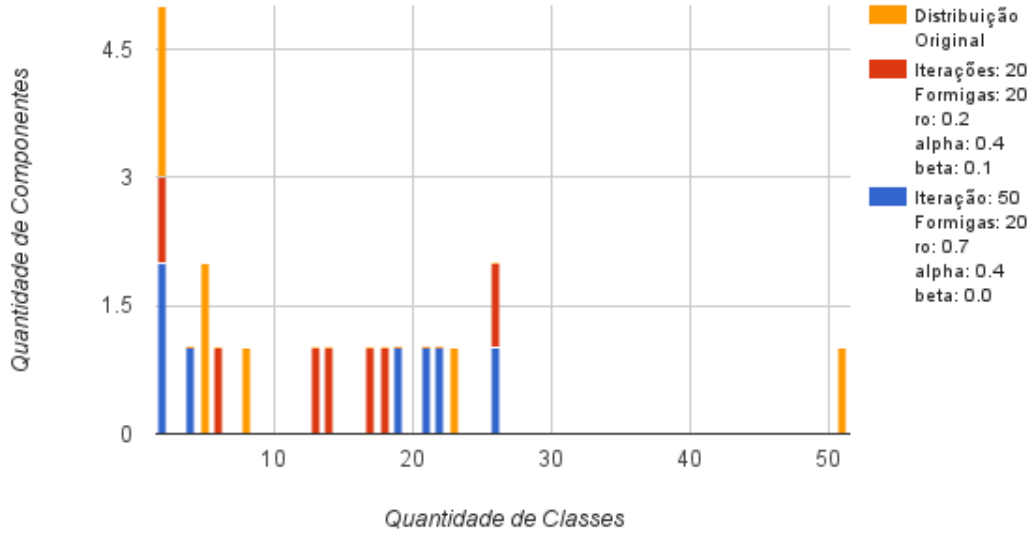

Figura 4.3. Distribuição das classes em seus respectivos componentes (Apache Ant com estilo em camadas).

A barra vermelha (relacionamento interno) e laranjada (relacionamento externo) representam a configuração de Id: 6 (iteração = 20, formigas = 20, $\rho = 0.2$, $\alpha = 0.4$ e $\beta = 0.1$)que obteve MQ = 0.56292. As barras azul (relacionamento interno) e verde (relacionamento externo) representam a configuração de Id: 1 (iteração = 50, formigas = 20, $\rho = 0.7$, $\alpha = 0.4$ e $\beta = 0.0$) ambas listadas na Tabela 4.8. Por fim, as barras roxa e preta representam a distribuição dos relacionamentos originais, da arquitetura de entrada. Podemos notar que a configuração com valor de MQ alto tem o valor de relacionamentos externos zerado, enquanto

