



Universidade Estadual de Feira de Santana  
Programa de Pós-Graduação em Computação Aplicada

# Avaliação da Recuperação Arquitetural de Visões Modulares de Software a partir de Técnicas de Agrupamento

Douglas Eder Uno Silva

Feira de Santana

2019



Universidade Estadual de Feira de Santana  
Programa de Pós-Graduação em Computação Aplicada

Douglas Eder Uno Silva

**Avaliação da Recuperação Arquitetural de Visões  
Modulares de Software a partir de Técnicas de  
Agrupamento**

Dissertação apresentada à Universidade Estadual de Feira de Santana como parte dos requisitos para a obtenção do título de Mestre em Computação Aplicada.

Orientador: Roberto Almeida Bittencourt

Coorientador: Rodrigo Tripodi Calumby

Feira de Santana

2019

Entrega o teu caminho ao Senhor, confia nele, e ele o fará.  
**Salmos 37:5**

## Ficha Catalográfica – Biblioteca Central Julieta Carteado

S578a Silva, Douglas Eder Uno

Avaliação da recuperação arquitetural de visões modulares de software a partir de técnicas de agrupamento / Douglas Eder Uno Silva . - Feira de Santana, 2019.

92 f.: il.

Orientador: Roberto Almeida Bittencourt

Dissertação (Mestrado) – Universidade Estadual de Feira de Santana, Programa de Pós-graduação em Computação Aplicada, 2019.

1. Arquitetura de software. 2. Avaliação experimental – visão modular. I. Bittencourt, Roberto Almeida, orient. II. Universidade Estadual de Feira de Santana. III. Título.

CDU: 681.3

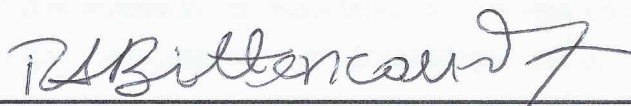
Douglas Eder Uno Silva

**Avaliação da Recuperação Arquitetural de Visões Modulares de  
Software a partir de Técnicas de Agrupamento**

Dissertação apresentada à Universidade Estadual de Feira de Santana como parte dos requisitos para a obtenção do título de Mestre em Computação Aplicada.

Feira de Santana, 14 de fevereiro de 2019

**BANCA EXAMINADORA**



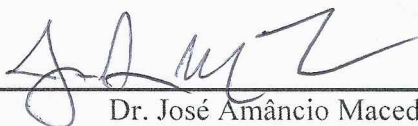
---

Dr. Roberto Almeida Bittencourt (Orientador)  
Universidade Estadual de Feira de Santana



---

Dr. Manoel Gomes de Mendonça Neto  
Universidade Federal da Bahia



---

Dr. José Amâncio Macedo Santos  
Universidade Estadual de Feira de Santana

# Abstract

Architecture module views of software are made up of modules with distinct functional responsibilities but with dependencies between them. Previous work has evaluated architecture recovery techniques of module views in order to better understand their strengths and weaknesses. In this context, different similarity metrics are used to evaluate such techniques, especially those based on clustering algorithms. However, few studies try to evaluate whether such metrics accurately capture the similarities between two clusters. Among the similarity metrics in the literature, we can cite examples from both the field of software engineering and from other fields (e.g., classification). This work evaluates six cluster similarity metrics through intrinsic quality and stability metrics and the use of software architecture models proposed by developers. To do so, we used the dimensions of stability and authoritativeness, in accordance with what has been discussed in the literature. For authoritativeness, the concentration statistics of the MeCl metric were higher, in comparison with the other similarity metrics. However, in the absence of architectural models, the Purity metric shows better results. As architecture models are very relevant to software engineers, we understand that the MeCl metric is the most appropriate. For stability, all metrics have values close to unity, despite the presence of outliers. Here as well, the MeCl metric was considered the best because of its superiority in this item. Being better in both dimensions, especially in authoritativeness, we decided to use the MeCl metric as the basis for comparison of clustering algorithms. We compared, using the MeCl metric, four agglomerative clustering algorithms in the context of four software systems. For both authoritativeness and stability, the SL90 algorithm produced higher values in two of the four systems studied by comparing the data series generated by all algorithms. In this case, the SL90 agglomerative algorithm was the best. In conclusion, we empirically realized that the MeCl metric is the best metric to measure group similarity; regarding the clustering algorithms, no algorithm exceeds the others in all comparisons, although SL90 presented better results in two of the four systems we analyzed.

**Keywords:** software evolution, software architecture, module view. software architecture recovery, experimental evaluation, metrics.

# Resumo

Visões arquiteturais modulares de software são formadas por módulos com responsabilidades distintas mas com dependências entre si. Diversos trabalhos avaliam as técnicas de recuperação arquitetural de visões modulares para entender melhor seus pontos fortes e fracos. Neste contexto, diferentes métricas de similaridade são utilizadas para avaliar tais técnicas, especialmente as que usam algoritmos de agrupamento. Contudo, poucos trabalhos avaliam se tais métricas realmente capturam de maneira fidedigna as similaridades entre dois agrupamentos. Dentre as métricas de similaridade existentes na literatura, pode-se citar métricas tanto da área da engenharia de software quanto de outras áreas (e.g., classificação). Este trabalho avalia seis métricas de similaridade de agrupamentos através de medidas intrínsecas de qualidade e estabilidade e da utilização de modelos arquiteturais propostos por desenvolvedores. Para tanto, usamos as dimensões de estabilidade e autoridade, em conformidade com a literatura. Para a autoridade, as estatísticas de concentração da métrica MeCl foram maiores, em comparação com as demais métricas de similaridade. Contudo, na ausência de modelos arquiteturais, a métrica Pureza apresenta melhores resultados. Como os modelos arquiteturais são muito relevantes para os engenheiros de software, entendemos que a métrica MeCl é a mais adequada. Para a estabilidade, todas as métricas apresentam valores próximos da unidade, apesar da presença de *outliers*. Aqui também, a métrica MeCl foi considerada a melhor devido à sua superioridade neste item. Sendo melhor nas duas dimensões, especialmente em autoridade, usamos a métrica MeCl como base para comparação de algoritmos de agrupamento. Comparamos, usando a métrica MeCl, quatro algoritmos de agrupamento aglomerativos no contexto de quatro sistemas de software. Tanto para a autoridade quanto a estabilidade, o algoritmo SL90 gerou valores mais altos em dois dos quatro sistemas estudados ao comparar as séries de dados geradas por todos os algoritmos. Neste caso, o algoritmo aglomerativo SL90 foi o melhor. Em conclusão, percebemos empiricamente que a métrica MeCl é a melhor métrica para medir similaridade de agrupamentos; já em relação aos algoritmos de agrupamento, nenhum algoritmo supera os demais em todas as comparações, apesar de o SL90 ter apresentado melhores resultados em dois dos quatro sistemas analisados.

**Palavras-chave:** evolução de software, arquitetura de software, visão modular, recuperação arquitetural, avaliação experimental, métricas.

# Prefácio

Esta dissertação de mestrado foi submetida a Universidade Estadual de Feira de Santana (UEFS) como requisito parcial para obtenção do grau de Mestre em Computação Aplicada.

A dissertação foi desenvolvida no Programa de Pós-Graduação em Computação Aplicada (PGCA) tendo como orientador o professor Dr. **Roberto Almeida Bitencourt** e como coorientador o professor Dr. **Rodrigo Tripodi Calumby**.



# Agradecimentos

Agradeço primeiramente a Deus pelo dom da vida e pela oportunidade de poder obter o grau de mestre. Dedico este trabalho a minha família. Meus pais Bosco e Sônia, minha esposa Cristiane e minhas irmãs Emile, Elaine e Elis. Sem o incentivo de vocês, nada disto seria possível. Dedico também este trabalho ao meu orientador prof. Roberto Almeida Bittencourt e ao meu coorientador prof. Rodrigo Tripodi Calumby pela atenção, dedicação e paciência ao longo deste processo.

