Universidade Federal da Bahia

Instituto de Matemática e Estatística

Departamento de Ciência da Computação

Dennis Lessa Dourado

Recuperação da arquitetura de software usando algoritmos de agrupamento

Salvador, Bahia

2016

Universidade Federal da Bahia

Instituto de Matemática e Estatística

Departamento de Ciência da Computação

Recuperação da arquitetura de software usando algoritmos de agrupamento

Discente: Dennis Lessa Dourado

Orientador: Prof. Dr. Rodrigo Rocha Gomes e Souza

Monografia orientada pelo Prof. Dr. Rodrigo Rocha Gomes e Souza e apresentada a Universidade Federal da Bahia, como parte dos requisitos necessários para a conclusão do curso de Ciência da Computação, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Salvador, Bahia

2016

Resumo

Sistemas de softwares evoluem constante e rapidamente. Atividades de manutenção são exigidas durante a sua existência útil e, dessa forma, aumentam sua complexidade. Contudo, essas atividades de manutenção são realizadas, muitas vezes, sem o conhecimento profundo da arquitetura do software, até porque esta não existe ou não está atualizada. Por isso, as técnicas de recuperação de software são utilizadas para propor uma representação da arquitetura do software e assim facilitar as tarefas de manutenção durante a evolução do software.

O objetivo desse trabalho foi comparar três algoritmos de agrupamento utilizados na recuperação da arquitetura de software: ACDC, Bunch e LIMBO. Para isso, foram escolhidos três sistemas de software: Bash, InGE e SIPOS. Como métrica foi utilizada a distância MoJo e os algoritmos foram comparados segundo o critério de autoridade e tempo de execução. Cada algoritmo foi executado 10 vezes em cada sistema escolhido.

Os resultados mostraram que o algoritmo ACDC apresentou os melhores resultados para a distância MoJo em todos os testes realizados, além de tempos de execução baixos.

Palavras-chave: algoritmos de agrupamento, arquitetura de software, recuperação de arquitetura de software

Abstract

Systems software evolve constantly and rapidly. Maintenance activities are required during their lifetime and thereby increase their complexity. However, these maintenance activities often carried out without a deep knowledge of software architecture, either because it is not documented or because the documentation it not up-to-date. As a result, software recovery techniques are used to propose a representation of the software architecture, thus facilitating maintenance tasks during the evolution of software.

The aim of this study was to compare three clustering algorithms used in the recovery of software architecture: ACDC, Bunch and LIMBO. For this, purpose we chose three software systems: Bash, InGE and SIPOS, using the MoJo distance to compare the recovered and the authoritative architecture, and runtime performance algorithms were compared according to the criterion of authoritativeness. Each algorithm we executed 10 times for each system.

The results showed that the ACDC algorithm has the best results for the distance MoJo in all tests, int addition to presenting acceptable runtime performance.

Keywords: clustering algorithms, software architecture, software architecture recovery

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Os softwares evoluem com o passar do tempo em consequência de mudanças de tecnologias ou dos seus requisitos [Lung and Zaman, 2004]. Essas alterações, que podem ser correções de falhas, adição, remoção ou alteração de funcionalidades [Silva and Bittencourt, 2013, Shtern and Tzerpos, 2012, Garcia et al., 2013], são medidas tomadas durante a evolução do software para mantê-lo em funcionamento. Muitas dessas atividades de manutenção ou implementação de novas funcionalidades são realizadas sem o conhecimento profundo da arquitetura do software e, por menor que seja, a alteração pode comprometer o funcionamento dos componentes do sistema [Lung and Zaman, 2004, Wiggerts, 1997, Mancoridis et al., 1998, Tzerpos and Holt, 2000b, Silva and Bittencourt, 2013]. Essas atividades de manutenção contínua que o sistema sofre durante sua evolução afetam diretamente o seu tamanho e complexidade [Lung and Zaman, 2004, Andritsos and Tzerpos, 2005], podendo torná-lo cada vez menos flexível e mais difícil de manter [Wiggerts, 1997, Maqbool and Babri, 2004]. Mesmo assim, reescrever todo o sistema ou substituí-lo não é viável [Maqbool and Babri, 2004], pois o software adquiriu complexidade ao longo do tempo, incluindo suas regras de negócios.

A Arquitetura de Software é um campo da Engenharia de Software que descreve a estrutura bruta do software, sendo fundamental para o desenvolvimento de sistemas, uma vez que, segundo Garlan [Garlan, 2000], um bom projeto de arquitetura auxilia no cumprimento de requisitos não-funcionais, como desempenho, confiabilidade, portabilidade, escalabilidade e interoperabilidade. Em consequência disso, um bom planejamento da arquitetura permite que durante a evolução do sistema a

manutenção e gerência do mesmo não sejam tarefas árduas.

Porém, muitos sistemas de software não possuem a sua arquitetura documentada [Bowman et al., 1999, Tzerpos and Holt, 2000b], como softwares legados, ou se a possui, muitas vezes, não se encontra sincronizada ou atualizada com a versão do software. Portanto, faz-se necessário recuperar a arquitetura do software a partir do código-fonte [Garcia et al., 2013, Maqbool and Babri, 2007, Beyer and Noack, 2005], utilizando diversas técnicas de recuperação de arquitetura, dentre as quais as chamadas técnicas de agrupamento, as quais serão abordadas neste trabalho.

As técnicas de agrupamento são bastante utilizadas [Lung and Zaman, 2004] para agrupar elementos semelhantes segundo algum critério, baseado em uma função de dissimilaridade [Linden, 2009], de forma que elementos em um mesmo grupo são semelhantes entre si, ou seja, possuem uma dissimilaridade menor, enquanto a dissimilaridade entre grupos é maior.

Em Engenharia de Software, as técnicas de agrupamento para a recuperação da arquitetura têm como um dos objetivos, através da decomposição do sistema de software em vários subsistemas [Lung and Zaman, 2004, Shtern and Tzerpos, 2012], identificar componentes que possam ser agrupados em um mesmo módulo, ou seja, com alta similaridade entre si, aumentando a coesão. Do mesmo modo identifica componentes que possuem baixa similaridade entre si, baixo acoplamento, sendo agrupados em módulos diferentes [Bittencourt and Guerrero, 2009].

Portanto, o objetivo deste estudo é avaliar os agrupamentos obtidos pelos algoritmos de agrupamento comumente utilizados para recuperação de arquitetura de software, segundo critérios de autoridade. Isto é, verificar o quanto que os agrupamentos resultantes se aproximam dos agrupamentos de referência, ou seja, aqueles feitos por um especialista com base na arquitetura do software.

Este trabalho está estruturado da seguinte maneira: no Capítulo 2 serão apresentados os conceitos gerais de agrupamento, conceitos sobre a Arquitetura de Software e as definições dos algoritmos utilizados neste trabalho. No Capítulo 3 será apresentada a metodologia para a realização deste estudo, quais os sistemas escolhidos e os resultados dos estudos. No Capítulo 4, serão apresentadas análises a partir dos resultados obtidos. E por último serão apresentadas as conclusões e trabalhos futuros no Capítulo 5.

Capítulo 2

Referencial teórico

2.1 Recuperação de arquitetura

A arquitetura de software é uma definição de alto nível que, segundo [Garlan, 2000], descreve a estrutura do software em que são apresentadaos os relacionamentos entre seus componentes e suas propriedades. Adicionalmente, a arquitetura de software desempenha um papel fundamental para o desenvolvimento e manutenção do software, servindo de ligação entre os requisitos do sistema e sua implementação [Garlan, 2000] como mostra a Figura 2.7.

Ainda, segundo [Garlan, 2000], a arquitetura de software desempenha um papel fundamental em seis áreas do desenvolvimento de software como: entendimento, reuso, construção, evolução, análise e gestão. Daí podemos concluir a importância de se possuir uma documentação arquitetural e mantê-la sempre atualizada com a versão do software.

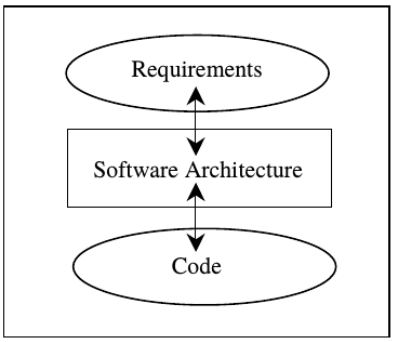


Figura 2.1: Arquitetura de software.

[Tzerpos and Holt, 2000b, Bowman et al., 1999] afirmam que muitos sistemas de softwares legados não a possuem ou a mesma não está sincronizada com a versão do software e, com isso, o uso de ferramentas para a recuperação da arquitetura faz-se necessário. A recuperação de arquitetura de software (SAR - Software Architecture Recovery) pode ser definida como sendo o processo de descrição de alto nível [Gall et al., 1996] que procura reconstruir visões arquiteturais de um sistema de software [Ducasse and Pollet, 2009], podendo ser realizado através do código-fonte ou pela nomenclatura dos arquivos que contêm os códigos.

[Ducasse and Pollet, 2009] ainda propõem uma taxonomia baseada no tempo de vida da reconstrução da arquitetura de software dividida em 5 aspectos: objetivo, processos, inputs, técnicas, e outputs, mostrado pela Figura 2.2.

O objetivo da SAR, assim como, já mencionado, e defendido por [Garlan, 2000], contribui para seis grandes metas do desenvolvimento de Software.

Os processos podem ser divididos em três abordagens: bottom-up, top-down e híbrido. A primeira abordagem inicia-se a partir do código-fonte ou baixo nível até alcançar o entendimento de alto nível da aplicação. A segunda abordagem inicia-se a partir de regras de negócio ou estilos de arquitetura. A última abordagem utiliza o melhor das abordagens anteriores, de forma que o conhecimento a partir da abordagem bottom-up ajuda no refino dos resultados da abordagem top-down.

Os inputs são os dados iniciais utilizados pela SAR os quais são divididos em dois grupos: não-arquiteturais e arquiteturais. O primeirao grupo, que inclui este trabalho, refere-se ao: código-fonte, informação textual (comentários, nome para métodos, entre outros), informação dinâmica (informações adquiridas em tempo de execução), organização física do sistema (arquivos e pastas), organizações humanas, informações históricas (informações sobre a co-evolução entre o código e o design) e informações de um especialista (usado para validar os resultados). O segundo grupo refere-se a inputs arquiteturais como: estilos (sistemas de camadas, fluxo de dados, entre outros) e os pontos de vista.