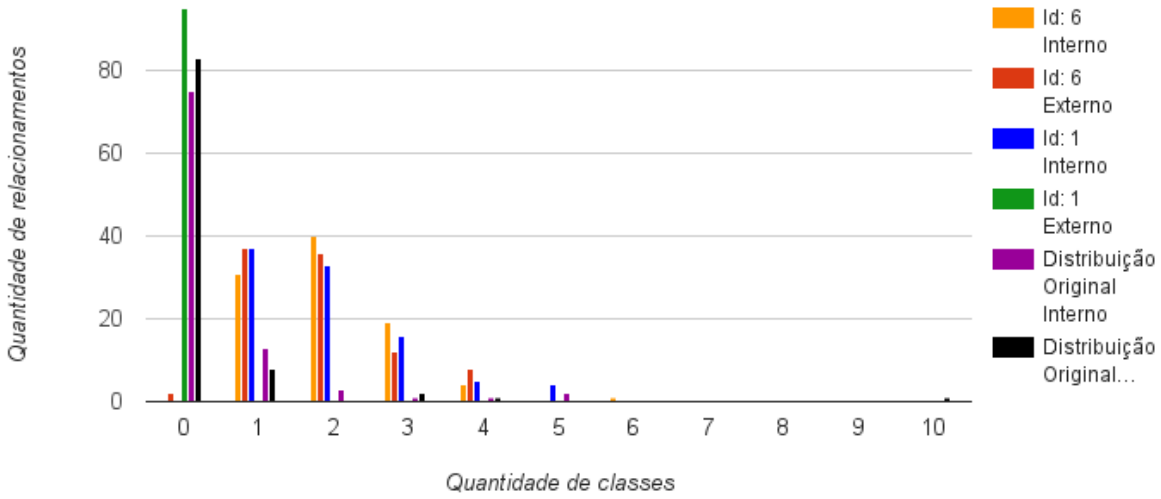

Figura 4.4. Quantidade de classes e de relacionamentos internos e externos (Apache Ant com estilo em camadas).

a configuração com baixo MQ possui muitos relacionamentos externos. Portanto concluímos que quando menor o número de relacionamentos externos maior o valor de MQ. Porém, podemos observar que a configuração de MQ baixa (representada pela barra vermelha e laranjada) possui mais relacionamentos externos que a arquitetura de entrada, sem otimização. No entanto, cabe observar que, apesar disto, a versão otimizada faz um rebalanceamento da quantidade de relacionamentos internos, que a solução de entrada praticamente não possuía o que aumenta a coesão da arquitetura.

As Figuras 4.5 e 4.6 mostram a distribuição das classes dentro dos componentes e a distribuição dos relacionamentos internos e externos referente a versão 1.3.0 do software Apache Ant sem estilo arquitetural definido. As configurações representadas nas imagens são as de Id: 1 e Id: 3 da Tabela 4.11.

A Figura 4.5 mostra a comparação entre duas configurações e a distribuição original da arquitetural. A barra azul representa a configuração de Id: 1 da Tabela 4.11 que obteve MQ = 0.60675. A barra vermelha mostra a configuração Id: 3 da mesma tabela com MQ = 0.56587. Nota-se que a quantidade de componentes com muitas classes é baixa, temos o máximo de 9 componentes com 6 classes (barra vermelha). Podemos perceber que não há uma diferença grande de distribuição entre as três arquiteturas.

A Figura 4.6 mostra o número de classes com relacionamentos internos e externos para as configuração Id: 1, Id: 3 da Tabela 4.11 e a distribuição da arquitetura original. As barras laranjada e vermelha representam a configuração de Id: 3. As barras azul e verde a configuração de Id: 1 e as barras roxa e preta a distribuição da arquitetural original. Podemos

