Um estudo sobre o uso do algoritmo de Colônia de Formigas para otimização de arquiteturas baseadas em componentes

Mariane Affonso Medeiros[1], **Filipe Roseiro Côgo**[1], **Marco Aurélio Graciotto Silva**[1]

[1] Departamento Acadêmico de Computação
Universidade Tecnológica Federal do Paraná
Campo Mourão, PR

mmedeiros@alunos.utfpr.edu.br

{filiper,magsilva}@utfpr.edu.br

Resumo. *O design arquitetural é uma fase crítica no desenvolvimento de software, pois decisões tomadas nesta fase têm um impacto significativo no custo e qualidade do sistema final. Para auxiliar engenheiros de software a definir arquiteturas adequadas, métodos de otimização arquitetural vêm sendo utilizados para propor diretrizes e recomendações a fim de identificar elementos arquiteturais, recuperar e otimizar arquiteturas. Este trabalho investiga a utilização e o comportamento da metaheurística Otimização por Colônia de Formigas (ACO) para otimização de arquiteturas de software baseado em componentes considerando métricas de coesão, acoplamento e o estilo arquitetural. A abordagem proposta foi avaliada a partir de experimentos, aplicando o algoritmo em um sistema real. Considerando as métricas adotadas, o ACO obteve resultados cerca de 11% melhor que o valor obtido pela arquitetura inicial.*

1. Introdução

A arquitetura de software é uma descrição de sistemas que auxilia na compreensão do comportamento de software [Garlan 2014]. Ela representa diversos atributos de qualidade do sistema, tais como desempenho, modificabilidade e segurança, além disso, a construção de uma arquitetura efetiva permite identificar riscos de design e mitigá-los nos primeiros estágios de desenvolvimento [SEI 2015].

Com a crescente demanda por sistemas de software complexos, o design arquitetural se tornou uma importante atividade, demandando o desenvolvimento de técnicas que auxiliem na construção de arquiteturas de software com qualidade [Garlan 2000]. Embora os arquitetos de software aprendam o ofício de construção arquitetural através de experiências vividas por diversos projetos passados [Garlan 2000] é interessante que 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 pertence a uma área de pesquisa chamada Engenharia de Software Baseada em Busca [Harman e Jones 2001]. A ideia central desta área é a conversão de problemas da Engenharia de Software em problemas de busca. A otimização arquitetural pode ser modelada como um problema de busca, considerando que utiliza critérios pré-estabelecidos e características do sistema para alterar a arquitetura até que

esta esteja na condição mais aceitável possível. Esta condição é medida por métricas de qualidade que avaliam a qualidade arquitetural. Este trabalho utiliza a metaheurística Otimização por Colônia de Formigas (ACO) para otimizar arquiteturas de software.

Algoritmos ACO utilizam uma colônia de formigas artificiais para solucionar problemas. Estas formigas se movimentam em um espaço de busca em várias direções a fim de construir uma solução para o problema em questão. 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 explorado. O feromônio depositado nos caminhos é utilizado pelas formigas durante o processo de construção da solução. 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. O ACO pode ainda utilizar uma informação heuristica para auxiliar na construção da solução.

O objetivo deste trabalho é observar o comportamento do algoritmo Otimização por Colônia de Formigas para otimização de arquiteturas de software baseada em componentes quanto a coesão, acoplamento e preservação do estilo arquitetural. Dessa forma, espera-se a diminuição do esforço para reparação da arquitetura pelo engenheiro de software durante o desenvolvimento do sistema.

O restante deste trabalho esta organizado da seguinte forma. A Seção 2 apresenta a abordagem de otimização baseada em ACO. Os resultados quanto à aplicação da abordagem são discutidos na Seção 3. Trabalhos relacionados são tratados na Seção 4. Por fim, a Seção 5 discorre sobre as conclusões e trabalhos futuros.

2. Abordagem Proposta

Esta seção descreve as etapas para o desenvolvimento do algoritmo de otimização baseado em colônia de formigas feito neste trabalho, denominado *ACO2SoftwareArchitecture*.