A próxima etapa são as técnicas usadas em SAR que podem ser divididas em 3 categorias: quase-manual – quando os elementos são identificados manualmente, semi-automático – instruções são passadas manualmente mas uma ferramenta automaticamente identifica elementos e, por fim, quase-automático, quando a ferramenta

tem o controle mas não conduz o processo de recuperação iterativa.

Por fim, os outputs, que são as formas como a SAR apresenta os resultados podendo ser por meio de: visualização de software, arquitetura, análise e conformidade - que pode ser horizontal, quando se compara os agrupamentos gerados por algoritmos de agrupamento com o agrupamento de referência, e a conformidade vertical quando se verifica se a arquitetura está de acordo com o código fonte [Ducasse and Pollet, 2009].

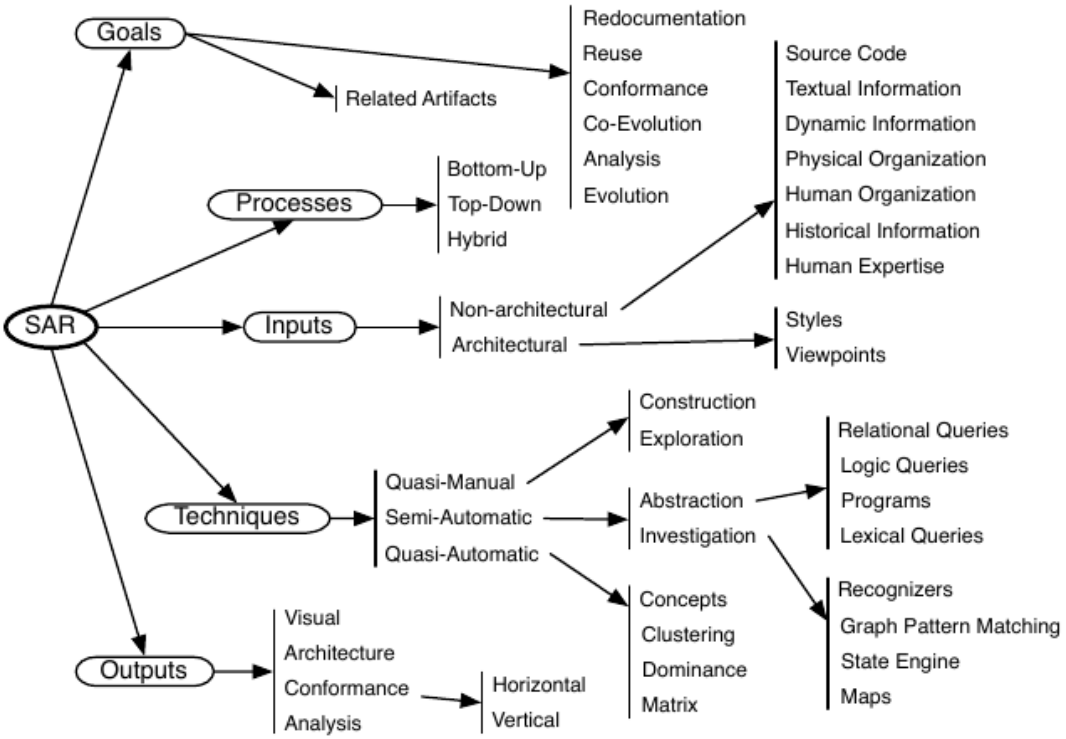


Figura 2.2: Aspectos da SAR.

Este trabalho utiliza técnicas e algoritmos de agrupamento para a recuperação de arquitetura de software que se enquadram como uma técnica quase automática. E ainda, utilizam uma abordagem bottom-up, tendo como input elementos não arquiteturais, pois será analisado apenas o código-fonte. O output gerado é de conformidade horizontal, uma vez que será comparado os agrupamentos gerados pelos algoritmos com a arquitetura de referência.

2.2 Agrupamento: uma abordagem geral

As técnicas de agrupamento (ou *clustering*) são muito utilizadas em diversas áreas [Lung and Zaman, 2004, Wiggerts, 1997]. Essa técnica consiste em juntar em um mesmo grupo entidades que são semelhantes entre si, ao passo que são diferentes das entidades que se encontram em outros grupos [Tzerpos and Holt, 2000a, Linden, 2009, Maqbool and Babri, 2007, Naseem et al., 2011], segundo regras de similaridades ou dissimilaridades, ver Figura 2.3.

Na Engenharia de Software, as técnicas de agrupamento geram *clusters* em que a coesão, que corresponde ao relacionamento de elementos em um mesmo grupo, seja maximizada, e o acoplamento, que representa as dependências entre grupos, seja minimizado [Lung and Zaman, 2004, Bittencourt and Guerrero, 2009, Lung, 1998]. Desse modo, o sistema é decomposto em vários subsistemas, o que facilita a sua compreensão [Tzerpos and Holt, 2000a, Naseem et al., 2011].

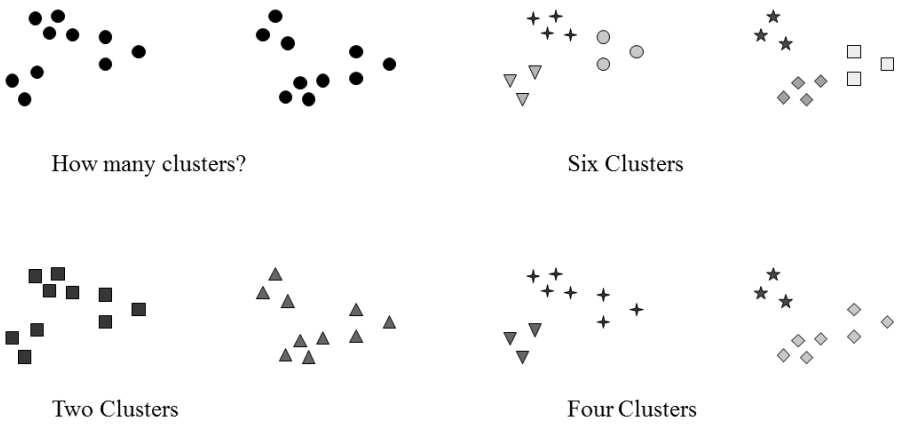


Figura 2.3: Exemplo de agrupamentos. Figura retirada do Google.

Há mais de vinte anos que trabalhos em agrupamento de software são desenvolvidos, segundo [Shtern and Tzerpos, 2012], e inúmeras técnicas de agrupamento têm sido propostas para a recuperação de arquitetura de software [SOUZA, 2010, Lung and Zaman, 2004, Ducasse and Pollet, 2009]. Essas técnicas são fundamentais para a recuperação da arquitetura de um sistema de software, uma vez que facilitam a representação do modelo arquitetural do software [Vasconcelos, 2004], a manutenção e evolução [Lung, 1998] do sistema.

[Oliveira and Andrade, 2014] afirmam que entre os algoritmos de agrupamento

existentes os mais utilizados atualmente são: ACDC, Bunch, LIMBO, WCA, ZBR e ARC.

2.3 Extração de dependências e DSM

Nesta seção são abordadas etapas fundamentais para o processo de recuperação de software, que são a extração de dependências e como representá-las. Contudo, as etapas não serão detalhadas uma vez que não são objetos de estudo deste trabalho.

2.3.1 Extração de dependências

Uma dependência é uma referência de uma entidade a outra, isto é, uma entidade depende de alguma ação ou atributo de outra entidade. Essas referências podem ser: chamadas de funções, procedimentos ou métodos, uso de variáveis ou atributos ou herança.

Essas dependências são representadas por meio de grafos orientados, como mostra a Figura 2.4 retirada do trabalho [Mancoridis et al., 1998]. Os cículos representam as entidades do sistema - uma entidade pode ser uma classe, atributos, métodos ou arquivos [Silva and Bittencourt, 2013]. As arestas direcionadas representam a delação de dependência da entidade. A entidade de partida da aresta indica o dependente e a entidade de chegada da aresta indica a dependência.

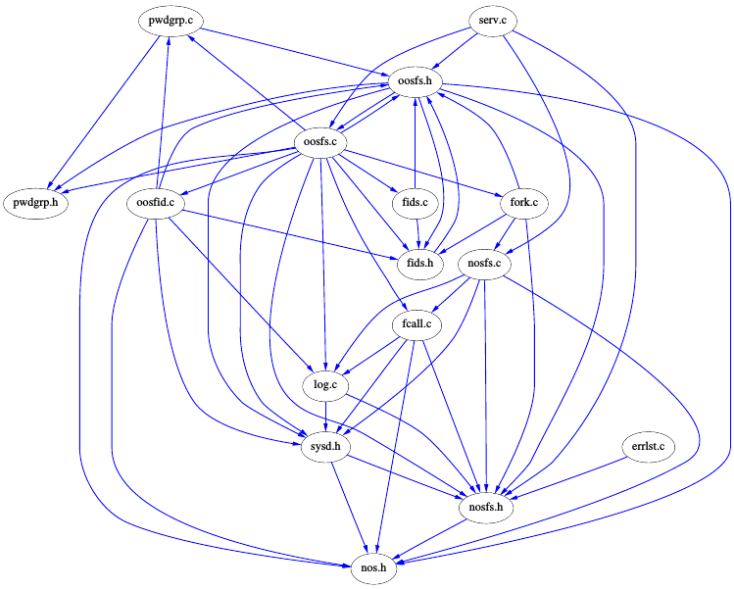


Figura 2.4: Representação de dependências usando grafos direcionados.

2.3.2 Design Structure Matrix - DSM

Para [Browning, 2001] uma DSM ou Matriz de Estrutura de Design dispõe uma visualização simples e compacta de um sistema complexo. Devido a sua simplicidade em apresentar o relacionamento entre os componentes do sistema têm se tornado populares. Usando as DSMs é possível representar grafos orientados.

Uma DSM é uma matriz quadrada em que as linhas e colunas e a diagonal principal possuem os mesmos rótulos e representam, cada uma, um elemento do sistema. As marcações fora da diagonal principal representam as dependências entre os elementos (ver Figura 2.5).