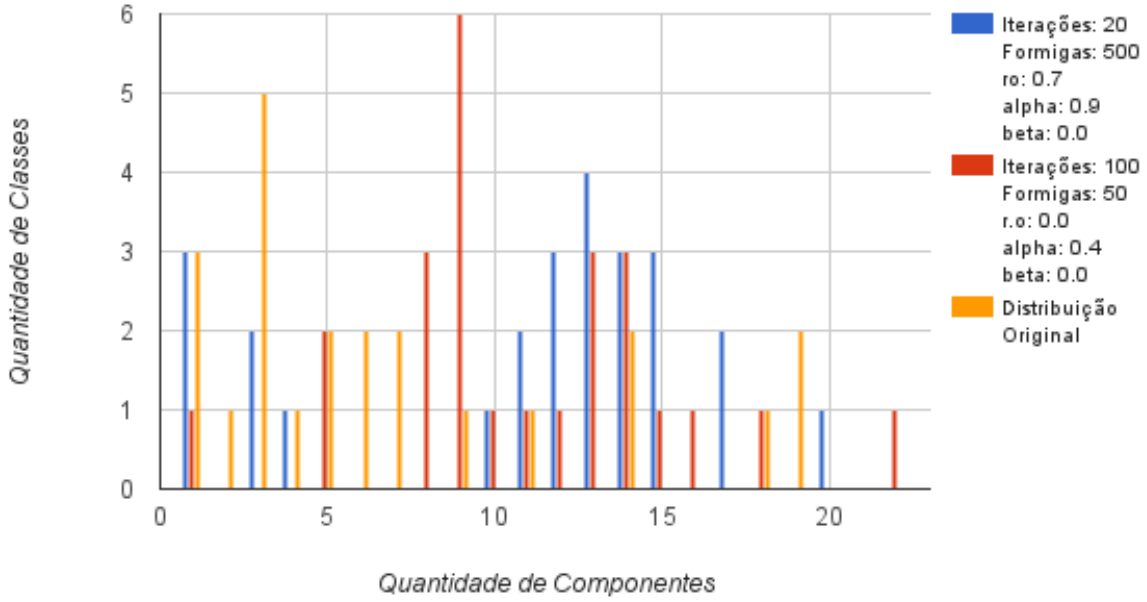

Figura 4.5. Quantidade de classes distribuidas em cada componentes.

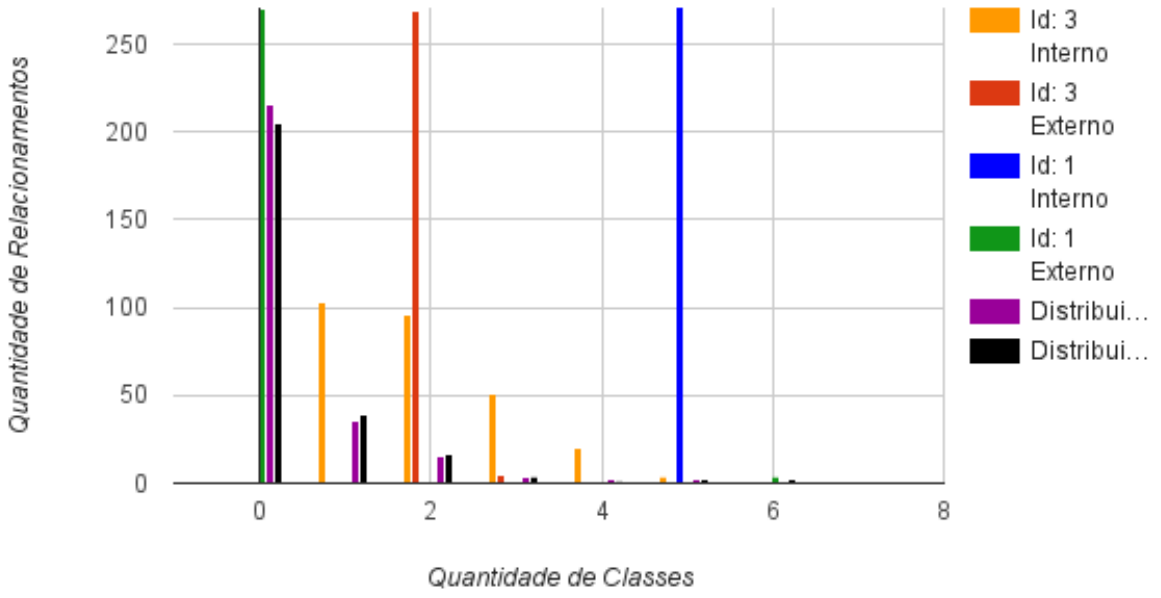

Figura 4.6. Quantidade de classes e de relacionamentos internos e externos Apache Ant 1.3.0.

notar que a configuração de Id: 1 que resultou no melhor valor de MQ obteve um número de relacionamentos externos igual a zero enquanto que o número de relacionamentos internos foi de 5 classes com 255 relacionamentos internos. Esta característica de ter um alto número de

relacionamentos internos para poucas classes não havíamos notado para a versão do Apache Ant 1.1.0 (Figura 4.3).

4.4. Considerações Finais

Nesta seção apresentamos os resultados obtidos pelo ACO para otimização da arquitetura das versões 1.1.0 e 1.3.0 do software Apache-Ant. Podemos observar que o algoritmo, considerando a métrica MQ produziu bons resultados de saída, otimizando as arquiteturas. Dado o alto tempo de execução quando considerada a versão 1.3.0, percebemos que conforme a complexidade da arquitetura de entrada aumenta, o tempo gasto pelo algoritmo durante a otimização também aumenta. Notamos ainda que, por causa característica do MQ de zerar os relacionamentos externos da arquitetura, a métrica utilizada para observar o estilo arquitetural acabou sendo invalidada.