2.1. Representação da arquitetura

O algoritmo utiliza como entrada uma arquitetura baseada em componentes. Esta arquitetura deve ser representada por um modelo UML, escrito em arquivo XMI, no qual devem estar presentes elementos da arquitetura como: relacionamentos, classes e componentes. Caso não exista um arquivo XMI, o modelo UML pode ser extraído a partir do código fonte com o auxílio da ferramenta Modisco[1]. Para as arquiteturas recuperadas a partir de código fonte e que não possuem componentes definidos explicitamente, os pacotes do projeto foram considerados como componentes.

O algoritmo parte do modelo UML para extrair relacionamentos, classes e componentes da arquitetura. Para fazer esta extração, utilizamos a API UML2[2], para identificar os elementos do modelo UML e extrair a quantidade de classes, componentes, relacionamentos entre classes e componentes e os relacionamentos entre as classes da arquitetura.

Como uma das propostas deste trabalho é, além da otimização da arquitetura, a preservação do estilo arquitetural, é possível informar ao algoritmo projetos com o estilo

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

[2] `http://www.eclipse.org/modeling/mdt/?project=uml2`.

arquitetural definido no modelo UML. Para isso, utilizamos Perfis (*Profiles*) e Esteriótipos (*Stereotypes*) para anotar o estilo no modelo UML. O perfil UML provê uma forma genérica para customizar ou personalizar modelos UML para domínios e plataformas específicas [Rumbaugh et al. 2004], definindo estereótipos, *tagged values* ou restrições que podem ser aplicados em determinados elementos do modelo. Sendo assim, criamos um Perfil chamado *ArchitectureStyle* que representa o estilo arquitetural, restringindo-nos, neste trabalho, a representar o estilo arquitetural em camadas. Ele contém o estereótipo chamado *Layered* que representa o tipo de estilo arquitetural considerado. Este estereótipo possui uma variável *id*, que contém o nome da camada a que o componente pertence.

O algoritmo ACO proposto neste trabalho 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.

O ACO desenvolvido neste trabalho possui duas matrizes de feromônio: uma representando os componentes e outra representando os relacionamentos entre as classes. 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. Conforme as formigas constroem suas soluções e as melhores são selecionadas, esta matriz é atualizada. A matriz que representa os relacionamentos entre classes possui uma coluna N, a qual representa a possibilidade de não combinar a classe C_x com outras, ou seja, C_x não ter relacionamentos.

2.2. Métricas e função objetivo

Uma solução proposta pela metaheurística precisa ser avaliada através de uma métrica de qualidade. As métricas podem ser representadas em termos de equações para que se possa utilizá-las como função objetivo da metaheurística. 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. Coesão mede quanto as classes de um componente estão intra-relacionadas (intra-conectividade) e acoplamento mede quão inter-dependentes (inter-conectividade) são dois componentes [Saed e Kadir 2011]. A intra-conectividade representa relacionamentos entre classes que estão em um mesmo componente. Inter-conectividade representa dependência entre componentes, ou seja, relacionamentos entre classes que pertencem a componentes diferentes.

As medidas de intra e inter-conectividade podem ser encapsuladas na métrica *Modularization Quality* (MQ), que determina a qualidade de um grafo de dependência de módulos. A métrica MQ será utilizada como função objetivo do algoritmo ACO. Os valores desta métrica 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 metaheurística irá buscar pelo maior valor de MQ possível.