# Sumário

Abstract	i
Resumo	ii
Prefácio	iii
Agradecimentos	iv
Sumário	vi
Lista de Publicações	vii
Lista de Tabelas	viii
Lista de Figuras	ix
Lista de Abreviações	x
<b>1 Introdução</b>	<b>1</b>
1.1 Objetivos . . . . .	5
1.2 Contribuições . . . . .	6
1.3 Organização do Trabalho . . . . .	6
<b>2 Fundamentação Teórica</b>	<b>7</b>
2.1 Recuperação Arquitetural . . . . .	7
2.2 Técnicas de recuperação arquitetural de visões modulares . . . . .	8
2.3 Representação de entidades e suas relações para a recuperação arquitetural . . . . .	9
2.4 Recuperação de Informação . . . . .	11
2.4.1 Extração de vocabulário de software . . . . .	11
2.4.2 Representação vetorial das entidades de código . . . . .	13
2.5 Similaridade entre entidades . . . . .	15
2.6 Algoritmos hierárquicos aglomerativos para recuperação arquitetural por <i>clustering</i> . . . . .	17
2.7 Métricas de avaliação da recuperação arquitetural . . . . .	18

2.7.1	Modelos de referência ou <i>ground truth architectures</i> . . . . .	18
2.7.2	Dimensões de avaliação . . . . .	19
2.7.3	Métricas de similaridade de agrupamentos . . . . .	20
2.7.4	Métricas intrínsecas de qualidade para avaliação de agrupamentos . . . . .	27
2.8	Trabalhos Relacionados . . . . .	28
<b>3</b>	<b>Metodologia</b>	<b>31</b>
3.1	Design Experimental . . . . .	34
3.2	Análise dos Resultados . . . . .	36
3.2.1	Avaliação das métricas de similaridade . . . . .	36
3.2.2	Avaliação dos algoritmos de agrupamento . . . . .	37
<b>4</b>	<b>Resultados</b>	<b>39</b>
4.1	Análise das métricas de similaridade . . . . .	39
4.1.1	Avaliação da autoridade . . . . .	39
4.1.2	Avaliação da estabilidade . . . . .	47
4.2	Resultados dos algoritmos de agrupamento . . . . .	55
4.2.1	Autoridade . . . . .	56
4.2.2	Estabilidade . . . . .	59
<b>5</b>	<b>Discussão</b>	<b>63</b>
5.1	Sobre as métricas de similaridade . . . . .	63
5.2	Sobre os algoritmos de agrupamento . . . . .	64
5.3	Ameaças à validade . . . . .	65
<b>6</b>	<b>Considerações Finais</b>	<b>66</b>
6.1	Trabalhos Futuros . . . . .	68
	<b>Referências Bibliográficas</b>	<b>69</b>
<b>A</b>	<b>Modelos</b>	<b>72</b>
A.1	SweetHome3D . . . . .	73
A.2	Ant . . . . .	74
A.3	Lucene . . . . .	76
A.4	ArgoUML . . . . .	77

# Lista de Publicações

Silva, D. E. U., Bittencourt, R. A., Calumby, R. T. (2019). Evaluation of Clustering Similarity Metrics for Architecture Recovery of Evolving Software. **Submitted to MSR 2019 – International Conference on Mining Software Repositories.**

# Lista de Tabelas

2.1	Palavras reservadas do Java . . . . .	12
2.2	Similaridade entre duas entidades $E_1$ e $E_2$ . . . . .	15
2.3	Coefficientes de associação para cálculo de similaridade . . . . .	16
2.4	Métricas de distância para cálculo de dissimilaridade . . . . .	16
2.5	Matriz de confusão para o cálculo da pureza . . . . .	27
3.1	Entidades de código e relações de dependência extraídos pela <i>Design Suite</i> [Bittencourt et al. 2009] . . . . .	33
3.2	Algoritmos para análise . . . . .	35
3.3	Métricas de similaridade de agrupamentos aplicadas na avaliação . . .	36
3.4	Sistemas alvo com <i>ground truth architectures</i> . . . . .	36
4.1	Autoridade relativa baseada em MeCl . . . . .	59
4.2	Autoridade absoluta baseada em MeCl . . . . .	59
4.3	Estabilidade relativa baseada em MeCl . . . . .	62
4.4	Estabilidade absoluta baseada em MeCl . . . . .	62

# Lista de Figuras

1.1	Grafo de dependências de um sistema de arquivos em C++ [Mancoridis et al. 1998] . . . . .	3
1.2	Grafo de organização de alto nível de um sistema de arquivos em C++ [Mancoridis et al. 1998] . . . . .	3
1.3	Arquitetura recuperada do <i>Linux</i> [Bowman et al. 1999] . . . . .	4
2.1	Características da recuperação arquitetural [Pollet et al. 2007] . . . . .	8
2.2	Sistema fictício . . . . .	10
2.3	Exemplo de <i>tokenização</i> . . . . .	12
2.4	Exemplo de Dendrograma [Anquetil e Lethbridge 1999] . . . . .	17
2.5	Dimensão de autoridade . . . . .	19
2.6	Dimensão de estabilidade . . . . .	20
2.7	Exemplo de cálculo de MoJo . . . . .	21
2.8	Grafo e dois possíveis agrupamentos. Adaptado de [Mitchell e Mancoridis 2001a] . . . . .	22
2.9	Conjuntos intracluster e intercluster [Mitchell e Mancoridis 2001a]. . . . .	23
2.10	Calculando MeCl [Mitchell e Mancoridis 2001a] . . . . .	24
2.11	Cálculo da B-Cubed <i>Precision-Recall</i> para um item [Amigó et al. 2009] . . . . .	26
3.1	Design experimental do estudo . . . . .	35
3.2	Comparação através da métrica absoluta HML . . . . .	37
4.1	Distribuição dos valores de autoridade e silhueta por sistema . . . . .	42
4.2	Distribuição dos valores de autoridade por métrica . . . . .	45
4.3	Correlações médias com a silhueta . . . . .	46
4.4	Distribuição dos valores de estabilidade por sistema . . . . .	49
4.5	Distribuição dos valores de estabilidade por métrica . . . . .	52
4.6	Correlações de estabilidade com delta de LOC . . . . .	54
4.7	Correlações de estabilidade com delta de classes . . . . .	55
4.8	Autoridade dos algoritmos de agrupamento em relação aos sistemas estudados . . . . .	58
4.9	Estabilidade dos algoritmos de agrupamento em relação aos sistemas estudados . . . . .	61

# Lista de Abreviações

<b>Abreviação</b>	<b>Descrição</b>
SL	Single Linkage
CL	Complete Linkage
MojoSim	Mojo Similarity
EdgeSim	Edge Similarity
MeCl	Merge Cluster Similarity
tf	Term Frequency
idf	Inverse Document Frequency

# Capítulo 1

## Introdução

A maioria dos sistemas de software úteis precisam ser alterados ao longo do tempo, tanto para adicionar novas funcionalidades como para corrigir falhas. Muitas vezes, sistemas mais antigos ou legados podem ser modificados através de medidas emergenciais para manter as aplicações funcionando. Neste caso, é comum fazer a manutenção dos sistemas sem o entendimento completo da estrutura e organização do software. Tais modificações, por menores que sejam, podem alterar o funcionamento de outros componentes do software, comprometendo o seu funcionamento, resultando em reaparecimento de *bugs* antigos ou na criação de novos. A estrutura do sistema pode deteriorar até o ponto em que a organização do código-fonte se torne tão caótica que o software necessite ser radicalmente reformulado ou abandonado [Mancoridis et al. 1998].

Porém, segundo Wiggerts (1997), reescrever todo o sistema, na maioria das vezes, não é uma solução viável. Por essa razão, os engenheiros de software dependem de técnicas e ferramentas a fim de ajudá-los a lidar com a complexidade dos grandes sistemas de software [Mancoridis et al. 1998]. Dependendo do tamanho do sistema, a tarefa de recuperar sua estrutura original pode ser muito árdua. Por isso, várias abordagens e técnicas vêm sendo propostas na literatura para auxiliar na recuperação arquitetural de software [Pollet et al. 2007].

Na engenharia de software, as visões arquiteturais permitem que os sistemas sejam visualizados de diferentes perspectivas. Três tipos de visões arquiteturais se destacam na literatura: componente-e-conector, de alocação e modular [Bass et al. 2003]. A visão componente-e-conector visa analisar os elementos arquiteturais existentes em tempo de execução. Já a visão de alocação objetiva em mostrar as relações entre os elementos do sistema com relação ao ambiente externo onde este é criado e executado. Por fim, a visão modular é formada por elementos denominados módulos, isto é, áreas do software com responsabilidades funcionais distintas, e por dependências entre os módulos.

A recuperação arquitetural de visões modulares possui um papel muito importante, uma vez que busca fornecer respostas para as perguntas dos arquitetos de software



sobre como decompor um sistema existente em subsistemas menores e como recuperar a estrutura original de um sistema legado ou até mesmo como comparar os modelos documentados com o código-fonte para o aperfeiçoamento do sistema.

As técnicas de recuperação arquitetural são classificadas em três tipos: quase-manuais; semi-automáticas e quase-automáticas [Pollet et al. 2007]. Nas técnicas quase-manuais, o engenheiro de software identifica manualmente os elementos arquiteturais com o auxílio de uma ferramenta. Nas técnicas semi-automáticas, o engenheiro de software instrui a ferramenta com o intuito de identificar elementos arquiteturais. Por fim, nas técnicas quase-automáticas, as ferramentas identificam os elementos arquiteturais através de algoritmos próprios.