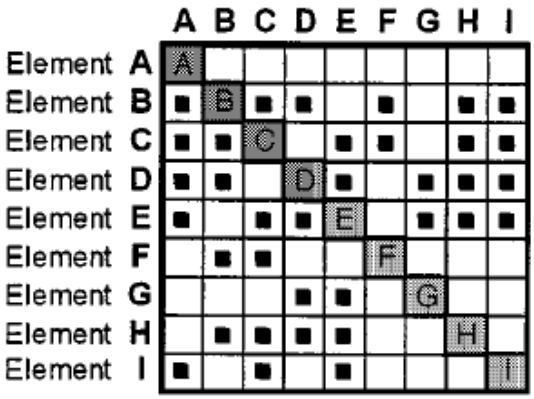


Figura 2.5: Exemplo de uma DSM.

Esse trabalho não irá utilizar DSM para análise de algoritmos ou sistemas. Sua utilização será apenas na fase de extração de dependências em que uma DSM é gerada e a partir dela serão gerados os inputs para cada algoritmo.

2.4 Algoritmos de agrupamento

Esta seção apresentará os algoritmos a serem utilizados neste trabalho. Como apresentado na seção anterior, vários são os algoritmos utilizados para a recuperação de arquitetura porém, neste trabalho somente 3 dos mais utilizados serão comparados. Portanto a seguir serão descritos os algoritmos ACDC, Bunch e LIMBO. Esses algoritmos foram escolhidos devido a facilidade de encontrá-los implementados.

2.4.1 ACDC

O algoritmo ACDC - *Algorithm for Comprehension-Driven Clustering*, proposto por [Tzerpos and Holt, 2000a], é um dos algoritmos mais utilizados para recuperação de arquitetura de software. Segundo eles, existem vários padrões de subsistemas dentre os quais o algoritmo ACDC se baseia para a realização dos agrupamento, são eles: padrão de arquivo fonte, padrão corpo-cabeçalho, padrão coleção de folhas e padrão do subgrafo dominante. Baseado nesses padrões o algoritmo ACDC realiza o agrupamento em duas fases.

Na primeira fase do algoritmo ACDC são realizados 5 passos, em ordem de precedência. O primeiro passo considera o arquivo de origem, contendo as dependências do sistema, como sendo uma entidade atômica para as próximas etapas, de maneira que, o nome do arquivo de origem será o nome do cluster. Os nomes dos subsistemas gerados possuirão o sufixo ".ss". No segundo passo, o conglomerado interface e implementação são agrupados em um mesmo cluster e o subsistema possuirá o nome comum aos dois arquivos ".ss".

No terceiro passo, o ACDC identifica arquivos que podem ser possíveis candidatos para o subsistema pela coleta de folhas e bibliotecas de suporte. No quarto passo, principal etapa do algoritmo, usando o padrão do despachante central, são desconsiderados os arquivos que possuem um grau de saída superior a 20, isso porque o ACDC tenta produzir *clusters* de cardinalidade entre 5 e 20. Os graus de saída indicam a quantidade de arestas que partem uma entidade do software (dependência a outros elementos); graus de entrada representam a quantidade de arestas que chegam a uma entidade do software (serve a outros elementos); a cardinalidade é a soma dos graus de entrada e saída em uma entidade do software.

Por fim, o último passo, o quinto passo, um grupo "support.ss" é criado e todos os arquivos identificados como bibliotecas de suporte são atribuídos a esse subsistema, exceto se o arquivo tenha sido atribuído a outro subsistema no passo 4.

A segunda fase do algoritmo usa a técnica de adoção de órfãos, realizando um agrupamento incremental em que os arquivos que não foram agrupados são os órfãos. Essa técnica é utilizada com a finalidade de que todos os arquivos sejam agrupados sendo colocados no *cluster* que possui maior conectividade com o órfão.

2.4.2 Bunch

A ferramenta Bunch, proposta por [Mitchell, 2002], implementa 3 algoritmos: Hill Climbing, Algoritmo Genético e Algoritmo de Busca Exaustiva. Esses algoritmos utilizam um MDG (*Module Dependency Graph*) como input, a qual, é uma linguagem independente utilizada para a representação das dependências entre as entidades do software [Mitchell, 2002]. Os algoritmos Bunch são avaliados por uma medida MQ (*Modularization Quality*) que é a função objetivo que se busca maximizar.

Segundo [Mitchell, 2002], a MQ é uma avaliação quantitativa que determina a qualidade de uma partição do MDG entre a intra-conectividade e inter-conectividade. Para medir a MQ, são definidas três funções objetivos: BasicMQ, TurboMQ e Incremental TurboMQ. A primeira função foi criada para produzir valores maximizados *intra-cluster* e valores minimizados *inter-cluster*. Porém, essa função objetivo tem duas limitações: uma diz respeito ao desempenho, limitado a sistemas pequenos (menos de 75 módulos, segundo Tzerpos e Holt [Tzerpos and Holt, 1999]). A outra limitação se refere ao não suporte à arestas com peso.

A segunda função objetivo foi desenvolvida para superar as limitações da função BasicMQ, inclusive para MDG com pesos nas arestas. Cada *cluster* tem um Fator de Cluster (FC) que é obtido pela divisão normalizada das suas arestas internas pela metade das suas arestas externas. E a soma de todos os FCs de cada *cluster* representa o valor da função TurboMQ. A última função objetivo, também chamada de ITurboMQ, expressa que quando realizada uma operação de mover um arquivo de um *cluster* para outro, apenas os *clusters* de origem e de destino terão seu FC alterados, portanto, é possível incrementalmente alterar o valor de TurboMQ apenas baseado nos fatores alterados.

Agora, depois de definidos conceitos essenciais para o entendimento dos algoritmos é o momento de descrever aqueles que serão objeto de estudo deste trabalho.

Algoritmo Hill Climbing

O algoritmo Hill Climbing da ferramenta Bunch é iniciado com uma partição aleatória do MDG. Outras partições melhores deverão ser encontradas e caso sejam encontradas, o processo é iterado de forma que a nova partição é usada como base para encontrar outras partições melhores. Ou seja, o algoritmo cria uma popula-

ção com as partições aleatórias do MDG e seleciona aquela que possui uma maior MQ. Uma partição para o algoritmo hill climbing é uma sugestão de agrupamento realizado e a partir dela seleciona-se aquela que possui maior MQ.

Esse algoritmo usa o conceito de partições vizinhas em que módulos são movidos entre os *clusters* de uma partição com o objetivo de melhorar a MQ. Uma partição vizinha é igual a partição original exceto por um único elemento que diferencia a partição original da partição vizinha - ver Figura 2.6.

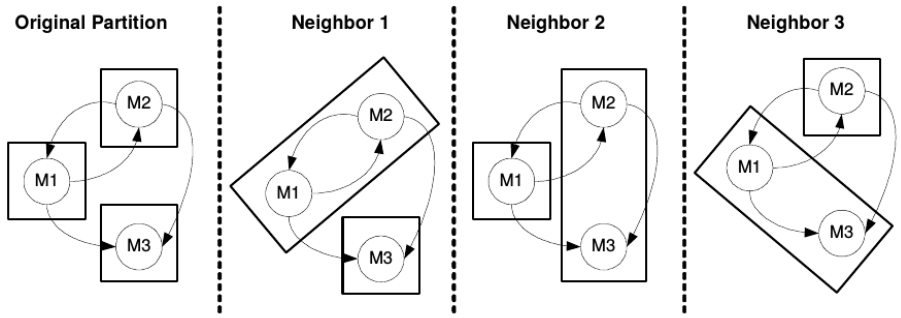


Figura 2.6: Exemplo de uma DSM.

Algoritmo Genético

Os algoritmos genéticos são aplicados para resolverem problemas em grandes espaços de busca utilizando o conceito de seleção natural [Mitchell, 2002]. Esses algoritmos atuam em uma população de strings que representam codificações do problema a ser resolvido. Uma função objetivo é utilizada para calcular o qualidade do valor da string e regras probabilísticas são utilizadas para direcionar a busca. Para [Mitchell, 2002], um algoritmo genético é composto por função objetivo, tamanho da população e operadores genéticos.

A função objetivo é utilizada para identificar os indivíduos ou agrupamento mais ou menos aptos atribuindo-lhes um valor de aptidão (*fitness value*), que representa o MQ, que por meio desse será realizada a seleção natural através dos operadores genéticos. O tamanho da população representa o número de strings (ou indivíduos). Os operadores genéticos são quatro: seleção, reprodução, *crossover* e mutação. Durante a fase de seleção e reprodução os indivíduos são escolhidos aos pares, de acordo com o seu valor de aptidão. Essa escolha pode acontecer de diferentes maneiras, a es-

colhida por [Mitchell, 2002], foi a roulette wheel, em que indivíduos são selecionados aleatoriamente proporcionalmente a seu valor de aptidão.

A seleção usa o método elitismo, em que o melhor indivíduo é mantido para as próximas gerações. O segundo operador, o *crossover* é utilizado para criar novos indivíduos a partir de dois indivíduos pais [Sivanandam and Deepa, 2007]. Por último, a mutação que, após o *crossover*, modifica o valor genético da população alterando o valor da cada gene sob uma probabilidade.

A ferramenta Bunch utiliza as seguintes configurações para seu algoritmo genético (para N igual ao número de nós no MDG):

Taxa de crossover : 80% para populações de 100 indivíduos ou menos ou 100% para populações com milhares de indivíduos ou mais;

Taxa de mutação : 0,004 log2(N);

Tamanho da população : 10N;

Número de gerações : 200N;

2.4.3 LIMBO

O algoritmo LIMBO foi proposto por [Andritsos et al., 2003] para trabalhar com grande conjunto de dados por meio de resumos de distribuição. Esse algoritmo trabalha com as descrições das tuplas ou *cluster* não sendo necessário armazená-los completamente na memória principal [Andritsos et al., 2003].

Esse algoritmo utiliza como input uma matriz normalizada de probabilidades, em que, a soma dos valores de cada linha deve ser igual a 1. As linhas representam os elementos (tuplas) e as colunas representam os atributos. Para o agrupamento de software, tanto as linhas quanto as colunas representam os elementos do software a serem agrupados.

O algoritmo LIMBO executa a tarefa de agrupamento em 4 fases: criação dos artefatos resumidos (DCFs), aplicação do algoritmo AIB, associação dos artefatos originais com o *cluster* e determinando o número de *cluster*. É definido artefato original como sendo um elemento a ser agrupado e artefatos resumidos como sendo a DCF de cada um dos artefatos originais.