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

Estabelecida a representação inicial da arquitetura e a função objetivo, procedemos para a execução das iterações de otimização, ao final das quais o algoritmo retornará um arquivo texto que representa a solução com o melhor valor da função objetivo obtida. 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 e de classe $\times$ classe, fazendo as seguintes combinações: da classe i dentro do componente j, da classe i com a classe j e calcula a probabilidade destas combinações. Esta probabilidade é dada pela Equação 1, cujo conjunto $Element$ pode ser tanto os componentes quanto as classes da arquitetura, dessa forma a variável m assume valor das classes e componentes dependendo do conjunto em questão. Caso seja componente, a equação representa a probabilidade da formiga k inserir a classe i no componente j na iteração t. Caso seja classe, ela estabelece a probabilidade de relacionamento entre duas classes.

$$P_{i,j}^{k}(t) = \frac{\tau_{i,j}^{\alpha}(t) \cdot \eta_{i,j}^{\beta}(t)}{\sum_{m=0}^{|Element|} \tau_{i,m}^{\alpha}(t) \cdot \eta_{i,m}^{\beta}(t)} \tag{1}$$

- $\tau_{i,j}$ 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 2.4;
- α e β são parâmetros que salientam a importância da informação definida pelo feromônio (τ) e da informação heurística definida (η).
- $Element$ é o conjunto de componentes e de classes da arquitetura, representado no algoritmo por uma matriz de componentes e uma matriz de classes.

Cada probabilidade calculada é inserida em um vetor de probabilidades. Ao final da linha é utilizado o método da roleta a fim de escolher uma das probabilidades. Esta solução parcial é armazenada e a formiga continua construindo sua solução, até o fim da matriz. Ao final destes procedimentos a formiga k terá sua solução construída.

No processo de penalizações as arquiteturas que infringem o estilo arquitetural têm por objetivo analisar quais combinações de classes quebram a regra do estilo em camada. Essa informação servirá para a informação heurística no momento do cálculo da probabilidade de combinação entre as classes.

Após todas as formigas construírem suas soluções, é verificado qual possui o melhor valor de função objetivo. Selecionada a melhor solução, os valores de feromônio da matriz de classe por componente e de classes por classe são atualizados. A atualização do feromônio é feita baseada na Equação 2.

$$\tau_{i,j}(t+1) = (1-\rho) \cdot \tau_{i,j}(t) + \Delta\tau_{i,j}(t), \tag{2}$$

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 para que, soluções com valores ruins de função objetivo acabem sendo desvalorizadas e não sejam mais selecionadas.

2.4. 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. O estilo arquitetural considerado para este trabalho foi: estilo arquitetural em camada. 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. O estilo arquitetural em camadas possui algumas regras, que são: um elemento de uma camada x só pode se relacionar com um elemento de uma camada $x + 1$ ou um elemento da camada x [Mariani et al. 2016].

Para verificar se a arquitetura infringe o estilo arquitetural, percorre-se a matriz de relacionamentos classe$\times$classe e verifica se o relacionamento entre a classe i e j pertencentes à camadas $layer_i$ e $layer_j$, quebra as regras do estilo em questão. Cada relacionamento que quebra a regra 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, seja possível 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:

$$\eta_{i,j} = 1 - \frac{total_i + total_j}{totalGeralDeQuebrasDaRegra} \tag{3}$$

A Equação 3, que caracteriza quebras de regras do estilo arquitetural para um relacionamento entre as classes i, j, é a informação heurística ($\eta_{i,j}$) utilizada para auxiliar na busca pela melhor solução.

3. Avaliação Experimental

Os experimentos foram executados sobre a versão 1.1.0 do software *Apache Ant* [Apache Software Foundation 2000]. Para que fosse possível avaliar a abordagem quanto ao estilo arquitetural, criamos um modelo com o estilo arquitetural em camadas, adotando a seguinte abordagem: cada pacote da raiz principal do projeto é uma camada; subpacotes pertencem à 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. A quantidade de componentes, classes, camadas e o valor de MQ da versão utilizada estão apresentados na Tabela 1.

Tabela 1. Medidas do Apache-Ant considerando estilo arquitetural em camadas.

#Classes	#Componentes	#Camadas	MQ
97	7	6	0.55534

Para cada teste feito, foi aplicado várias combinações dos parâmetros de configuração para verificar qual obtinha melhor resultado do valor MQ. Além disso, cada configuração foi executada 10 vezes devido à característica probabilística do algoritmo. Os resultados obtidos foram comparados com o modelo original da arquitetura, comparando-se resultados de MQ da solução indicada pelo ACO com o valor MQ da solução original.