No contexto da recuperação arquitetural de visões modulares de *software*, existem três tipos de técnicas quase-automáticas: análise de Conceitos, análise de dominância, e algoritmos de agrupamento. A análise de conceitos visa elaborar uma hierarquia de conceitos a partir de um conjunto de objetos e suas propriedades. A análise de dominância visa identificar nós dominantes, assim como em um grafo, com o propósito de detectar partes relacionadas de um sistema. Os algoritmos de agrupamento são técnicas quase-automáticas que procuram identificar grupos de entidades semelhantes a partir de suas características. De maneira geral, pode-se dizer que o agrupamento de entidades em vários grupos, ou módulos, a partir de uma relação de semelhança entre elas é uma forma de modularizar o sistema. Estes grupos formados por tais algoritmos de agrupamento, podem ser também chamados de *clusters*.

Um *cluster* define uma coleção de entidades agrupadas a partir de algum critério. Como consequência disso, Wiggerts (1997) também afirma que cada algoritmo de agrupamento (ou *clustering*) aplicado produz diferentes respostas, isto é, impõe, como resultado, uma estrutura ao software ao invés de recuperar uma estrutura existente.

Para sistemas menores, é possível extrair manualmente e visualizar as entidades e suas relações através de um grafo de dependências, como na Figura 1.1, onde os vértices são as entidades de software (e.g., arquivos, classes ou funções) e as arestas representam as relações de dependência entre as entidades (e.g., chamadas de métodos, uso de atributos).

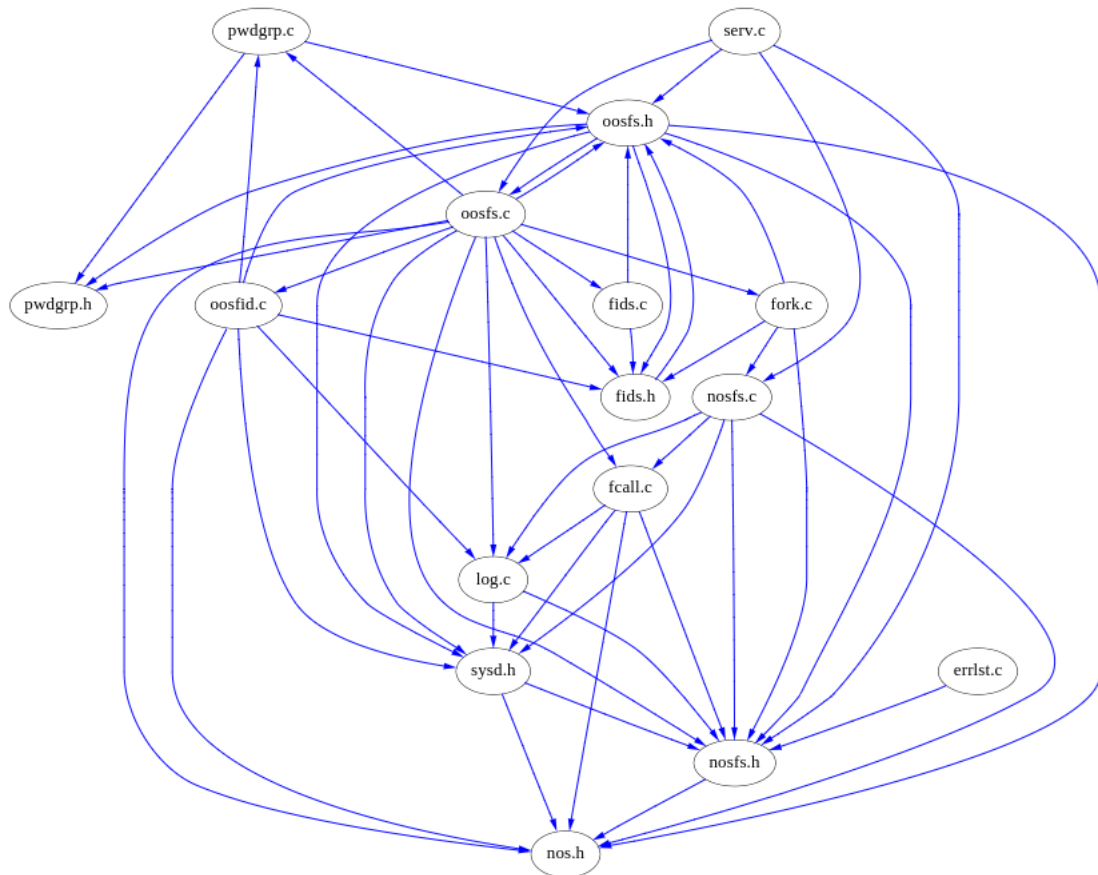


Figura 1.1: Grafo de dependências de um sistema de arquivos em C++ [Mancoridis et al. 1998]

Um possível resultado da aplicação de uma das técnicas de agrupamento pode ser visto na Figura 1.2. Neste caso, foram gerados seis grupos, dois maiores e dois menores dentro de cada um dos dois maiores.

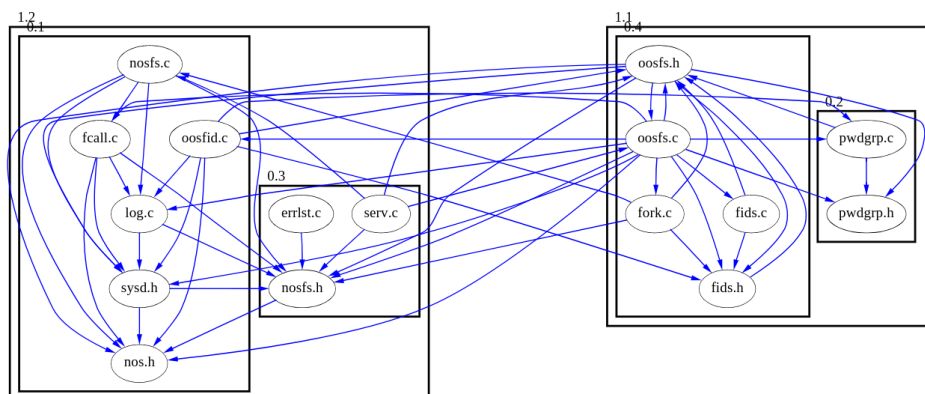


Figura 1.2: Grafo de organização de alto nível de um sistema de arquivos em C++ [Mancoridis et al. 1998]

Para sistemas maiores, a extração manual das entidades e sua visualização pode se tornar inviável devido à quantidade de nós e dependências que aumenta consideravelmente. A Figura 1.3 ilustra um agrupamento resultante do sistema *Linux* obtido por Bowman et al. (1999). Pode-se verificar que o sistema é dividido em partes (e.g., sistema de arquivos, interface de rede, bibliotecas) relativamente independentes e desacopladas com o intuito de entender melhor o funcionamento do sistema como um todo. Estas partes ou módulos foram definidos a partir da análise das dependências presentes no código-fonte.

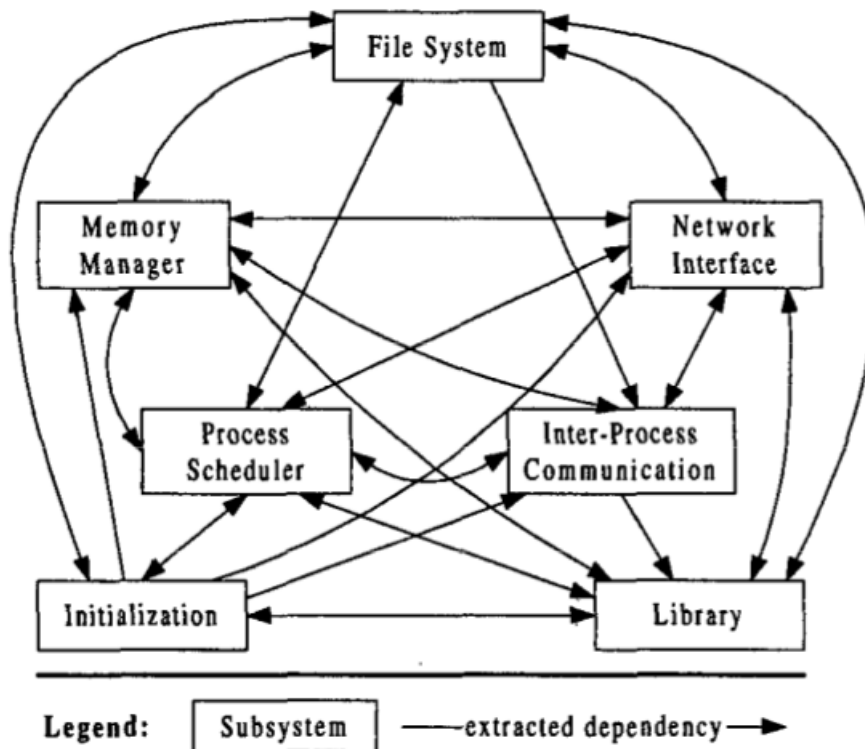


Figura 1.3: Arquitetura recuperada do *Linux* [Bowman et al. 1999]

Diversas ferramentas já foram elaboradas para auxiliar na extração automática destas entidades e suas dependências como o *Chava Reverse* [Korn e Koutsofios 1999], *Design Suite* [Bittencourt et al. 2009] e *Analizo* [Terceiro et al. 2010]. Dentre estas, a *Design Suite* realiza a extração das entidades e relações de sistemas em Java, além de permitir implementar e aplicar algoritmos de agrupamento para recuperação arquitetural.