Uma DCF (*Distributional Cluster Feature*) ou Recursos de Cluster Distributivo é o recurso utilizado pelo LIMBO, em que se resume um agrupamento de tuplas, para calcular a distância entre *cluster*s ou entre *clusters* e tuplas.

Na primeira fase os artefatos são lidos um a um e gerados suas DCFs correspondentes. Na segunda fase é executado o algoritmo AIB para produzir uma série de agrupamentos, de baixa carinalidade, dos artefatos resumidos. Na terceira fase o conjunto de agrupamentos gerados na fase anterior são convertidos em conjunto de agrupamentos dos artefatos originais. Na quarta e última fase um desses agrupamentos é escolhidos.

Na segunda fase do LIMBO é usado o algoritmo *Agglomerative Information Bottleneck* (AIB) [Slonim and Tishby, 1999] que propõe uma solução para a otimização do problema de busca do agrupamento com a menor cardinalidade e menor perda de informação. Tem como input o conjunto de DCFs gerado na primeira fase do LIMBO e como saída um conjunto de agrupamentos dos artefatos resumidos.

2.4.4 Avaliação dos algoritmos de agrupamento

[Wu et al., 2005] propuseram 3 critérios para avaliação dos agrupamentos arquitetural: estabilidade, autoridade e extremidade. A primeira métrica, estabilidade, diz que para versões similares de um software, agrupamentos similares devem ser gerados, isto é, a estrutura do agrupamento deve ser mantida para versões sucessivas do software. A segunda métrica proposta, é a autoridade, o que significa que o agrupamento gerado por um algoritmo de agrupamento deve ser semelhante ao agrupamento proposto por um especialista (ou agrupamento de referência).

A terceira métrica, extremidade, diz que o tamanho dos agrupamentos não podem ser extremos, isto é, não pode haver poucos agrupamentos com a maioria dos elementos e nem muitos agrupamentos com um único elemento. Neste trabalho, utilizaremos apenas a autoridade como critério de avaliação, visto que pelas definições de estabilidade e extremidade não se enquadram no objetivo desse trabalho.

Muitos métodos têm sido apresentados por vários autores [Wen, 2003], como métricas para cálculo da distância para algoritmos hierárquicos [Lakhotia and Gravley, 1995], similaridade baseado em precisão e recall [Anquetil and Lethbridge, 1999], ainda podem ser citados os coeficientes de associação, de distância, de correlação e proba-

bilísticos. [Tzerpos and Holt, 1999] propuseram uma distância chamada MoJo que calcula a diferença entre dois agrupamentos que será descrita em seguida.

MoJo

Proposto por [Tzerpos and Holt, 1999], MoJo é uma métrica que mede o quanto distante é um agrupamento de outro. Assim, baseado no critério de autoridade discutido na seção anterior, essa métrica será utilizada para se atingir o objetivo deste estudo que é comparar o agrupamento gerado por um algoritmo de agrupamento com um agrupamento predefinido por um especialista.

Segundo eles, uma forma de comparar algoritmos de agrupamento é comparar os resultados obtidos por eles quando aplicados em grandes sistemas de software. Porém, isso pode não ser uma atividade fácil, visto que, grandes sistemas de software possuem grandes quantidades de recursos. Por isso, eles propuseram um método automático para comparar as saídas geradas por algoritmos de agrupamento calculando-se as distância entre grupos a qual representa a dissimilaridade entre os grupos gerados. Quanto maior for essa distância maior será a diferença entre os grupos.

Essa métrica permite apenas duas operações: move (mover) e join (juntar) com peso igual a 1. A primeira operação move um elemento de um *cluster* para outro. A segunda operação junta dois *clusters* em um. Com isso, podemos afirmar que a métrica MoJo representa a quantidade mínima de operações de mover e juntar necessárias para se transformar um agrupamento em outro.

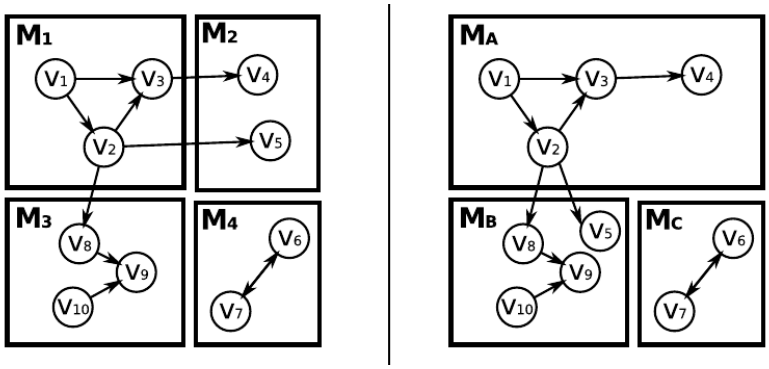


Figura 2.7: Transformando o agrupamento à esquerda no agrupamento à direita. Movendo V5 de M2 par M3 e, em seguida, juntando M2 a M1.

Essa métrica faz uso de um algoritmo heurístico que calcula um valor aproximado da distância MoJo de dois agrupamentos. Isso porque calcular a distância MoJo entre agrupamentos de sistemas softwares grandes pode requerer muito tempo.

Capítulo 3

Metodologia

Este trabalho tem como objetivo comparar algoritmos de recuperação de arquitetura usando a métrica MoJo utilizando o critério de autoridade. Para isso, foram escolhidos três sistemas de software Bash, SIPOS e InGE, aos quais, serão aplicados os algoritmos anteriormente abordados, cujas implementações foram feitas pelas ferramentas ACDC[1], Bunch[2] e Weka[3] (LIMBO). A ferramenta MoJo[1] para comparar os agrupamentos gerados pelos algoritmos com os agrupamentos gerados por um especialista.

Cada algoritmo foi executado 10 vezes em cada sistema, totalizando 30 execuções por algoritmo. Foram levantados a quantidade de *clusters* por algoritmo, o tempo de execução e a distância MoJo. Essas informações são relevantes porque a quantidade de *clusters* representam os módulos do sistema e seus elementos e serão comparados com o agrupamento de referência. O tempo é importante para informar qual das ferramentas executa em menos tempo a atividade de agrupamento. Mesclando as duas informações, no mesmo contexto, significaria dizer qual das ferramentas realiza o agrupamento mais próximo do agrupamento de referência em menos tempo. A distância MoJo de cada resultado mostra a distância entre cada agrupamento apresentado pelo algoritmo e o agrupamento de referência feito por um especialista.

[1] http://www.cse.yorku.ca/~bil/downloads/

[2] https://www.cs.drexel.edu/~spiros/bunch/

[3] https://sourceforge.net/projects/feed4weka/

3.1 Ferramental

Para a realização desse trabalho os algoritmos foram executados a partir de ferramentas que os implementam. Para a execução do algoritmo ACDC foi utilizada ferramenta que leva o mesmo nome do algoritmo e é implementada em Java, do próprio desenvolvedor. Para os algoritmos Hill Climbing e Algoritmo Genético foi utilizada a ferramenta Bunch implementada em Java, também, obtida a partir do seu desenvolvedor. Para executar o algoritmo LIMBO foi utilizada a ferramenta Weka.

A ferramenta Weka (*Waikato Environment for Knowledge Analysis*) é um software de código aberto desenvolvido pela Universidade de Waikato, Nova Zelândia. Essa ferramenta possui um conjunto de algoritmos para aprendizado de máquina. Com Weka é possível realizar tarefas de classificação, agrupamento, associação e regressão.

Para a fase de extração de dependências foi utilizada a ferramenta Analizo [Terceiro et al., 2010], um analisador multi-linguagem de código aberto. Essa ferramenta extrai as dependências entre arquivos de sistemas em C, C++ e Java, e gera uma matriz DSM em formato PNG ou HTML.

3.2 Algoritmos escolhidos

Softwares que implementam esses algoritmos foram extraídos da página de seus desenvolvedores na internet. A Tabela 3.1 mostra os algoritmos que serão analisados e suas respectivas versões.

Algoritmo	Versão
ACDC	1.0.0
Bunch - Hill Climbing	3.5.0
Bunch - Algoritmo Genético	3.5.0
LIMBO	3.6.3

Tabela 3.1: Agoritmos escolhidos.

Um fato importante de ser apresentado, apresentado na Tabela 3.2, é que a ferramenta Bunch e Weka apresentam comportamento não-determinístico, isto é, para a mesma entrada os resultados gerados podem não ser os mesmos. Porém, o algoritmo ACDC apresenta sempre o mesmo resultado para o mesmo input.

Algoritmos	Determinístico	# Clusters Pré-definidos
ACDC	SIM	NÃO
Bunch	NÃO	NÃO
LIMBO	NÃO	SIM

Tabela 3.2: Características dos algoritmos.

3.3 Sistemas de software

Os sistemas escolhidos para serem aplicados os algoritmos são mostrados na Tabela 3.3, e são mostrados a versão, o tamanho em megabytes, número de entidades, quantidade de linhas de código (LOC) de cada sistema. O software SIPOS (Sistema para Inscrição e Seleção de Candidatos a Pós-Graduação) é um sistema utilizado na Universidade Federal da Bahia para que alunos de nível superior possam se inscrever para seletivas de Pós-Graduação e está implementado em Java.

O InGE (Indigente Game Engine), implementado em C/C++, é um motor multiplataforma de código aberto para o desenvolvimento de jogos 3D. O software Bash (Bourne Again Shell) é um interpretador de comandos para sistemas operacionais GNU implementado em C. Esse programas foram escolhidos pelo fato de encontrar suas arquiteturas de referência com facilidade, visto que, encontrar a referência da arquitetura dos softwares não é uma tarefa fácil.

A arquitetura de referência do Bash foi escrita por pesquisadores e disponibilizada no trabalho de Lutellier et al [Lutellier et al., 2015]. A arquitetura de referência do SIPOS e InGE foi disponibilizada por um desenvolvedor que participou de ambos os projetos e as escreveu em demanda deste trabalho.

Sistemas	Versão	Tamanho	Entidades	LOC
SIPOS	1.0.0	4,1 MB	215	19.557
InGE	1.0.0	3,8MB	227	59.723
Bash	4.2.53	6.7 MB	284	117.457

Tabela 3.3: Softwares escolhidos.