A partir da análise dos resultados obtidos pelo ACO, é possível averiguar que, partindo da métrica MQ, o algoritmo consegue encontrar soluções melhores do que as iniciais, conforme apresentado na coluna MQ da Tabela 2, sem considerar a avaliação de estilos arquiteturais. A coluna *Id* representa um identificador para a configuração, para que possa ser referenciada mais adiante no texto. É possível verificar que mesmo a pior solução encontrada pelo algoritmo (Id 7), com MQ de 0,5629, é melhor que a solução de entrada, que possui valor MQ de 0,5553, ou seja a melhor solução obteve um valor 11% melhor. 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 bons resultados.

Tabela 2. Resultados para Apache Ant 1.1.0 sem avaliar o estilo arquitetural.

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 reporta os resultados obtidos a partir de uma arquitetura de entrada com estilo arquitetural em camadas da versão 1.1.0 do software Apache-Ant. A ordenação dos dados da Tabela se dão de forma decrescente considerando a coluna MQ. Podemos analisar pela coluna Penalizações que as soluções com maiores valores de MQ e iterações, infringiram menos as regras do estilo arquitetural 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 são piores e o número de relacionamentos que violam a regra do estilo é mais alto em relação às melhores soluções. 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 mais baixos.

A Tabela 3 mostra o tempo de execução do ACO em uma máquina com sistema operacional Fedora, processador Intel Core i3 2.20GHz e memória de 4 Gb. As config-

Tabela 3. 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

Tabela 4. Resultados para Apache Ant 1.1.0 considerando o estilo arquitetural.

Id	Iterações	Formigas	ρ	α	β	MQ	Penalizações
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

urações mostradas são referentes as que obtiveram melhor e pior resultados de MQ. A coluna *Estilo* significa se há estilo arquitetural. As colunas *Iterações*, *Formigas*, ρ, α e β mostram os valores dos parâmetros e a coluna *Tempo de Execução* marca o tempo que o ACO demorou para executar a configuração.

A Figura 1 apresenta a taxa de convergência dos valores MQ obtidos pelo algoritmo considerando estilo arquitetural em camadas. Observa-se a evolução do MQ para três configurações definidas na Tabela 4 e distintas com exceção da quantidade de formigas (sempre 20): a linha azul representa a configuração Id 1 ($\rho = 0,7$, $\alpha = 0,4$ e $\beta = 0,0$) e que obteve $MQ = 0,60244$; a linha vermelha representa a configuração Id 3 ($\rho = 0,7$, $\alpha = 0,4$, $\beta = 0,7$) e com $MQ = 0,59648$; e a linha alaranjada representa a configuração Id 4 ($\rho = 0,7$, $\alpha = 0,4$, $\beta = 0,3$) com $MQ = 0,59298$. Observa-se que as configurações, após convergirem para a melhor solução, ficaram algumas iterações produzindo os mesmos valores MQs 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 um bom resultado.

A Figura 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 que possui $\beta = 0,0$, iniciou com soluções que infringiam muito as regras do estilo, enquanto as configurações com $\beta = 0,7$ (linha vermelha) e $0,3$ (linha laranjada) produziram soluções que infringiam menos as regras 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 que os valores de β, acabam sendo invalidados pela função MQ, isso pelo fato de que a regra do estilo arquitetural em camadas penaliza relacionamentos entre