De modo geral, os resultados dos algoritmos de agrupamento no contexto de recuperação arquitetural podem ser avaliados através de métricas, em especial, por métricas que revelem quão similares os resultados de um algoritmo são em relação a um agrupamento de referência produzido manualmente por arquitetos de software de um dado sistema em análise.

Trabalhos mais antigos como o de Wu et al. (2005) extraíram versões mensais de alguns sistemas de código aberto e aplicaram diferentes algoritmos de agrupamento nessas versões, definindo as métricas de estabilidade e autoridade, baseadas na métrica de similaridade MoJo, além da não-extremidade, para qualificar os resultados dos algoritmos. De maneira informal, a estabilidade indica que, quando um sistema sofre mudanças, os agrupamentos gerados devem refletir em alto nível estas mudanças. Já a autoridade avalia o quanto o agrupamento resultante se aproxima de um agrupamento criado por um arquiteto de software. Finalmente, a não-extremidade avalia se os agrupamentos resultantes possuem *clusters* indesejados, ou seja, muito grandes ou muito pequenos. Infelizmente, este trabalho utiliza apenas visões arquiteturais de alocação de arquivos em diretórios de código-fonte como modelo de autoridade e não um modelo de referência dos módulos do sistema explicitamente definido pelos arquitetos. Trabalhos mais recentes como os de Garcia et al. (2013) e Lutellier et al. (2015) avaliaram vários algoritmos a partir de cinco sistemas de código aberto com visões arquiteturais definidas por arquitetos na perspectiva da comparação com modelos de referência e com quatro métricas de similaridade distintas.

Apesar de os trabalhos anteriores representarem um avanço no processo de avaliação de técnicas de agrupamento, é necessário ir além da avaliação dos algoritmos de agrupamento, avaliando as próprias métricas de similaridade usadas, já que não existe consenso sobre qual delas é mais adequada. A partir desta avaliação destas métricas é, então, possível avaliar melhor os algoritmos de agrupamento a partir da comparação com os modelos de referência produzidos por arquitetos de software.

## 1.1 Objetivos

O objetivo deste estudo foi avaliar um conjunto de métricas de similaridade de agrupamentos usadas para avaliar técnicas de agrupamento usadas na recuperação arquitetural de visões modulares. Esta avaliação foi realizada a partir das dimensões de estabilidade e autoridade, utilizando modelos arquiteturais de referência e métricas baseadas tanto no posicionamento das entidades nos *clusters* como no acoplamento gerado entre os *clusters*. Além disso, medidas intrínsecas de qualidade de agrupamentos e de variabilidade de *software* em evolução também são usadas no processo de avaliação para melhor avaliar as métricas.

Os objetivos específicos deste trabalho são:

1. Extrair versões de um conjunto de sistemas de software disponíveis em repositórios abertos de modo a formar uma base de análise;
2. Implementar um conjunto de algoritmos de agrupamento aglomerativos em uma suíte de ferramentas de recuperação arquitetural;
3. Implementar métricas de similaridade de agrupamentos em uma suíte de ferramentas de recuperação arquitetural;

4. Executar experimentos aplicando os algoritmos de agrupamento implementados às versões dos sistemas alvo;
5. Avaliar as métricas de similaridade de agrupamentos a partir dos dados gerados pelos algoritmos de agrupamento;
6. Avaliar, comparativamente, os algoritmos de agrupamento aglomerativos a partir da(s) melhor(es) métrica(s) implementada(s).

## 1.2 Contribuições

As principais contribuições desta dissertação são:

1. Extensão da *Design Suite*, uma suíte de ferramentas de recuperação arquitetural, com a implementação de algoritmos de agrupamento aglomerativos e um conjunto de métricas de similaridade de agrupamentos;
2. Avaliação das métricas de similaridade de agrupamentos no contexto de recuperação arquitetural de software;
3. Avaliação de um conjunto de algoritmos de agrupamento aglomerativos usados como técnicas quase-automáticas para recuperação arquitetural de software.

## 1.3 Organização do Trabalho

Este documento está organizado como a seguir. Este capítulo apresenta uma introdução do trabalho, o Capítulo 2 apresenta os fundamentos teóricos e os trabalhos relacionados, o Capítulo 3 descreve a metodologia empregada, o Capítulo 4, os resultados obtidos, o Capítulo 5 apresenta uma discussão sobre os resultados e, finalmente, o Capítulo 6 encerra a dissertação com as conclusões.

# Capítulo 2

## Fundamentação Teórica

Nesta seção, são abordados os conceitos principais sobre as técnicas de recuperação arquitetural, avaliação da recuperação arquitetural e métricas de similaridade de agrupamentos.

### 2.1 Recuperação Arquitetural

A recuperação arquitetural de software (do inglês *Software Architecture Reconstruction* - SAR) possui diversos aspectos a serem considerados segundo Pollet et al. (2007). A figura 2.1 apresenta graficamente estas particularidades. Primeiramente, para fazer uma recuperação arquitetural, deve-se ter em mente quais são os seus objetivos. Alguns exemplos de objetivos são a redocumentação para atualização de documentos desatualizados ou o acompanhamento da evolução com checagem de conformidade entre a implementação e o design. Definido os objetivos, o processo pode se dar de um nível mais alto para níveis mais baixos (*top-down*), do nível mais baixo para o mais alto (*bottom-up*) ou uma mistura das duas abordagens. Geralmente suas entradas definem o tipo de processo utilizado. Ao utilizar código-fonte como entrada, conseqüentemente se caracteriza por um processo *bottom-up*. Ao utilizar informação histórica, a abordagem pode mudar para *top-down* ou será híbrida se utilizar as duas entradas. Para processar as entradas, é necessário aplicar técnicas apropriadas. Elas podem ser quase-manuais, semi-automáticas ou quase-automáticas. As quase-manuais são totalmente realizadas pelos profissionais de forma manual sem (ou com quase nenhuma) automatização durante o processo. As técnicas semi-automáticas já pressupõem alguma ferramenta de auxílio ao especialista durante o processo do agrupamento. As técnicas quase-automáticas visam aplicar critérios próprios para definir os agrupamentos a serem gerados sem interferência do especialista. Ao aplicar as técnicas nas entradas, são geradas saídas. As saídas podem ser visualizações arquiteturais resultantes do processo ou checagens de conformidade entre a arquitetura planejada e a implementada, por exemplo.

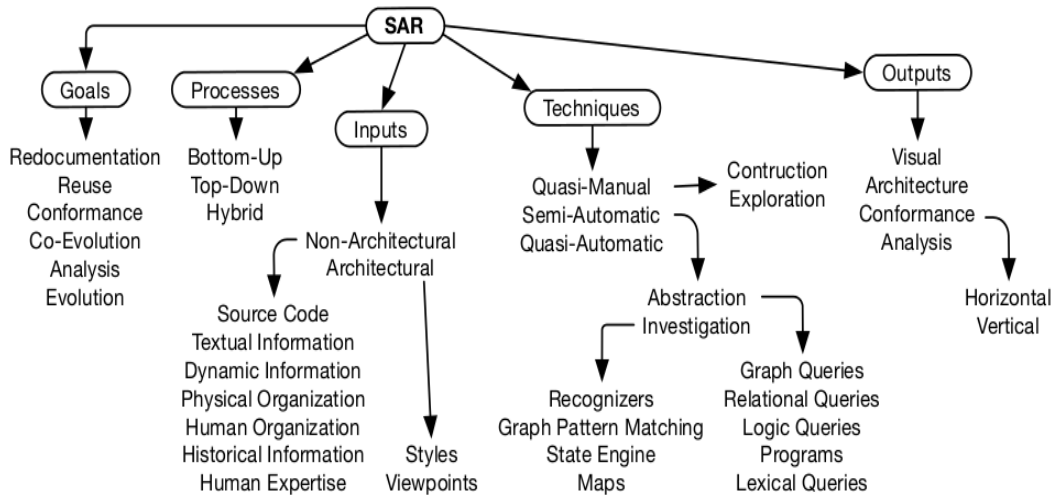


Figura 2.1: Características da recuperação arquitetural [Pollet et al. 2007]

O foco deste trabalho é **analisar** técnicas *bottom-up* de agrupamento quase-automático que utilizam código-fonte como entrada e que produzem *clusters* de elementos de código como saídas, ou seja, *clusters* organizados como visões arquiteturais modulares. A seguir, estas técnicas automatizadas serão descritas em mais detalhes.