Capítulo 5

Conclusões

A área de otimização arquitetural apresenta-se ativa e necessária de ser explorada, para que possamos descentralizar o processo de modelagem de arquitetura do engenheiro de software e consequentemente tornar o processo de definição da arquitetura menos aberto a erros, que podem causar um alto custo ao software. Este trabalho propôs a utilização da metaheuristica otimização por colônia de formigas para otimização de arquitetura de software.

A modelagem de arquitetura de software tem sido estudada durante décadas e diversos conceitos e ferramentas foram propostos para o desenvolvimento desta fase. Utilizando linguagens de descrição arquitetural, o processo de modelagem arquitetural pode ser padronizada e torna mais fácil sua compreensão. Porém ainda assim, este é um processo altamente crítico, pois decisões tomadas nesta fase podem causar alto custo ao sistema caso precisem ser refatoradas. Outro ponto crítico é que como este é um processo desempenhado por seres humanos, há grandes possibilidades de erros, além do que o ensino do ofício de modelagem arquitetural não é algo trivial, tornando assim difícil passar o conhecimento adiante.

Dessa maneira métodos de busca que utilizam métricas de qualidade arquiteturais para encontrar a melhor arquitetura possível para um dado software é algo interessante. Estes métodos têm o poder de auxiliar o engenheiro de software no processo de modelagem e reduzir custos de refatoração do sistema. Este tipo de abordagem, busca por uma melhor arquitetura baseada em métricas de qualidade, sendo assim a escolha das métricas é um passo importante quando se considera a utilização de métodos de busca para otimização arquitetural. A utilização de heuristicas e metaheuristica em problemas recorrentes da engenharia de software faz parte de uma área de pesquisa chamada engenharia de software baseada em busca.

Neste trabalho exploramos a utilização da metaheuristica otimização por colônia de

formigas para resolver o problema de otimização de arquitetura de software. A implementação desta abordagem proporcionou um melhor conhecimento de como o ACO se comporta para este tipo de problema. A métrica utilizada foi *Modularization Quality* (MQ), que busca reduzir acoplamento e aumentar a coesão dos componentes da arquitetura e a métrica proposta neste trabalho que verifica o estilo arquitetural do sistema, aplicando penalidades as soluções que quebram regras do estilo arquitetural imposto.

Sendo assim, neste trabalho apresentamos uma releitura dos conceitos que compõem a arquitetura de software, os conceitos da metaheuristica ACO, das métricas utilizadas, a implementação da ferramenta de software chamada *ACO2Arcitecure*, experimentos e os resultados obtidos pela abordagem.

Os experimentos foram conduzidos para que pudéssemos averiguar se o ACO era apto a obter resultados satisfatórios em relação a arquitetura inicial do sistema. Para a execução dos experimentos aplicamos a abordagem proposta sobre duas versões do software Apache Ant. Os resultados obtidos mostraram que o ACO tem capacidade para produzir resultados satisfatórios em relação a arquitetura de entrada considerando as métricas usadas. Dessa maneira, os resultados obtidos pelo algoritmo mostraram melhores valores MQ que a arquitetura inicial. Portanto o ACO se mostra uma opção interessante de ser utilizada para problemas de otimização arquitetural.

As ameaças a validade deste trabalho, são: a forma de extração a arquitetura, onde desconsideramos a análise do corpo dos métodos de um projeto. Poucos casos de testes aplicados, necessidade de utilização do ACO em mais arquiteturas.