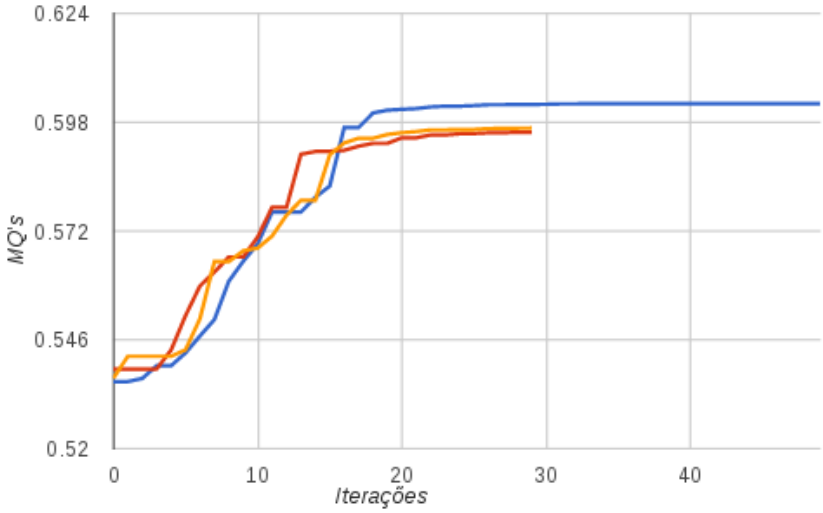

Figura 1. Evolução do valor de MQ para Apache Ant 1.1.0 com estilo em camadas.

classes em que: 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 à 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 no resultado final das penalizações para o estilo arquitetural tratado neste artigo, pois a função objetivo já reduz os relacionamentos externos o que automaticamente fará com que não haja penalidades entre os relacionamentos.

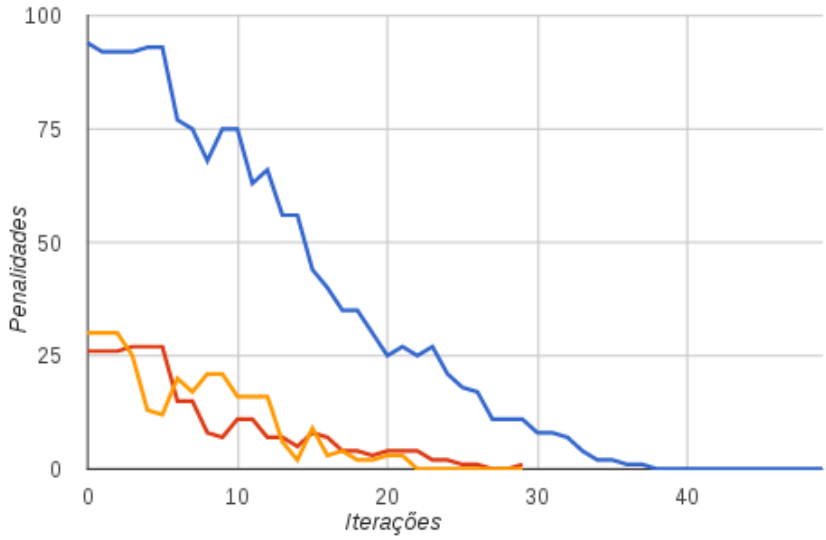

Figura 2. Evolução das penalidades das soluções geradas ao longo das iterações.

4. Trabalhos Relacionados

Dois trabalhos tratam da otimização do design de arquiteturas considerando ACO e da avaliação da abordagem proposta. O primeiro utiliza ACO para propor um design

otimizado de diagrama de classes de um software de acordo com métricas de coesão e acoplamento (CK) [Tawosi et al. 2015]. A representação da arquitetura é um grafo direcionado acíclico, cujos nós são organizados em duas dimensões: responsabilidades e classes. Dessa maneira, a seleção de um determinado nó representa a possível atribuição de uma responsabilidade a uma classe. O algoritmo inicia com o preprocessamento da arquitetura de entrada e, em seguida inicia, a busca, gerando designs candidatos para encontrar a solução mais próxima da ótima possível. As formigas procuram caminhos no grafo acíclico para gerar suas soluções e depositam feromônio nestes caminhos percorridos de acordo com o valor de uma função multi-objetivos [Tawosi et al. 2015]. Os autores propõem ainda uma métrica chamada *Meaningfulness Metric* (MM), utilizada para medir quanto o diagrama de classes é compreensível para um humano. Em comparação ao nosso trabalho, a escolha de métricas possui objetivos similares. No entanto, ao invés de utilizar diversas métricas e uma função multiobjetivo, optamos por uma medida que trata coesão e acoplamento (MQ). Embora não tratemos na compreensibilidade do modelo gerado diretamente, a utilização de penalidades em nossa abordagem busca preservar importantes decisões sobre a arquitetura relacionadas à facilidade de compreensão lógica.