## 2.2 Técnicas de recuperação arquitetural de visões modulares

Como dito anteriormente, as técnicas de recuperação arquitetural podem ser de três tipos: quase-manuais, semi-automáticas ou quase-automáticas. Com o aumento do tamanho dos sistemas devido à alta demanda por software em diversas áreas do conhecimento, uma recuperação arquitetural quase-manual ou até semi-automática pode se tornar inviável. As técnicas quase-automáticas possuem um grande potencial para auxiliar os profissionais em diversas tarefas do dia-a-dia.

Para melhor compreensão, Bass et al. (2003) fazem uma analogia muito interessante entre os sistemas de software e a anatomia humana. Assim como um sistema de computação, o corpo humano é formado por diferentes sistemas (e.g., circulatório, nervoso, muscular). Além disso, os profissionais de saúde enxergam o corpo humano através de diferentes visões. Por exemplo, um cardiologista possui uma visão mais preocupada com o sistema circulatório, o que é diferente do neurologista, mais preocupado com o sistema nervoso. Logo, um diagnóstico completo de um paciente pode ser feito a partir da análise de cada visão conhecida.

Assim como as visões do corpo humano expressas pelos especialistas médicos, a arquitetura de software também oferece diversas visões, cada uma abordando diferentes elementos do software. Ao uni-las todas, o sistema estará bem documentado arquiteturalmente e todas as suas propriedades e relações de alto nível estarão claras. Segundo Bass et al. (2003), as visões de software são de três tipos:

- Componente-e-conector;
- Alocação;
- Modular.

Na visão componente-e-conector, os elementos são componentes e conexões em tempo de execução. Deste modo, esta visão busca entender melhor a interação entre os processos em execução, a concorrência e os dados compartilhados entre eles.

A visão de alocação se preocupa com a relação dos componentes de software com o meio externo onde o sistema é criado e executado. O componente pode estar relacionado tanto a um elemento de *hardware* quanto a uma equipe de desenvolvedores que irá implementá-lo.

A visão modular é constituída de elementos ou entidades de código do sistema (e.g., arquivos, classes, pacotes, métodos) e as relações são as dependências estruturais estáticas entre estes elementos (e.g., uso, chamada, herança).

As visões modulares costumam ser mais utilizadas na recuperação arquitetural de software devido à fácil disponibilidade de código-fonte, principalmente em projetos *open-source*. Esta disponibilidade facilita o estudo de uma abordagem *bottom-up* de recuperação arquitetural, usando o código-fonte como entrada para o processo.

As técnicas de agrupamento podem ser utilizadas para recuperar visões modulares de alto nível automaticamente a partir da análise dos elementos de código e seus relacionamentos, agrupando-os em *clusters*. Como afirmado anteriormente, os *clusters* ou grupos são conjuntos de entidades de código associados através de regras. No caso dos algoritmos de agrupamento, estas regras são os critérios utilizados durante o processo de agrupamento automático para determinar os *clusters* a serem formados.

## 2.3 Representação de entidades e suas relações para a recuperação arquitetural

As técnicas de *clustering* definem critérios próprios para o agrupamento das entidades. Estes critérios são baseados, muitas vezes, nas similaridades entre as entidades de código. As similaridades podem ser calculadas de diversas formas.



Devido à maior disponibilidade e facilidade de acesso, diversos estudos anteriores extraem as entidades de baixo nível diretamente do código-fonte dos sistemas para fazer a recuperação arquitetural. Através do código-fonte, é possível extrair seus elementos essenciais, tais como arquivos, classes, interfaces, métodos, atributos, funções, entre outras possibilidades.

É possível descrever as entidades de código de diferentes maneiras, seja por suas características intrínsecas ou por suas relações com outras entidades de código. No caso das características intrínsecas, pode-se usar, por exemplo, uma estratégia de recuperação de informação, descrevendo as entidades pela frequência dos termos específicos do vocabulário do texto particular da entidade (e.g., termos que fazem parte dos identificadores de um método ou mesmo comentários no interior de um método). No caso de descrição pela relações, cada entidade carrega um vetor de características onde se armazena a força das ligações que aquela entidade possui com todas as outras do software. Tais ligações entre entidades ocorrem de diversas formas, tais como como herança, implementação de interface, chamada de métodos, uso de tipos, retorno de tipos, entre outros. Através dos vetores de características das entidades de código, é possível definir critérios para agrupá-las.

Se a representação de entidades é através de dependências estruturais, elas podem ser agrupadas a partir das dependências diretas entre duas dadas entidades (ligação direta) ou a partir de entidades com dependências similares (e.g., dois métodos que, no seu corpo, chamam um mesmo terceiro método, também chamado de ligação irmã). No exemplo da Figura 2.2, a classe **F10** herda da classe **Carro** assim como a classe **Roadster** herda da classe **Motocicleta**. As classes **Carro** e **Volante** podem ser agrupadas a partir da dependências entre elas (C1) ou as classes **Motocicleta** e **Carro** podem ser agrupadas juntas por ambas utilizarem a Classe **Roda** (C2).

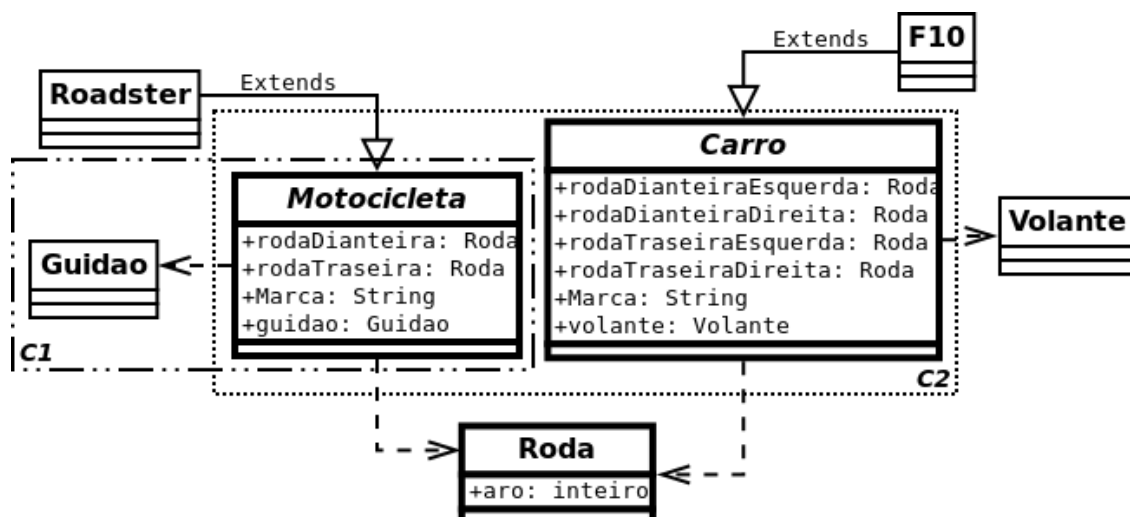


Figura 2.2: Sistema fictício

Outra opção é tratar o código-fonte como um conjunto de documentos textuais

utilizando técnicas de recuperação de informação. A Seção 2.4 irá tratar com mais detalhes de como a recuperação de informação é utilizada para extrair as entidades de código e como os vetores de características são estruturados.

## 2.4 Recuperação de Informação

A Recuperação de Informação (RI) procura material de natureza não-estruturada que satisfaça a uma necessidade de informação a partir de largas coleções. Geralmente, este material é um conjunto de documentos em forma de texto armazenados em computadores [Manning et al. 2008]. No caso da engenharia de software, a recuperação de informação é muito utilizada para a extração dos vocabulários de software dos sistemas-alvo.

Para a extração do vocabulário de software através de RI, realizam-se duas etapas prévias:

1. Coletar os arquivos de código-fonte do sistema-alvo;
2. Realizar pre-processamento linguístico na coleção de códigos-fonte do sistema alvo

Nesse caso, os documentos são arquivos contendo o código-fonte dos sistemas e o pré-processamento linguístico trata a sequência de caracteres a partir de quatro etapas: *Tokenização*, remoção de *stop words*, normalização e *stemming*. Ao fim da execução dessas quatro etapas, obtém-se o vocabulário do software, ou seja, o conjunto de termos utilizados para construir o sistema.

### 2.4.1 Extração de vocabulário de software

A seguir, são descritas as etapas de *Tokenização*, remoção de *stop words*, normalização e *stemming*.