3.4 Execuções

Depois de selecionados os algoritmos de agrupamento e os sistemas de software para aplicá-los é necessário definir como serão realizados os testes. Assim, as execuções foram realizadas em duas fases: geração das entradas e execuções dos algoritmos. A Figura 3.1 apresenta o processo de execução deste trabalho.

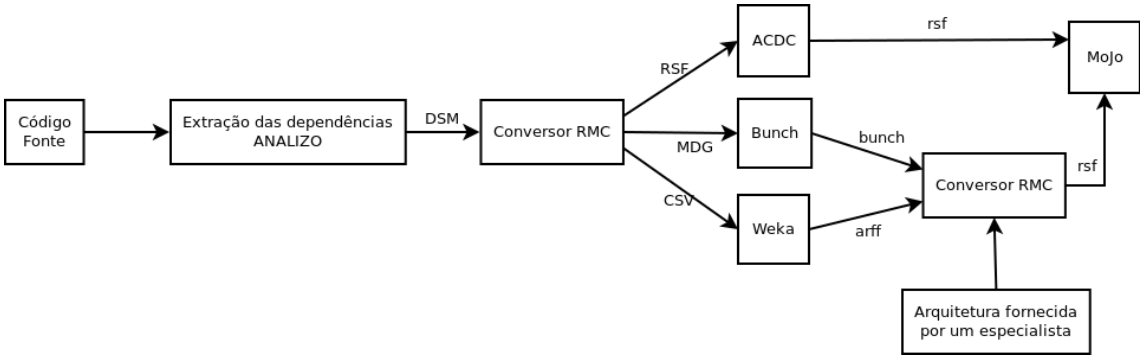


Figura 3.1: Fluxo de pesquisa.

3.4.1 Primeira fase - Geração dos inputs

Como já mencionado, a ferramenta Analizo gera uma DSM em formato HTML. A DSM em HTML foi considerada para a geração dos inputs para os algoritmos. Um conversor, chamado de Conversor RMC[1] (um acrônimo para RSF, MDG e CSV - formato de entrada para cada algoritmo estudado) mostrado na Figura 3.1, foi desenvolvido nas linguagens PHP e Javascript para a extração das informações contidas na matriz DSM em HTML e convertidas para os formatos de entrada de cada algoritmo. A seguir serão apresentados os diversos formatos de inputs utilizados por cada algoritmo.

[1] https://github.com/dennislessa/conversor_rmc

Input em formato RSF

O formato RSF é o formato gerado pela ferramenta Rigi Editor utilizado para representar os diversos tipos de relacionamentos entre os elementos de um sistema. Seu formato está representado abaixo.

RelationshipType Module1 Module2

Em "RelationshipType" representa o tipo do relacionamento (contain, call, depend, etc) entre "Module1" e "Module2" e estes representam os elementos do sistemas. Para este trabalho foi considerado apenas o tipo "depend" para representar a dependência entre os módulos 1 e 2, isto é, o elemento 1 depende do módulo 2, como apresentado abaixo.

depend Module1 Module2

Também é possível representar agrupamentos de um sistema utilizando o formato RSF. Para isso, é necessário que o tipo de relacionamento seja "contain" e o "Module1" deve representar um cluster. Um cluster é representado com o sufixo ".ss" ao final do nome do módulo.

contain Cluster.ss Module2

Esse formato tem como relacionamento o valor "contain" que significa que o "Cluster.ss" contém o elemento "Module2".

O formato RSF é requerido pelas ferramentas ACDC e MoJo. A primeira ferramenta tem como input e output arquivos de formato RSF. Como entrada o arquivo deve ter relacionamento "depend" expressando o relacionamento de dependência entre as entidades do sistema. A saída gerada pela ferramenta é um arquivo de mesmo formato contendo o agrupamento do sistema.

A segunda ferramenta, MoJo, tem como inputs dois arquivos de formato RSF representando agrupamentos. O primeiro arquivo deve ser a saída gerada por algum algoritmo de agrupamento e o segundo deve ser um agrupamento realizado por

um especialista. A saída gerada pela métrica é um número inteiro positivo expressando a quantidade mínima de operações necessárias para se transformar o primeiro agrupamento no segundo.

Input para a ferramenta Bunch

O formato de entrada exigido pela ferramenta Bunch para representar as dependências de cada elemento do sistema é uma 4-tupla. Cada uma, possui 4 campos, dos quais os dois primeiros são os módulos, o terceiro, opcional, é o peso para a relação entre os módulos e o último, também opcional, representa o tipo de relação entre os módulos. Quando não há definição das ponderações ou o tipo dos relacionamentos, significa que todos os tipos de relacionamento possuem mesma ponderação igual a 1.

Module1 Module2 [RelationshipType [RelationshipType]]

Para a realização deste trabalho não foram especificados os tipos de relacionamento e todas os relacionamentos possuem o mesmo peso. Portanto o formato de entrada para a ferramenta, representa apenas relação de dependência do módulo 1 para com o módulo 2.

Module1 Module2

Assim, como foi definido o formato de entrada, há uma dependência entre o módulo 1 e o módulo 2, ou seja, o módulo 1 depende do módulo 2. Neste trabalho, usando os scripts desenvolvidos, esse formato é gerado a partir da DSM seguindo as definições apresentadas anteriormente em que as dependências da DSM são traduzidas para o formato usado pela ferramenta Bunch.

Ao realizar o agrupamento, a ferramenta Bunch gera um arquivo *.bunch* que representa o resultado do agrupamento [Wen, 2003], isto é, os *clusters* gerados e os elementos contidos nesses *clusters*, representado abaixo.

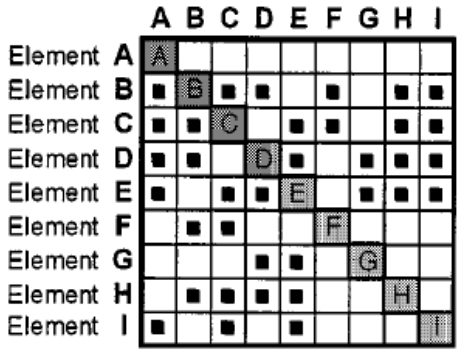
SS(ModuleName) = elemento1, elemento2, ..., elementoN

Como apresentado anteriormente, a ferramenta MoJo utiliza o formato RSF como input, portanto, fez-se necessário a conversão do tipo *.bunch* para o tipo *.rsf*. Assim, após a conversão, o arquivo RSF gerado terá o seguinte formato:

contain ModuleName.ss elemento1
contain ModuleName.ss elemento2
⋮
contain ModuleName.ss elementoN

Input em formato CSV

Esse formato é um dos vários formatos utilizados pela ferramenta Weka para a utilização de algoritmos de categorização, agrupamento, associação e regressão. Utilizando os scripts desenvolvidos, converteu-se os dados contidos na DSM para o formato CSV. Onde há dependência entre os elementos na DSM é representado como 1 em CSV e 0 caso contrário, como mostra a Figura 3.2.



(a) DSM

A,	B,	C,	D,	E,	F,	G,	H,	I
1,	0,	0,	0,	0,	0,	0,	0,	0
1,	1,	1,	1,	1,	1,	0,	1,	1
1,	1,	1,	0,	1,	1,	0,	1,	1
1,	1,	0,	1,	1,	0,	1,	1,	1
1,	0,	1,	1,	1,	0,	1,	1,	1
0,	1,	1,	0,	0,	1,	0,	0,	0
0,	0,	0,	1,	1,	0,	1,	0,	0
0,	1,	1,	1,	1,	0,	0,	1,	0
1,	0,	1,	0,	1,	0,	0,	0,	1

(b) Arquivo CSV

Figura 3.2: Conversão da DSM para um arquivo CSV.

3.4.2 Segunda fase - Execução dos algoritmos

Após a conversão dos dados contidos na DSM para os formatos de cada algoritmo é preciso definir como serão executados os algoritmos, isto é, quantas execuções serão realizadas e quais informações serão apreciadas.

Para se obter o tempo de execução e a quantidade de *clusters* obtidos por cada algoritmo, foram analisados ao final das execuções de cada um. A ferramenta Bunch, apresenta a quantidade de *clusters* e o tempo de execução. O algoritmo LIMBO possui como um dos argumentos a quantidade de *clusters* informada pelo usuário e o tempo de execução é mostrado a hora de início e fim do agrupamento como log da execução. Com isso, a diferença entre elas representa o tempo de execução do algoritmo.

O algoritmo ACDC foi executado pelo terminal Linux e para calcular o tempo de execução do algoritmo foi utilizado o comando "*time*" antes da sequência de comandos do algoritmo e assim calcula o "*time real*" da execução do algoritmo.

Configurações das ferramentas

Nessa seção será apresentado as configurações de cada ferramenta separadamente. Primeiramente serão apresentadas as configurações da ferramenta Bunch (algoritmos Hill Climbing e Genético), em seguida ferramenta Weka (LIMBO) e ACDC.

Para a ferramenta ACDC foram mantidas as configurações padrões. Visto que a ferramenta permite a alteração de parâmetros como a ordem ou quais padrões devem ser executados, tamanho do subgrafo formado e argumentos de visualização. Por tanto, foram deixados padrões para que todos os padrões (*corpo-cabeçalho*, *subgrafo dominante* e *adoção de órfãos*) fossem executados naturalmente, assim como, os demais argumentos visto que esses não gerariam resultados relevantes a este estudo.

Para a ferramenta Bunch, primeiramente foram configuradas as opções do agrupamento, isto é, foi escolhida a função objetivo, como mostra a Figura A.1. Em seguida foi escolhida o formato de saída Figura A.2. A função objetiva escolhida, já descrita no capítulo anterior foi *ITurboMQ* e o formato escolhido foi *Text* - esse formato ao final do agrupamento gera um arquivo *.bunch*.

Para a ferramenta Weka foram configurados os números de *clusters* a cada execução. O valor inicial para o número de cluster é 5 *clusters* e incrementado em 5 *clusters* a cada teste. A variável *phi* foi mantida padrão com o valor **0.0** visto que para esse valor não há perda de informação [Andritsos et al., 2003]. Ainda, segundo [Andritsos et al., 2003], o fator de ramificação (*branchingFactor*) não afeta significativamente a qualidade de agrupamento. Baseado nisso, este valor foi mantido

padrão com valor **4**. As variáveis *seed* e *space* foram mantidas com seu valor padrão **318** e **5000** respectivamente. Essas configurações são apresentadas na Figura A.3.