O segundo trabalho compara três métodos para otimização de design de software: otimização por colônia de formigas, algoritmo genético e busca gulosa [Simons e Smith 2013]. O modelo de representação de um design arquitetural (considerando classes, métodos e atributos) é um conjunto com uma sequência de inteiros, em que cada valor do conjunto representa um elemento do design do software. Na implementação apresentada, 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. As métricas consideradas 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. Em nosso trabalho, não consideramos os atributos de uma classe, o que não permite comparar sob a perspectiva de elegância, mas aspectos de acoplamento são considerados.

5. Conclusões

Neste trabalho foi explorado a utilização da metaheurística de 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, melhorando assim a manutenibilidade do software. A métrica proposta neste trabalho busca verificar o estilo arquitetural do sistema, aplicando penalidades às soluções que quebram regras do estilo arquitetural imposto.

Os experimentos foram conduzidos para que fosse possível averiguar se o ACO era apto a obter resultados satisfatórios em relação a arquitetura inicial do sistema. Para a execução dos experimentos foi aplicado a abordagem proposta sobre uma versão do software Apache Ant. Os resultados obtidos mostraram que o ACO tem capacidade para produzir resultados satisfatórios considerando as métricas usadas, obtendo para o primeiro

experimento resultados cerca de 11% maior que da arquitetura inicial e para o segundo experimento obtendo MQ 7% maior que a arquitetura inicial. Dessa maneira, os resultados obtidos pelo algoritmo mostraram melhores valores MQ que a arquitetura inicial. Portanto o ACO se mostra uma opção interessante para ser utilizada e explorada para problemas de otimização arquitetural.

Dado o desenvolvimento deste trabalho podemos apontar alguns trabalhos futuros necessários para melhorar a utilização do ACO para otimização arquitetural. Experimentos com outras metaheurísticas, a fim de comparar os resultados de diferentes abordagens. Experimentos considerando outros estilos arquiteturais, para analisar como se comporta a métrica de penalidades proposta. Necessidade do uso de técnicas para redução do tempo de execução do algoritmo. Utilização de outras métricas de qualidade para verificar a capacidade do ACO de lidar com funções multiobjetivas.

Referências

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., e Meedeniya, I. (2013). Software architecture optimization methods: A systematic literature review. *IEEE Trans. Softw. Eng.*, 39(5):658–683.

Apache Software Foundation (2000). Ant. Programa de Computador.

Garlan, D. (2000). Software architecture: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, p. 91–101, New York, NY, USA. ACM.

Garlan, D. (2014). Software architecture: A travelogue. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, p. 29–39, New York, NY, USA. ACM.

Garlan, D. e Shaw, M. (1993). An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3.4).

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

Mancoridis, S., Mitchell, B. S., Chen, Y., e Gansner, E. R. (1999). Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, p. 50–59.

Mariani, T., Colanzi, T. E., e Vergilio, S. R. (2016). Preserving architectural styles in the search based design of software product line architectures. *Journal of Systems and Software*, 115:157 – 173.

Rumbaugh, J., Jacobson, I., e Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, Boston, MA, USA.

Saed, A. e Kadir, W. (2011). Applying particle swarm optimization to software performance prediction an introduction to the approach. In *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, p. 207–212.

SEI (2015). Defining software architecture.

Simons, C. e Smith, J. (2013). A comparison of meta-heuristic search for interactive software design. *Soft Computing*, 17(11):2147–2162.

Tawosi, V., Jalili, S., e Hasheminejad, S. M. H. (2015). Automated software design using ant colony optimization with semantic network support. *Journal of Systems and Software*, 109:1 – 17.