#### *Tokenização*

Dada uma sequência de caracteres e uma unidade de documento definida, a *tokenização* é a tarefa de quebrá-lo em pedaços chamados *tokens* e talvez retirar outros como pontuação [Manning et al. 2008]. Um *token* é uma instância de uma sequência de caracteres de um documento particular que são agrupados como uma útil unidade semântica para serem processada. A Figura 2.3 ilustra a *tokenização* da assinatura de um método.

```
public void setNome(String nome)
Tokens: public void setNome (String nome)
```

Figura 2.3: Exemplo de *tokenização*

O conjunto de *tokens* é transformado em termos ou palavras a partir da atribuição de tipos para cada *token* reconhecido. Desta forma, cada *token* está incluído em, exclusivamente, um tipo. A classificação dos *tokens* é importante pois é a forma mais simples de diferenciar um do outro.

### Remoção de *Stop words*

Às vezes, algumas palavras extremamente comuns são provavelmente de pouco valor com relação ao auxílio na seleção de documentos correspondentes à necessidade do usuário. Essas palavras são chamadas de *stop words* [Manning et al. 2008].

A estratégia mais empregada para determinar uma lista de *stop words* é ordenar os termos pela quantidade de vezes que aparecem em toda a coleção de documentos e, por fim, retirar as palavras mais frequentes. Duas *stop words* muito conhecidas na linguagem Java são as expressões *get* e *set*, pois são muito utilizadas e não são muito úteis para diferenciar um documento do outro. Além do *get* e *set* existem as palavras reservadas da linguagem que obrigatoriamente são *stop words*. A Tabela 2.1 apresenta uma lista das palavras reservadas do Java.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert					

Tabela 2.1: Palavras reservadas do Java

### Normalização

Segundo Manning et al. (2008), existem casos em que duas sequências de caracteres não são exatamente iguais mas é desejável que sejam correspondentes. Desta

forma, os *tokens* passam por uma etapa de equivalência, onde se eliminam diferenças superficiais entre eles.

## Stemming

As linguagens permitem o uso das palavras em diferentes flexões, como o uso de plural, gerúndio, presente e passado. Aliado a isso, existem famílias de palavras que são derivadas de uma outra e com significados similares como *Control* e *Controller* muito utilizados em modelos *MVC* de programação. A partir daí, os radicais são extraídos das palavras com o intuito de fazê-las equivalentes quando buscadas. Logo, ao se fazer uma consulta por *Control*, *Controller* também será retornado ao usuário.

### 2.4.2 Representação vetorial das entidades de código

Com o aumento do tamanho das coleções, o número de documentos correspondentes pode exceder a quantidade que um ser humano poderia examinar [Manning et al. 2008]. Logo, torna-se necessário pontuar e ordenar os termos e documentos. Os termos são contados em cada documento e sua importância é então calculada baseada nas estatísticas de seu aparecimento no documento. Por fim, cada documento é tratado como um vetor onde cada posição indica o peso de cada termo. Esse modelo de representação é conhecido como modelo de espaço vetorial (*Vector Space Model*). Dessa forma cada vetor possui dimensão igual a quantidade de termos no documento que está sendo representado.

Para que o modelo de espaço vetorial seja construído, deve-se montar o vetor de pesos de cada documento de acordo com a coleção a ser pesquisada. A abordagem mais simples é associar o peso do termo à quantidade de vezes em que apareceu no documento. Esta forma de pontuar é chamada de frequência dos termos (*term frequency*) e é denotado por  $tf_{t,d}$  onde  $t$  é o termo e  $d$  é o documento [Manning et al. 2008].

Porém, quando termos se repetem muito na coleção, é necessário adotar outra abordagem para contagem, pois esses termos repetitivos acabam perdendo relevância. Uma segunda abordagem é contar a quantidade de documentos que contenham um termo  $t$ , chamada de frequência em documento (*document frequency*) e definida como  $df_t$ . A partir daí, é definida a frequência inversa do documento (*inverse document frequency*) denotada por  $idf$ , onde:

$$idf_t = \log \frac{N}{df_t} \quad (2.1)$$

onde  $N$  é o número de documentos da coleção e o logaritmo é na base 10.

Assim, quando o termo for muito frequente, o seu  $idf$  será baixo e quando o termo for muito raro, o seu  $idf$  será alto.

Combinando as duas abordagens tem-se a composição de pesos para cada termo em cada documento [Manning et al. 2008]. Assim, *tf-idf* é definido como a multiplicação do valor retornado pelo *tf* e pelo *idf* como na equação 2.2:

$$tf-idf_{t,d} = tf_{t,d} \cdot idf_t \quad (2.2)$$

O *tf-idf* atribui ao termo  $t$  um peso no documento  $d$  que assume valores mais altos quando  $t$  ocorre muitas vezes em um conjunto de documentos e valores menores quando  $t$  ocorre poucas vezes em um ou em diversos documentos. Os valores de *tf-idf* são ainda menores quando o termo ocorre em todos os documentos.

Após o cálculo da pontuação, seja com *tf*, *idf* ou *tf-idf*, o vetor de pesos de um documento pode ser montado. Mas como saber se dois documentos são similares ou diferentes entre si? Tratando-se de dois vetores, pode-se calcular a similaridade entre eles a partir da computação da distância entre eles. Uma das formas de se computar a similaridade entre dois vetores é o cosseno do ângulo entre os vetores, que pode ser calculado pelo produto escalar entre os vetores dividido pelo produto das normas euclidianas de cada vetor, ou seja:

$$sim(d1, d2) = \frac{|\vec{V}(d1)| \cdot |\vec{V}(d2)|}{|\vec{V}(d1)| |\vec{V}(d2)|} \quad (2.3)$$

onde o produto escalar é dado por  $\sum_{i=1}^M x_i y_i$  e a norma euclidiana, por  $\sqrt{\sum_{i=1}^M \vec{V}_i^2(d)}$ .

O conjunto desses vetores de pesos de cada documento formam uma matriz de tamanho  $m \times n$ , onde  $m$  é o tamanho do vocabulário do conjunto de documentos e  $n$  é a quantidade de documentos na coleção, que é chamada de matriz termo-documento (*term-by-document matrix*) [De Lucia et al. 2012].

Por outro lado, existem dois problemas clássicos na área de RI que são intrínsecos à linguagem natural: A sinonímia e a polissemia [Manning et al. 2008]. A sinonímia é o fenômeno de palavras diferentes que significam a mesma coisa. Já a polissemia indica que uma única palavra pode trazer consigo diversos significados.

A técnica de Indexação por Semântica Latente (*LSI*, do inglês *Latent Semantic Indexing*), reduz o espaço de conceitos decompondo uma matriz termo-documento em:

$$A = T \cdot S \cdot D^T \quad (2.4)$$

onde  $A$  é a matriz termo-documento,  $T$  é a matriz termo-conceito,  $D$  é a matriz documento-conceito e  $S$  é a matriz diagonal com os autovalores dos conceitos [De Lucia et al. 2012].

Dessa forma, *LSI* tenta identificar palavras sinônimas e polissêmicas, diminuindo o tamanho da matriz termo-documento e, em consequência, diminuindo também o processamento sobre esta matriz agora reduzida. É comum usar *LSI* para reduzir o tempo de processamento em consultas de recuperação de informação.

## 2.5 Similaridade entre entidades

Uma vez definida a representação das entidades do código-fonte, é necessário definir um método para agrupamento das entidades. Para que os algoritmos de agrupamento consigam classificar as entidades em *clusters*, é preciso conhecer a similaridade entre elas a partir de suas características.

A determinação da similaridade baseada em dependências estruturais entre duas entidades de software pode ser obtida através de duas abordagens: ligação direta (*direct link*) ou ligação irmã (*sibling link*) [Maqbool e Babri 2004]. A ligação direta analisa a similaridade através da força da dependência entre duas entidades. Analogamente a um grafo, se as entidades forem os vértices e as relações forem as arestas, a força da relação é determinada pela quantidade de arestas conectando as duas entidades. Já a ligação irmã é baseada nas características compartilhadas. Os vértices continuam sendo as entidades, porém, as relações são as características dos nós que eles conectam. A similaridade é então medida através da dependência dos recursos em comum. Segundo Maqbool e Babri (2004), a representação com a ligação irmã permite a aplicação das técnicas de agrupamento na análise de software, ao contrário da ligação direta pois se duas classes chamam uma mesma função, então elas possuem algo em comum, tornando inclusive o resultado final mais compreensível.

No contexto dos algoritmos de agrupamento, existem três tipos de métricas de similaridade entre entidades: coeficientes de associação, medidas de distância e coeficientes de correlação [Maqbool e Babri 2007]. Os coeficientes de associação são aplicados para calcular a semelhança quando as características são binárias, ou seja, 0 para ausência e 1 para presença da característica. A similaridade entre duas entidades  $E_1$  e  $E_2$  pode ser ilustrado através da Tabela 2.2.