Dado o desenvolvimento deste trabalho podemos apontar alguns trabalhos futuros necessários para melhorar a utilização do ACO para otimização arquitetural. Experimentos com outras metaheuristicas, a fim de comparar os resultados de diferentes abordagens. Experimentos utilizando arquiteturas com estilo cliente/servidor, para analisar como se comporta a métrica de penalidades proposta. Dado o tempo de execução elevado que o ACO obteve para arquiteturas complexas, identificamos a necessidade da utilização de técnicas de otimização para o algoritmo. Utilização de outras métricas de qualidade para verificar a capacidade do ACO de lidar com funções multiobjetivas e a implementação de outros estilos arquiteturais.

Referências

ALETI, Aldeida; BUHNOVA, Barbora; GRUNSKE, Lars; KOZIOLEK, Anne; MEEDENIYA, Indika. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 39, n. 5, p. 658–683, maio 2013. ISSN 0098-5589.

Apache Software Foundation. *Ant*. jan. 2000. Programa de Computador. Disponível em: `http://ant.apache.org/`.

BARROS, Márcio Oliveira; FARZAT, Fábio de Almeida; TRAVASSOS, Guilherme Horta. Learning from optimization: A case study with apache ant. *Information and Software Technology*, Elsevier, v. 57, p. 684–704, 2015.

BECKER, Steffen; KOZIOLEK, Heiko; REUSSNER, Ralf. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, Elsevier, v. 82, n. 1, p. 3–22, 2009.

BIANCHI, Leonora; DORIGO, Marco; GAMBARDELLA, Luca Maria; GUTJAHR, Walter J. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, v. 8, n. 2, p. 239–287, 2008.

BLUM, Christian; LI, Xiaodong. *Swarm intelligence in optimization*. Berlin, Heidelberg: Springer, 2008.

BLUM, Christian; ROLI, Andrea. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 3, p. 268–308, set. 2003. ISSN 0360-0300.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1999.

BOSCH, Jan. Software architecture: The next step. In: OQUENDO, Flavio; WARBOYS, BrianC.; MORRISON, Ron (Ed.). *Software Architecture*. St Andrews, UK: Springer Berlin Heidelberg, 2004, (Lecture Notes in Computer Science, v. 3047). p. 194–199. ISBN 978-3-540-22000-8.

CAI, Xia; LYU, M. R.; WONG, Kam-Fai; KO, Roy. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In: *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, 2000. p. 372–379. ISSN 1530-1362.

CHHABRA, Jitender Kumar *et al.* Preserving core components of object-oriented packages while maintaining structural quality. *Procedia Computer Science*, Elsevier, v. 46, p. 833–840, 2015.

CHIDAMBER, Shyam R; KEMERER, Chris F. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, IEEE, v. 20, n. 6, p. 476–493, 1994.

CLEMENTS, Paul C. *Software architecture in practice*. Tese (Doutorado) — Software Engineering Institute, 2002.

COLANZI, T.E.; VERGILIO, S.R. A feature-driven crossover operator for product line architecture design optimization. In: *IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), 2014*, 2014. p. 43–52.

DIJKSTRA, Edsger W. Complexity controlled by hierarchical ordering of function and variability. Notas of Edsger W. Dijkstra. 1968. Disponível em: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD236.html`.

GARLAN, David. Software architecture: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*, 2000. (ICSE '00), p. 91–101. ISBN 1-58113-253-0.

GARLAN, David. Software architecture: A travelogue. In: *Proceedings of the on Future of Software Engineering*, 2014. (FOSE 2014), p. 29–39. ISBN 978-1-4503-2865-4.

GARLAN, David; MONROE, Robert T.; WILE, David. Foundations of component-based systems. In: LEAVENS, Gary T.; SITARAMAN, Murali (Ed.). New York, NY, USA: Cambridge University Press, 2000. cap. Acme: Architectural Description of Component-based Systems, p. 47–67.

GARLAN, David; SHAW, Mary. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, Singapore, v. 1, n. 3.4, 1993.

GARLAN, David; SHAW, Mary. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

GLOVER, Fred. Tabu search-part i. *ORSA Journal on computing*, INFORMS, Norwell, MA, USA, v. 1, n. 3, p. 190–206, 1989.