Capítulo 4

Apresentação e Discussão dos Resultados

Como descrito no Capítulo 3, cada algoritmo foi configuarados e foram executados 10 vezes em cada software. Os dados como quantidade de cluster, a distância MoJo e o tempo de execução são mostrados em tabelas neste capítulo.

Este capítulo está dividido em duas partes. A primeira parte são apresentados os resultados obtidos em cada teste realizado em cada sistema de software. Esses resultados estão divididos em duas tabelas por software: a primeira mostra os dados referentes a quantidade de cluster e a distânca MoJo e a segunda tabela mostra o tempo de execução dos algoritmos em cada teste realizado. E a segunda parte é a discussão sobre os resultados.

Para a exibição dos resultados, bem como para as análises e discussões seguintes são utilizadas siglas para se referirem aos algoritmos Hill Climbing (HC) e o Algoritmo Genético (AG).

4.1 Resultados obtidos

Nesta seção são mostrados os resultados dos testes realizados. Como mencionado anteriormente são duas tabelas para cada software. A Tabela 4.1 mostra valores para o número de clusters e a distância MoJo referentes ao software Bash. A Tabela 4.4 mostra os tempos de execuções de cada algoritmo para o mesmo software. Assim como, as Tabelas 4.2 e 4.5 para o software InGE e as Tabelas 4.3 e 4.6 para o software

SIPOS.

Teste	ACDC		HC		AG		LIMBO	
	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo
01	26	80	20	84	54	122	5	113
02	26	80	13	108	49	120	10	114
03	26	80	21	73	86	132	15	109
04	26	80	21	80	49	112	20	108
05	26	80	17	84	50	117	25	108
06	26	80	22	90	81	130	30	105
07	26	80	12	85	52	116	35	108
08	26	80	24	88	84	126	40	109
09	26	80	12	86	46	118	45	123
10	26	80	11	98	77	125	50	109
Media	26	80	17,3	87,6	62,8	121,7	27,5	110,6
DP	0	0	4,9	9,64	16,804	6,342	15,138	5,06

Tabela 4.1: Número de clusters e distância MoJo - software Bash.

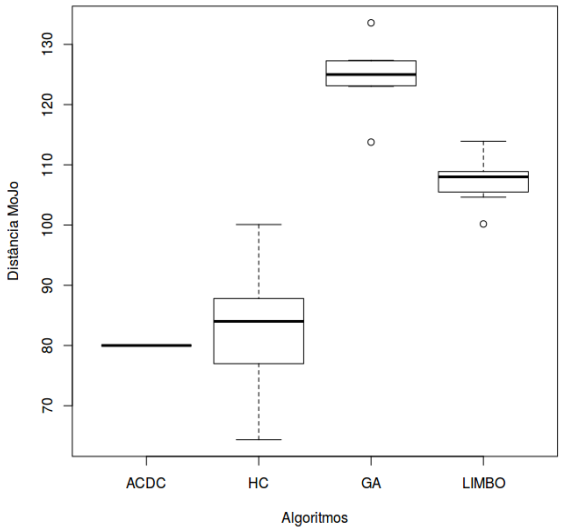
Teste	ACDC		HC		AG		LIMBO	
	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo
01	15	23	5	51	50	89	5	53
02	15	23	4	40	41	83	10	53
03	15	23	9	41	40	84	15	52
04	15	23	8	29	44	81	20	48
05	15	23	6	30	45	79	25	50
06	15	23	8	32	65	99	30	55
07	15	23	7	34	38	82	35	50
08	15	23	4	32	45	88	40	57
09	15	23	12	37	39	87	45	60
10	15	23	6	30	41	83	50	61
Media	15	23	6,9	35,6	44,8	85,5	27,5	53,9
DP	0	0	2,469	6,85	7,941	5,7	15,138	4,332

Tabela 4.2: Número de clusters e distância MoJo - software InGE.

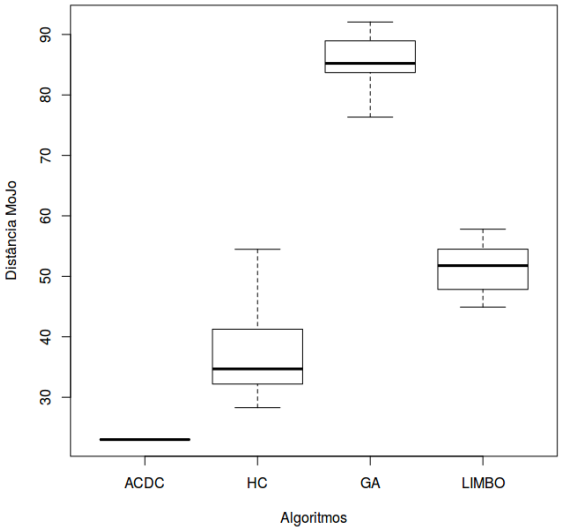
Teste	ACDC		HC		AG		LIMBO	
	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo	Clusters	MoJo
01	21	81	13	93	31	104	5	130
02	21	81	8	86	29	96	10	120
03	21	81	15	94	33	100	15	116
04	21	81	8	87	30	99	20	109
05	21	81	7	92	26	91	25	118
06	21	81	16	89	33	100	30	118
07	21	81	12	84	27	101	35	118
08	21	81	23	96	32	102	40	121
09	21	81	7	91	34	97	45	130
10	21	81	8	96	28	99	50	123
Media	21	81	11,7	90,8	30,3	98,9	27,5	120,3
DP	0	0	5,208	4,185	2,751	3,604	15,138	6,308

Tabela 4.3: Número de clusters e distância MoJo - software SIPOS.

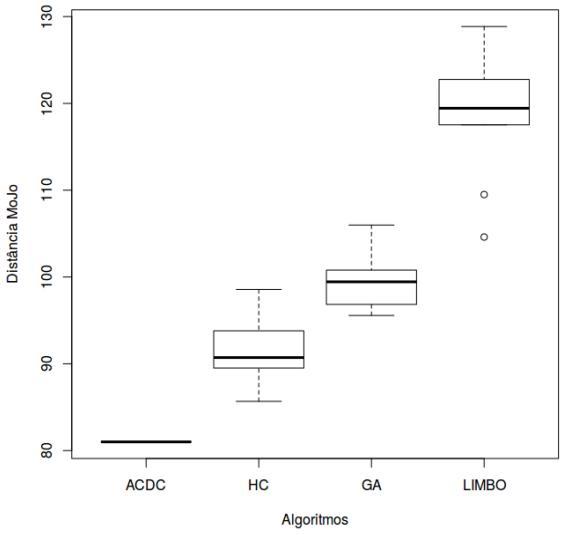
Os resultados apresentados pelas tabelas acima mostraram que o algoritmo ACDC apresentou os melhores resultados, seguido pelo algoritmo Hill Climbing, em todos os testes realizados. O algoritmo LIMBO foi melhor que o Algoritmo Genético quando executado para o software Bash e InGE (ver Figuras 4.1(a) e 4.1(b)) e o pior para o software SIPOS como mostra a Figura 4.1(c). A Figura 4.1 apresenta uma visão melhor desses resultados.



(a) Distância MoJo por algoritmo para Bash



(b) Distância MoJo por algoritmo para InGE



(c) Distância MoJo por algoritmo para SIPOS

Figura 4.1: Gráficos mostrando a distância MoJo de cada algoritmo para cada software.

A seguir são apresentadas as tabelas dos tempos de execução de cada algoritmo para cada teste em cada software.

Teste	ACDC	HC	AG	LIMBO
01	0,369	0,249	8204,638	1,064
02	0,428	0,286	8832,409	1,057
03	0,395	0,302	7210,191	1,092
04	0,402	0,273	9018,269	1,088
05	0,369	0,300	8470,535	1,069
06	0,414	0,483	5989,290	1,081
07	0,339	0,258	9434,895	1,133
08	0,392	0,383	7105,257	1,131
09	0,297	0,325	7886,437	1,068
10	0,374	0,293	7362,417	1,052
Media	0,378	0,315	7951,433	1,084
DP	0,038	0,07	1049,609	0,029

Tabela 4.4: Tempo de execução dos algoritmos - software Bash.

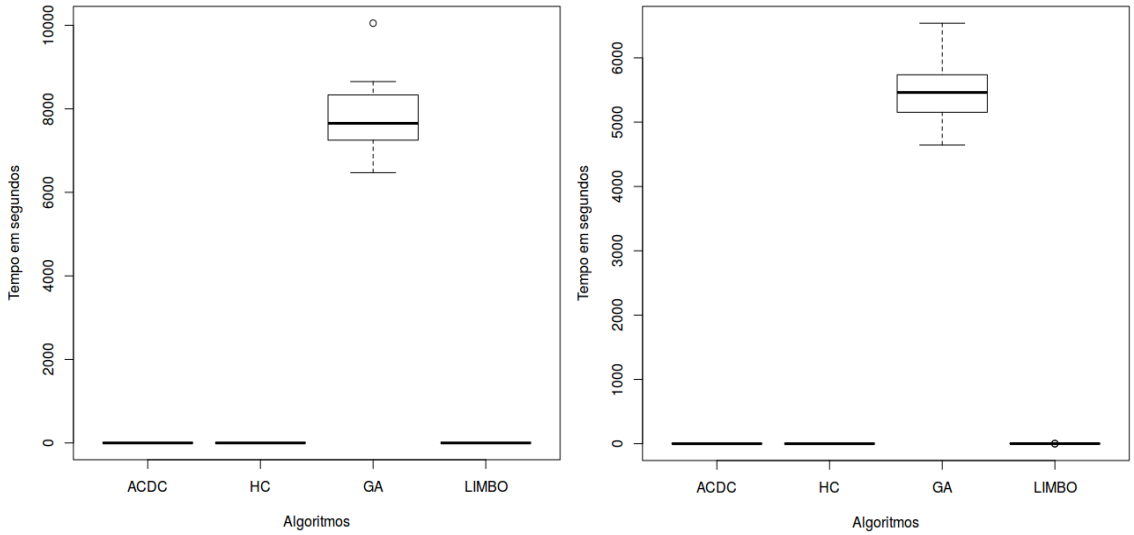
Teste	ACDC	HC	AG	LIMBO
01	0,308	0,117	5499,711	1,025
02	0,327	0,168	6035,428	1,057
03	0,318	0,161	5690,281	1,047
04	0,309	0,141	6164,771	1,103
05	0,324	0,117	5865,741	1,048
06	0,317	0,14	4179,552	1,093
07	0,328	0,134	4838,804	1,069
08	0,297	0,115	5713,211	1,114
09	0,321	0,114	5517,61	1,029
10	0,03	0,15	5134,15	1,044
Media	0,315	0,141	5463,926	1,063
DP	0,01	0,022	600,55	0,031

Tabela 4.5: Tempo de execução dos algoritmos - software InGE.