		$E_2$	
		<b>1</b>	<b>0</b>
$E_1$	<b>1</b>	a	b
	<b>0</b>	c	d

Tabela 2.2: Similaridade entre duas entidades  $E_1$  e  $E_2$

Na tabela 2.2 estão presentes quatro valores:  $a$ ,  $b$ ,  $c$  e  $d$ , onde  $a$  equivale à quantidade de características presentes em ambas as entidades,  $b$  simboliza o total de características presentes em  $E_1$  mas ausentes em  $E_2$ ,  $c$  representa as características presentes

em  $E_2$  mas ausentes em  $E_1$  e, por fim,  $d$  representa as características ausentes em ambas as entidades. É bastante comum na área de software a variável  $d$  ser muito maior do que  $a$ ,  $b$  e  $c$  visto que os vetores de características de cada entidade se espalham ao longo de diversas áreas do código-fonte. A Tabela 2.3 ilustra alguns coeficientes de associação mais conhecidos na literatura.

Tabela 2.3: Coeficientes de associação para cálculo de similaridade

Coeficiente de Jaccard	$J = a/(a + b + c)$
Coeficiente Simples	$S = (a + d)/(a + b + c + d)$
Coeficiente de Sorensen-Dice	$SD = 2a/(2a + b + c)$

As métricas de distância calculam a dissimilaridade entre duas entidades. Quanto maior for a distância, menor a similaridade. A Tabela 2.4 lista algumas medidas de distâncias mais conhecidas. O uso de distância pode ocasionar em cálculos errôneos de similaridade pois se duas entidades forem completamente distintas, a distância poderá ser zero principalmente em características binárias onde a distância Euclidiana é simplificada para  $\sqrt{b + c}$  e a distancia Canberra reduz para  $b + c$ . Neste caso, duas entidades podem ter uma distância zero não por serem próximas, e sim, por serem totalmente diferentes, o que pode resultar em agrupamentos indesejáveis. A Tabela 2.4 ilustra algumas métricas de distância mais conhecidas na literatura.

Tabela 2.4: Métricas de distância para cálculo de dissimilaridade

Distância Euclidiana	$D(X, Y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$
Distância Canberra	$C(X, Y) = \sum_{i=1}^n  x_i - y_i  / ( x_i  +  y_i )$
Distância Minkowski	$M(X, Y) = (\sum_{i=1}^n  x_i - y_i ^r)^{1/r}$

Os coeficientes de correlação são utilizados para correlacionar as características de duas entidades. O coeficiente de correlação de Pearson é bastante conhecido e é dado por:

$$\rho = \frac{\sum_{i=1}^n (x_i - \bar{x}_i)(y_i - \bar{y}_i)}{\sqrt{\sum_{i=1}^n (x_i - \bar{x}_i)^2 \sum_{i=1}^n (y_i - \bar{y}_i)^2}} \quad (2.5)$$

Para características binárias, a equação é simplificada para:

$$\rho = \frac{ad - bc}{\sqrt{(a + b)(c + d)(a + c)(b + d)}} \quad (2.6)$$

Como, geralmente, o  $d$  é normalmente muito maior do que  $a$ ,  $b$  e  $c$ , a equação é reduzida a um formato similar ao coeficiente de Jaccard:

$$\rho = \frac{a}{\sqrt{(a+b)(a+c)}} \quad (2.7)$$

Definidas as formas de representação das entidades e como calcular a similaridade ou distâncias entre elas, são apresentados a seguir alguns dos algoritmos de agrupamento conhecidos na literatura.

## 2.6 Algoritmos hierárquicos aglomerativos para recuperação arquitetural por *clustering*

Os algoritmos hierárquicos aglomerativos iniciam o processo de agrupamento com as entidades individuais, cada uma em um *cluster* próprio. Em cada passo, as entidades são agrupadas até que se forme um único grande *cluster*, contendo todas as entidades [Anquetil e Lethbridge 1999]. O algoritmo aglomerativo constrói uma estrutura particular de árvore a partir das folhas, chamada de dendrograma, onde as folhas são as entidades do sistema e o nó raiz é composto por todas as entidades juntas em um único cluster.

Assim, a fim de obter um agrupamento que faça sentido, deve-se definir um ponto de corte pertinente no dendrograma. Se o corte for feito na altura zero, serão obtidos *clusters* unitários. Se o corte for feito na altura máxima, todo o sistema será o único *cluster* [Anquetil e Lethbridge 1999]. A Figura 2.4 ilustra um dendrograma produzido por um algoritmo aglomerativo.

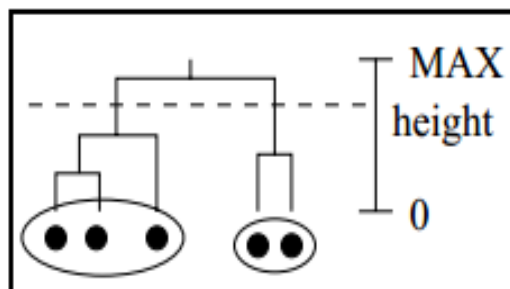


Figura 2.4: Exemplo de Dendrograma [Anquetil e Lethbridge 1999]

Partindo da altura zero, critérios de distância para decidir quais entidades serão agrupadas influenciam diretamente no resultado final do agrupamento, dado um ponto de corte previamente estabelecido. Para determinar a distância entre as entidades, são utilizados quatro métodos nos algoritmos aglomerativos mais típicos: *Single Linkage* (SL), *Complete Linkage* (CL), *Weighted Average Linkage* (WLA) e *Unweighted Average Linkage* (ULA) [Maqbool e Babri 2007]. Para melhor ilustrar as regras de cálculo de distância entre as entidades, considere as entidades  $E_0$ ,  $E_1$  e



$E_2$ . Considere também  $E_1 \cup E_2$  um novo *cluster* formado pelas entidades  $E_1$  e  $E_2$ . As equações 2.8, 2.9, 2.10 e 2.11 mostram o cálculo das distâncias de cada método. A partir do cálculo da distância entre os *clusters*, as entidades então são agrupadas de acordo com sua regra de distância.

$$SL : d = \min(d_1, d_2) \quad (2.8)$$

$$CL : d = \max(d_1, d_2) \quad (2.9)$$

$$WLA : d = \frac{1}{2}d_1 + \frac{1}{2}d_2 \quad (2.10)$$

$$ULA : d = \frac{d_1 \cdot \text{tamanho}(E_1) + d_2 \cdot \text{tamanho}(E_2)}{\text{tamanho}(E_1) + \text{tamanho}(E_2)} \quad (2.11)$$

onde  $d_1$  é a distância entre  $E_0$  e  $E_1$  e  $d_2$  é a distância entre  $E_0$  e  $E_2$ .

## 2.7 Métricas de avaliação da recuperação arquitetural

Com o crescimento do interesse pelas técnicas quase-automáticas de recuperação arquitetural, algumas delas, como as listadas na Seção 2.6, surgiram para auxiliar nesta tarefa. No entanto, é necessário verificar a eficácia destes algoritmos aplicados à recuperação arquitetural de visões modulares. Alguns trabalhos propuseram métricas com este objetivo e estas métricas são apresentadas a seguir, mas antes, é preciso tratar de algumas questões preliminares, como os modelos de referência arquiteturais e as dimensões de avaliação.

### 2.7.1 Modelos de referência ou *ground truth architectures*

A ideia do modelo de referência ou *ground truth architecture* é utilizar agrupamentos realizados pelos arquitetos do sistema como modelos de referência para comparação com os agrupamentos gerados por técnicas quase-automáticas de recuperação arquitetural. Estes modelos visam servir de parâmetro para mensurar quanto um algoritmo consegue reproduzir a organização de sistemas reais desenvolvidos por especialistas.

Koschke e Eisenbarth (2000) propuseram um *framework* para avaliação experimental das técnicas de agrupamento. Nesta abordagem, existem dois tipos de avaliação, os *benchmarks*, para as técnicas quase-automáticas, e o experimento controlado para as outras que possuem interferência humana durante o processo de geração dos agrupamentos (ou partições). No caso dos *benchmarks*, é necessário que haja um modelo de referência e fazer testes dos candidatos propostos pelos algoritmos com o

modelo de referência no intuito de verificar os pontos positivos e negativos de cada algoritmo. Para obter um modelo de referência, eles sugerem que é necessário fazer a validação por pelo menos dois indivíduos com os componentes encontrados em comum.

Mitchell e Mancoridis (2001a) desenvolveram uma ferramenta denominada CRAFT para automatizar a geração de modelos de referência entendendo que a geração manual de modelos de referência pode ser muito entediante apesar de os resultados tenderem a ser melhores devido à utilização do conhecimento dos próprios *designers*. Estes autores acrescentam que os modelos de referência também são úteis aos profissionais para adquirir um pouco mais de certeza em relação aos resultados produzidos pelas técnicas automáticas de agrupamento.