GLOVER, F.W.; KOCHENBERGER, G.A. *Handbook of Metaheuristics*. Springer US, 2003. (International Series in Operations Research & Management Science). ISBN 9781402072635. Disponível em: `https://books.google.com.au/books?id=0\_10T\_KeqOgC`.

GLOVER, Fred; LAGUNA, Manuel. In: *Handbook of Combinatorial Optimization*. Norwell, MA, USA: Springer New York, 2013. p. 3261–3362.

GRUNSKE, Lars. Identifying "good"architectural design alternatives with multi-objective optimization strategies. In: *Proceedings of the 28th International Conference on Software Engineering*, 2006. (ICSE '06), p. 849–852. ISBN 1-59593-375-1.

GRUNSKE, Lars. Early quality prediction of component-based systems - a generic framework. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 80, n. 5, p. 678–686, 2007. ISSN 0164-1212.

GRUNSKE, Lars; LINDSAY, Peter; BONDAREV, Egor; PAPADOPOULOS, Yiannis; PARKER, David. An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In: LEMOS, Rogério de; GACEK, Cristina; ROMANOVSKY, Alexander (Ed.). *Architecting Dependable Systems IV*. Berlin: Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4615). p. 188–209. ISBN 978-3-540-74033-9.

HARABAGIU, Sanda; MILLER, George; MOLDOVAN, Dan. Wordnet 2-a morphologically and semantically enhanced resource. In: *Proceedings of SIGLEX*, 1999. v. 99, p. 1–8.

HARMAN, M.; JIA, Y.; KRINKE, J.; LANGDON, W. B.; PETKE, J.; ZHANG, Y. Search based software engineering for software product line engineering: A survey and directions for future work. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*, 2014. (SPLC '14), p. 5–18. ISBN 978-1-4503-2740-4.

HARMAN, Mark; JONES, Bryan F. Search-based software engineering. *Information and Software Technology*, v. 43, n. 14, p. 833 – 839, 2001. ISSN 0950-5849.

HARMAN, Mark; MANSOURI, S. Afshin; ZHANG, Yuanyuan. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300.

HARRIS, J.; HIRST, J.L.; MOSSINGHOFF, M. *Combinatorics and Graph Theory*. New York: Springer New York, 2008. (Undergraduate Texts in Mathematics).

HASHEMINEJAD, Seyed Mohammad Hossein; JALILI, Saeed. An evolutionary approach to identify logical components. *Journal of Systems and Software*, Elsevier, v. 96, p. 24–50, 2014.

HUANG, Gang; MEI, Hong; YANG, Fu-Qing. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, Kluwer Academic Publishers, p. 257–281, 2006.

HUSSAIN, Ibrar; KHANUM, Aasia; ABBASI, Abdul Qudus; JAVED, Muhammad Younus. A novel approach for software architecture recovery using particle swarm optimization. *International Arab Journal of Information Technology (IAJIT)*, v. 12, n. 1, 2015.

JIFENG, He; LI, Xiaoshan; LIU, Zhiming. Component-based software engineering. In: *Theoretical Aspects of Computing–ICTAC 2005*. St. Louis, USA: Springer, 2005. p. 70–95.

KOZIOLEK, Heiko; REUSSNER, Ralf. A model transformation from the palladio component model to layered queueing networks. In: *Performance Evaluation: Metrics, Models and Benchmarks*. Darmstadt, Germany: Springer, 2008. p. 58–78.

LI, Rui; ETEMAADI, Ramin; EMMERICH, Michael TM; CHAUDRON, Michel RV. An evolutionary multiobjective optimization approach to component-based software architecture design. In: Ieee. *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2011. p. 432–439.

MAGEE, Jeff; DULAY, Naranker; EISENBACH, Susan; KRAMER, Jeff. Specifying distributed software architectures. In: *Proceedings of the 5th European Software Engineering Conference*, 1995. p. 137–153. ISBN 3-540-60406-5. Http://www-dse.doc.ic.ac.uk/Research/Darwin/.