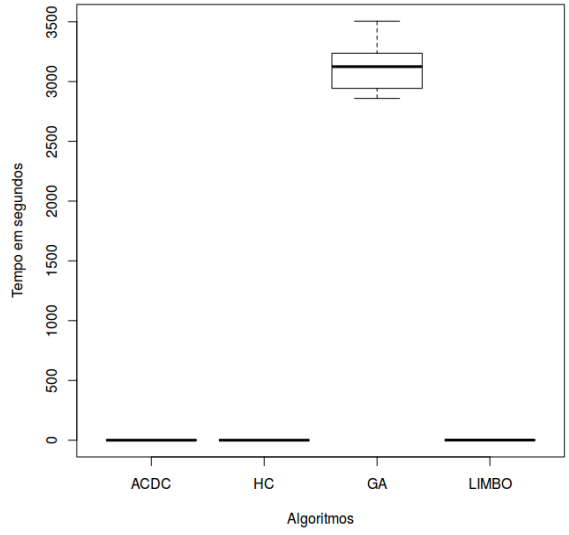
Teste	ACDC	HC	AG	LIMBO
01	0,273	0,094	3287,529	1,062
02	0,276	0,131	3094,973	1,078
03	0,244	0,167	3167,3	1,106
04	0,297	0,074	3190,929	1,064
05	0,299	0,076	2781,629	1,056
06	0,294	0,101	3179,17	1,125
07	0,3	0,096	2702,194	1,06
08	0,315	0,111	3123,966	1,004
09	0,28	0,167	3127,256	1,011
10	0,24	0,074	2740,201	1,06
Media	0,282	0,109	3039,518	1,063
DP	0,024	0,035	212,932	0,037

Tabela 4.6: Tempo de execução dos algoritmos - software SIPOS.

Os dados apresentados pelas tabelas anteriores, referentes aos tempos de execução, indicam que os todos os algoritmos, exceto o Algoritmo Genético, obteveram ótimos resultados, isto é, executaram a tarefa de agrupamento em tempos muito pequenos. E ainda é possível ressaltar que o desvio padrão (DP) dos mesmos algoritmos referidos se manteve próximo de zero o que indica uma estabilidade dos algoritmos quanto a sua execução.



(a) Tempo de execução por algoritmo para Bash (b) Tempo de execução algoritmo para InGE



(c) Tempo de execução algoritmo para SIPOS

Figura 4.2: Gráficos mostrando o tempo de execução de cada algoritmo para cada software.

4.2 Discussão dos resultados

Nesta seção os resultados obtidos pelos experimentos são discutidos e comparados com algumas obras da literatura.

[Andritsos and Tzerpos, 2005] afirmam em seus experimentos que o algoritmo LIMBO se mostrou melhor que os demais algoritmos por eles escolhidos (ACDC, Bunch, SL, CL, WA e UA). Porém, a Figura 4.1 mostrou que em nosso experimento o algoritmo LIMBO não apresentou tal desempenho. A Tabela 4.7 mostra a média dos valores arrendondados para o inteiro mais próximo como foi feito sem seus experimentos. Os valores de cada algoritmo podem ser comparados ao valor de *"Autoridade"* de cada sistema - que representa a arquitetura de referência dos softwares.

Assim, também é possível afirmar que o algoritmo LIMBO não obteve os melhores resultados. Uma possível explicação para isso poderia ser o fato deles terem obtido as arquiteturas de referência dos softwares, por eles testados, através do uso de ferramentas de extração de depedências e da interação entre pesquisadores e os desenvolvedores do sistema o que pode ter influenciado na descrição da arquitetura de referência.

Algoritmo	Bash		InGE		SIPOS	
	Cluster	MoJo	Cluster	MoJo	Cluster	MoJo
Autoridade	14	—	17	—	10	—
ACDC	26	80	15	23	21	81
HC	17	88	7	36	12	91
AG	63	122	45	86	30	99
LIMBO	28	111	28	54	28	120

Tabela 4.7: Número de clusters e distância MoJo - software Bash.

O Algoritmo Genético também não obteve bons resultados em dois dos sistemas, Bash e InGE. Contudo, os seus resultados para o software SIPOS não ficaram muito distantes dos algoritmos ACDC e Bunch. Porém, é importante ressaltar que o tempo médio de execução desse algoritmo atingiu valores muito elevados se comparados aos

demais algoritmos, como mostra a Tabela 4.8.

Algoritmo	Bash	InGE	SIPOS
ACDC	0,378	0,315	0,282
HC	0,315	0,141	0,109
AG	7951,433	5463,926	3039,518
LIMBO	1,084	1,063	1,063

Tabela 4.8: Tempo médio de execução dos algoritmos.

O algoritmo ACDC obteve os melhores resultados para os três sistemas de software, o que significa que este é mais autoritário que o Bunch. Porém, [Wu et al., 2005] em seu trabalho observou o contrário, afirmando que ACDC é um pouco menos autoritário que Bunch. Talvez uma explicação para tal acontecimento seria o fato de que em seu trabalho, diferentemente deste, Wu compara algoritmos de agrupamento a partir de versões mensais de softwares baseando-se na estrutura hierárquica de diretórios além do uso de ferramentas para extrair a arquitetura de referência dos sistemas estudados.

Em seu trabalho [Wen, 2003] diz que os agrupamentos obtidos pelo Bunch e ACDC podem ser de grande ajuda na atividade de recuperação de arquitetura. Os dados apresentados pela Figura 4.1 e pelas Tabelas 4.7 e 4.8 também demonstram que os algoritmos com melhores resultados foram o ACDC e Bunch Hill Climbing.

Contudo, [Lutellier et al., 2015] em seus estudos não afirmam qual técnica é melhor, porém, definem algumas diretrizes para ajudar na escolha da técnica a ser escolhida. Segundo eles, ACDC e Bunch são mais aptos para grandes softwares e o algoritmo LIMBO quando já se tem algum conhecimento da arquitetura, já que este permite a escolha do número de clusters.

Capítulo 5

Conclusão

A evolução dos softwares ocorre muito rápido e com a mesma velocidade atividades de manutenção, adição, remoção ou alteração de funcionalidades, correção de bugs entre outras são exigidas e realizadas para manter o software em pleno funcionamento. Com isso, é natural que a complexidade do software aumente e este se torne menos flexível.

Por isso, a documentação de um software deve estar sempre sincronizada com as versões ou alterações realizadas. A fim de facilitar a realização dessas tarefas ou, até mesmo, para que tais intervenções no software não possam comprometer o funcionamento do mesmo. Porém, muitos softwares não possuem documentação ou quando a possuem esta não está atualizada. Assim, atividades de manutenção, ocorridas durante a fase de evolução do software, facilitam o aumento da complexidade e a diminuição da sua flexibilidade do sistema.

Nesse contexto, as técnicas de recuperação de arquitetura de software se fazem necessárias para obter a documentação da arquitetura, quando esta é inexistente, ou para obter a arquitetura atualizada do software depois de tantas interveções de manutenção. Uma dessas técnicas é utilizar algoritmos de agrupamento para propor um modelo da arquitetura do software. Essas técnicas são avalidas segundo algumas métricas e critérios para testar o quão diferentes ou semelhantes são com agrupamentos criados por expecialistas ou o quanto são estáveis e confiáveis.

Este trabalho comparou quatro dos seis algoritmos mais utilizados em recuperação da arquitetura de software, segundo [Oliveira and Andrade, 2014]: ACDC, Bunch e LIMBO. Para a avaliação foi utilizada a métrica MoJo avaliando o critério

de autoridade. Os resultados obtidos mostraram que o algoritmo ACDC obteve os melhores resultados em todos testes realizados. O trabalho contribuiu com o desenvolvimento de um conversor (Conversor RMC) para a conversão das DSMs em HTML para o formato de entrada de cada algoritmo.

5.1 Limitações

Uma das limitações consiste nos critérios de avaliação dos algoritmos, isto é, foi utilizado apenas o critério de *Autoridade* que mede unicamente se um agrupamento gerado por um algoritmo é semelhante a um agrupamento gerado manualmente por um especialista. E alguns aspectos como qualidade do agrupamento e a estabilidade do algoritmo não são avaliados.

Outra limitação está em obter softwares que disponibilizem suas arquiteturas de referência. Visto que esse tipo de documentação não se encontra disponível com facilidade. Um número de apenas três softwares pode ser considerado um número pequeno. E ainda é preciso ressaltar que a variação da quantidade de entidades de um software para outro é pequena, o que pode refletir nos resultados, uma vez que os número de entidades sendo semelhantes os resultados seriam semelhantes.

5.2 Trabalhos futuros

Muitas técnicas têm sido propostas no últimos anos para a recuperação da arquitetura de software. Um complemento a este estudo seria comparar os mesmos algoritmos com softwares que possuam grandes variações de entidades entre si, isto é, a variação da quantidade de entidades entre os softwares deve ser maior do que as aqui apresentadas.

Como apresentado no Capítulo 2, do ponto de vista da taxonomia da SAR, um interessante estudo poderia ser realizado levando em consideração os *inputs* não arquiteturais baseado em código fonte e organização física e analisar os *ouputs* de conformidade vertical. Desta maneira, o objetivo seria verificar se a arquitetura proposta pelo algoritmo de agrupamento está de acordo com o código-fonte e agrupado de acordo com a estruturas de pastas.

Referências Bibliográficas

[Andritsos et al., 2003] Andritsos, P., Tsaparas, P., Miller, R. J., and Sevcik, K. C. (2003). Limbo: A scalable algorithm to cluster categorical data.

[Andritsos and Tzerpos, 2005] Andritsos, P. and Tzerpos, V. (2005). Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165.