MANCORIDIS, S.; MITCHELL, B. S.; CHEN, Y.; GANSNER, E. R. Bunch: a clustering tool for the recovery and maintenance of software system structures. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999. p. 50–59. ISSN 1063-6773.

MARIANI, Thainá. *PRESERVANDO O ESTILO ARQUITETURAL NO PROJETO BASEADO EM BUSCA DE LINHA DE PRODUTO DE SOFTWARE*. Dissertação (Mestrado) — Universidade Feredal do Paraná, UFPR, 2015.

MARTENS, Anne; BROSCH, Franz; REUSSNER, Ralf. Optimising multiple quality criteria of service-oriented software architectures. In: Acm. *Proceedings of the 1st international workshop on Quality of service-oriented software systems*, 2009. p. 25–32.

MARTENS, Anne; KOZIOLEK, Heiko; BECKER, Steffen; REUSSNER, Ralf. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: Acm. *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, 2010. p. 105–116.

MICROSOFT, Developer Network. *Architectural Patterns and Styles*. 2015. Disponível em: https://msdn.microsoft.com/en-us/library/ee658117.aspx.

MIRANDOLA, Raffaela; POTENA, Pasqualina; SCANDURRA, Patrizia. Adaptation space exploration for service-oriented applications. *Science of Computer Programming*, Elsevier, v. 80, p. 356–384, 2014.

MUELLER, Carsten. Multi-objective optimization of software architectures using ant colony optimization. *Lecture Notes on Software Engineering*, IACSIT Press, v. 2, n. 4, p. 371, 2014.

PARNAS, David Lorge. Information distribution aspects of design methodology. 1971.

POUR, Gilda. Software component technologies: Javabeans and activex. In: Ieee. *tools*, 1998. p. 375.

RAMÍREZ, Aurora; ROMERO, José Raúl; VENTURA, Sebastián. A novel component identification approach using evolutionary programming. In: Acm. *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, 2013. p. 209–210.

RAMíREZ, Aurora; ROMERO, José Raúl; VENTURA, Sebastián. An approach for the evolutionary discovery of software architectures. *Information Sciences*, v. 305, n. 0, p. 234 – 255, 2015. ISSN 0020-0255.

RECHTIN, Eberhardt. *Systems Architecting: creating and building complex systems*. New Jersey: Prentice Hall Englewood Cliffs, NJ, 1991.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Boston, MA, USA: Pearson Higher Education, 2004. ISBN 0321245628.

RUSSELL, Stuart J.; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 2. ed. Upper Saddle River, NJ, USA: Pearson Education, 2003. ISBN 0137903952.

SAED, A.A.A.; KADIR, W.M.N.W. Applying particle swarm optimization to software performance prediction an introduction to the approach. In: *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, 2011. p. 207–212.

SEI. *Defining Software Architecture*. 2015. Disponível em: http://www.sei.cmu.edu/architecture/.

SIMONS, C.L.; SMITH, J.E. A comparison of meta-heuristic search for interactive software design. *Soft Computing*, Springer Berlin Heidelberg, v. 17, n. 11, p. 2147–2162, 2013. ISSN 1432-7643.

TAWOSI, Vali; JALILI, Saeed; HASHEMINEJAD, Seyed Mohammad Hossein. Automated software design using ant colony optimization with semantic network support. *Journal of Systems and Software*, v. 109, p. 1 – 17, 2015. ISSN 0164-1212.

VATHSAVAYI, Sriharsha; KOSKIMIES, Kai *et al.* Interleaving human and search-based software architecture design. In: Estonian academy publishers. *Proceedings of the Estonian Academy of Sciences. Chemistry*, 2013. v. 62, n. 1, p. 16–26.

VITHARANA, Padmal. Risks and challenges of component-based software development. *Commun. ACM*, ACM, New York, NY, USA, v. 46, n. 8, p. 67–72, ago. 2003.

WHITLEY, Darrell. A genetic algorithm tutorial. *Statistics and computing*, Springer, v. 4, n. 2, p. 65–85, 1994.