[Anquetil and Lethbridge, 1999] Anquetil, N. and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255. IEEE.

[Beyer and Noack, 2005] Beyer, D. and Noack, A. (2005). Clustering software artifacts based on frequent common changes. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 259–268. IEEE.

[Bittencourt and Guerrero, 2009] Bittencourt, R. A. and Guerrero, D. D. S. (2009). Comparison of graph clustering algorithms for recovering software architecture module views. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 251–254. IEEE.

[Bowman et al., 1999] Bowman, I. T., Holt, R. C., and Brewster, N. V. (1999). Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering*, pages 555–563. ACM.

[Browning, 2001] Browning, T. R. (2001). Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering management*, 48(3):292–306.

[Ducasse and Pollet, 2009] Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

[Gall et al., 1996] Gall, H., Jazayeri, M., Klosch, R., Lugmayr, W., and Trausmuth, G. (1996). Architecture recovery in ares. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 111–115. ACM.

[Garcia et al., 2013] Garcia, J., Ivkovic, I., and Medvidovic, N. (2013). A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496. IEEE.

[Garlan, 2000] Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM.

[Lakhotia and Gravley, 1995] Lakhotia, A. and Gravley, J. M. (1995). Toward experimental evaluation of subsystem classification recovery techniques. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 262–269. IEEE.

[Linden, 2009] Linden, R. (2009). Técnicas de agrupamento. *Revista de Sistemas de Informação da FSMA*, 1(4):18–36.

[Lung, 1998] Lung, C.-H. (1998). Software architecture recovery and restructuring through clustering techniques. In *Proceedings of the third international workshop on Software architecture*, pages 101–104. ACM.

[Lung and Zaman, 2004] Lung, C.-H. and Zaman, M. (2004). Using clustering technique to restructure programs. In *Software Engineering Research and Practice*, pages 853–860.

[Lutellier et al., 2015] Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., and Kroeger, R. (2015). Comparing software architecture re-

covery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78. IEEE.

[Mancoridis et al., 1998] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y.-F., and Gansner, E. R. (1998). Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, volume 98, pages 45–52.

[Maqbool and Babri, 2007] Maqbool, O. and Babri, H. (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780.

[Maqbool and Babri, 2004] Maqbool, O. and Babri, H. A. (2004). The weighted combined algorithm: A linkage algorithm for software clustering. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 15–24. IEEE.

[Mitchell, 2002] Mitchell, B. S. (2002). *A heuristic search approach to solving the software clustering problem*. PhD thesis, Drexel University.

[Naseem et al., 2011] Naseem, R., Maqbool, O., and Muhammad, S. (2011). Improved similarity measures for software clustering. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 45–54. IEEE.

[Oliveira and Andrade, 2014] Oliveira, L. P. d. and Andrade, S. (2014). Um framework para recuperação arquitetural independente de plataforma.

[Shtern and Tzerpos, 2012] Shtern, M. and Tzerpos, V. (2012). Clustering methodologies for software engineering. *Advances in Software Engineering*, 2012:1.

[Silva and Bittencourt, 2013] Silva, D. E. U. and Bittencourt, R. A. (2013). Avaliação da recuperação arquitetural de visões modulares através de técnicas de agrupamento.

[Sivanandam and Deepa, 2007] Sivanandam, S. and Deepa, S. (2007). *Introduction to genetic algorithms*. Springer Science & Business Media.

[Slonim and Tishby, 1999] Slonim, N. and Tishby, N. (1999). Agglomerative information bottleneck. *Neural Information Processing Systems*, pages 617–623.

[SOUZA, 2010] SOUZA, R. R. G. (2010). Modelos computacionais realistas para dependências entre entidades de software.

[Terceiro et al., 2010] Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L. R., Almeida, L., Chavez, C., and Kon, F. (2010). Analizo: an extensible multi-language source code analysis and visualization toolkit. In *Brazilian conference on software: theory and practice (Tools Session)*.

[Tzerpos and Holt, 1999] Tzerpos, V. and Holt, R. C. (1999). Mojo: A distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 187–193. IEEE.

[Tzerpos and Holt, 2000a] Tzerpos, V. and Holt, R. C. (2000a). Acdc: An algorithm for comprehension-driven clustering. In *wcre*, pages 258–267.

[Tzerpos and Holt, 2000b] Tzerpos, V. and Holt, R. C. (2000b). On the stability of software clustering algorithms. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 211–218. IEEE.

[Vasconcelos, 2004] Vasconcelos, A. (2004). Uma abordagem para recuperação de arquitetura de software visando sua reutilização em domínios específicos. *Exame de Qualificação, COPPE, UFRJ, Rio de Janeiro, Brasil*.

[Wen, 2003] Wen, Z. (2003). *An optimal algorithm and extensions for the MoJo distance measure*. PhD thesis, Citeseer.

[Wiggerts, 1997] Wiggerts, T. A. (1997). Using clustering algorithms in legacy systems remodularization. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 33–43. IEEE.

[Wu et al., 2005] Wu, J., Hassan, A. E., and Holt, R. C. (2005). Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 525–535. IEEE.

Apêndice A

Configurações das ferramentas

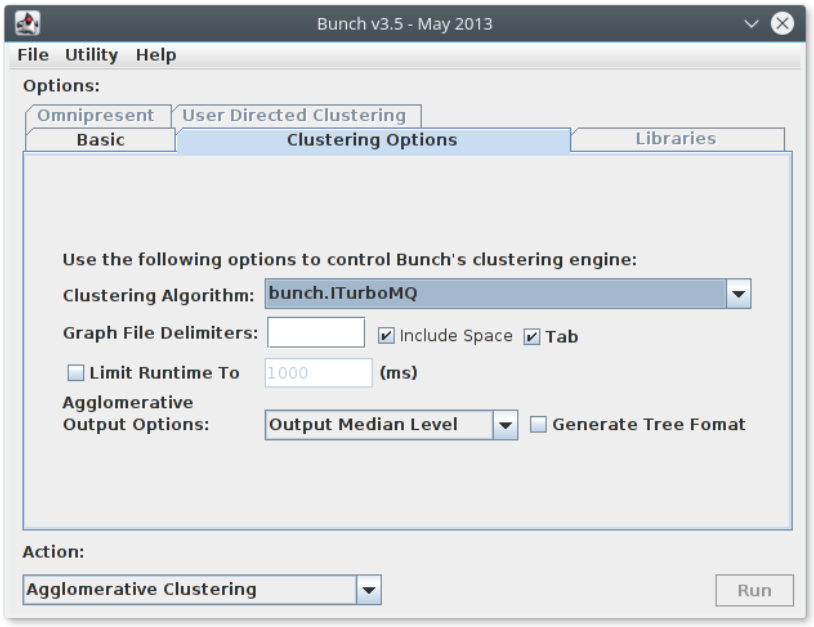


Figura A.1: Opções do agrupamento - Bunch.

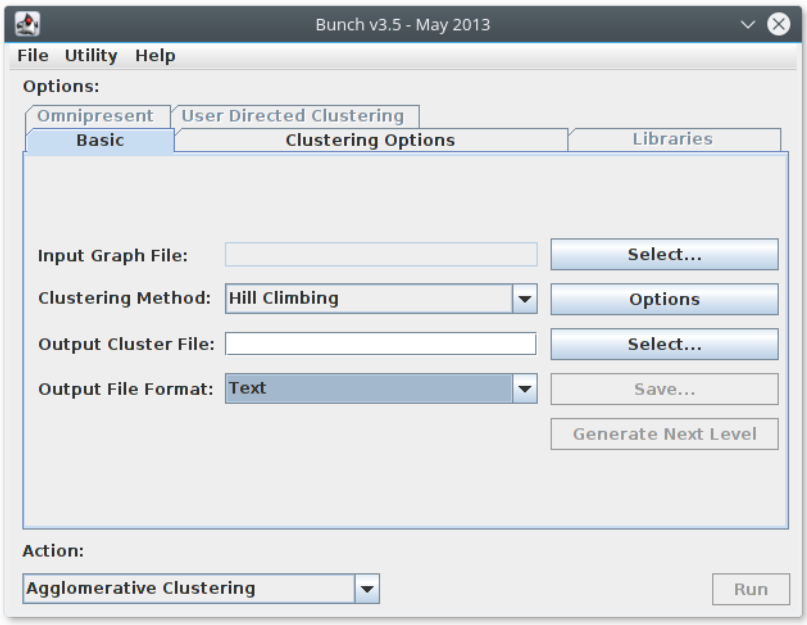


Figura A.2: Escolha do formato de saída - Bunch.

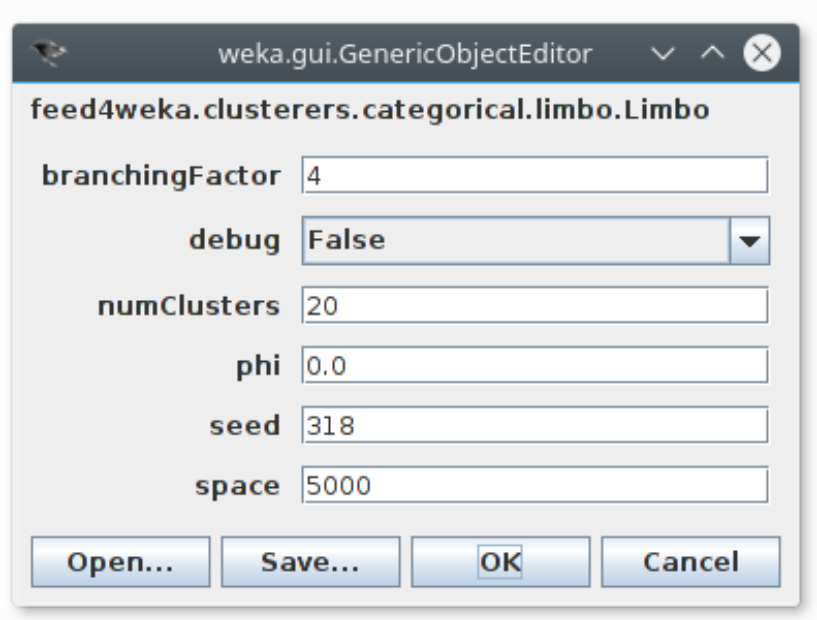


Figura A.3: Configurações - Weka