### 2.7.2 Dimensões de avaliação

O trabalho de Wu et al. (2005) propôs a avaliação dos algoritmos a partir de três dimensões de avaliação descritas a seguir: autoridade, estabilidade e não-extremidade.

**Autoridade.** Os agrupamentos gerados pelos algoritmos devem se assemelhar com alguma autoridade [Wu et al. 2005]. Um agrupamento oficial pode ser gerado por um arquiteto de software ou derivado da estrutura de diretórios do sistema alvo. Este agrupamento oficial nada mais é do que um modelo de referência citado na seção anterior. A Figura 2.5 ilustra a aplicação da autoridade sobre um conjunto de versões consecutivas de um mesmo sistema de software, onde  $C_1$  a  $C_n$  são os agrupamentos gerados pelos algoritmos e  $C_{1A}$  a  $C_{nA}$  são os modelos de referência das respectivas versões.

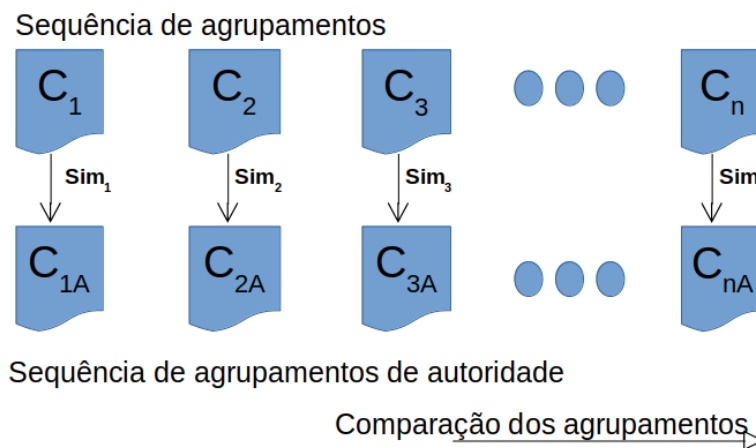


Figura 2.5: Dimensão de autoridade

**Estabilidade.** Agrupamentos similares devem ser produzidos por versões similares de um sistema de software [Wu et al. 2005]. Pequenas modificações no software

devem ser refletidas em pequenas mudanças nos agrupamentos gerados pelos algoritmos, assim como grandes modificações devem resultar em maiores mudanças também nos agrupamentos. Para a dimensão de estabilidade, costuma-se comparar os agrupamentos gerados por versões consecutivas do sistema alvo como ilustrado na figura 2.6.

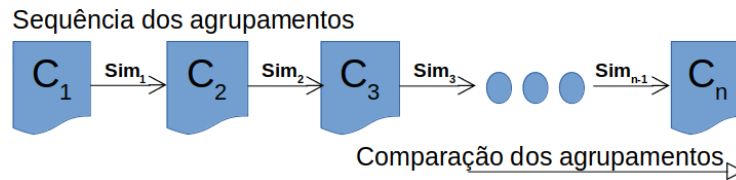


Figura 2.6: Dimensão de estabilidade

**Não-Extremidade.** Segundo Wu et al. (2005), um agrupamento deve evitar duas situações: Agrupar a maioria das entidades em um ou poucos grandes *clusters*, denominados de **buracos negros**, e evitar gerar *clusters* com somente um elemento, denominados de **nuvens de poeira**. Neste trabalho, não utilizaremos a dimensão de não-extremidade pois acreditamos que a geração de *clusters* muito pequenos ou muito grandes pode ser somente uma manifestação de uma característica intrínseca dos sistemas quando foram originalmente projetados. Dificilmente um algoritmo de agrupamento irá gerar grupos com quantidade balanceada de entidades caso o software apresente nós muito semelhantes ou muito diferentes uns dos outros. Dito isto, julgamos que a não-extremidade pode não acrescentar significativamente a este trabalho e, por isso, não será usada aqui como dimensão de avaliação dos algoritmos de agrupamento.

### 2.7.3 Métricas de similaridade de agrupamentos

O conjunto das métricas definidas na Seção 2.7.2 ajuda a traçar uma linha de trabalho para avaliar agrupamentos gerados pelos algoritmos de *clustering*. Contudo, todas elas partem da premissa do cálculo da similaridade entre dois agrupamentos. Nesta seção, são algumas das métricas conhecidas na literatura para o cálculo de similaridade entre dois dados agrupamentos.

Buscamos utilizar um conjunto de métricas representativas o suficiente para que o estudo fosse viável. Assim, usamos métricas de áreas e naturezas distintas o suficiente para conseguirmos uma grande quantidade de maneiras distintas de calcular similaridade entre os agrupamentos.

#### MoJo e MoJoSim

Tzerpos e Holt (1999) definiram uma métrica de dissimilaridade entre duas partições arquiteturais chamada de *MoJo*. Seu cálculo é baseado nas operações para

transformar um agrupamento em outro. *MoJo* permite basicamente duas operações: **move** e **join** [Tzerpos e Holt 1999]. **Move** implica retirar a entidade de um *cluster*, alocando-a em outro, e **Join** implica unir dois *clusters* existentes, decrementando o número de *clusters* em uma unidade. Assim, o valor de *MoJo* entre duas partições A e B pode ser calculado através da Equação 2.12 [Tzerpos e Holt 1999].

$$MoJo(A, B) = \min(mno(A, B), mno(B, A)) \quad (2.12)$$

onde  $mno(A, B)$  retorna o número mínimo de *moves* e *joins* para transformar o agrupamento A em B. O operador  $\min()$  deve ser aplicado devido ao *MoJo* não ser reflexivo, isto é, geralmente  $mno(A, B) \neq mno(B, A)$ . Para medir similaridade entre dois agrupamentos, define-se *MoJoSim* como na Equação 2.13 [Bittencourt et al. 2009].

$$MoJoSim(A, B) = 1 - \frac{MoJo(A, B)}{n} \quad (2.13)$$

onde  $n$  é o número de entidades a serem agrupadas. A Figura 2.7 ilustra um exemplo didático de cálculo do *MoJo*.

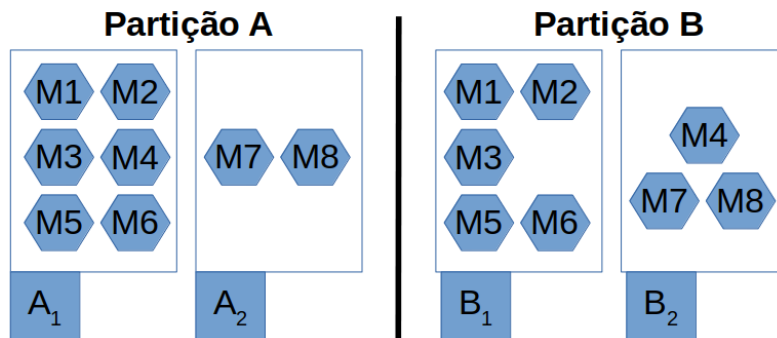


Figura 2.7: Exemplo de cálculo de *MoJo*

Da Figura 2.7, é possível perceber que executar uma operação de *move* do módulo *M4* do *cluster*  $A_1$  para o  $A_2$  é o suficiente para transformar o agrupamento A em B. Dessa forma o valor do *MoJo* seria igual a 1 e o *MoJoSim* seria  $1 - \frac{1}{8} = 0,875$ .

Contudo, com o aumento da quantidade de nós, torna-se cada vez mais difícil computar a menor quantidade possível de *moves* e *joins* devido ao aumento exponencial das possibilidades possíveis na realização destas duas operações. Para obtenção do valor ótimo de similaridade, utilizamos a implementação proposta no trabalho de Wen e Tzerpos (2003) para diminuir o tempo de cálculo através de um algoritmo de otimização e garantir a menor quantidade de *moves* e *joins* para converter um agrupamento no outro.

Com a definição do *MoJoSim*, pode-se descrever a autoridade como o  $MoJoSim(C, C_A)$ , onde  $C$  é um agrupamento gerado por um dado algoritmo de

agrupamento e  $C_A$  é o agrupamento de referência relativo àquela versão. Já a estabilidade pode ser definida como  $MoJoSim(C_i, C_{i+1})$ , onde  $C_i$  é um agrupamento gerado por um algoritmo sobre a versão  $i$  de um sistema e  $C_{i+1}$  é o agrupamento gerado pelo mesmo algoritmo para uma versão consecutiva do mesmo sistema.

## EdgeSim

Esta métrica foi definida por Mitchell e Mancoridis (2001b) visando suprir as deficiências do *MoJo*, já que esta medida de similaridade, por não considerar as arestas entre as entidades ao realizar operações de *moves* ou *joins*, faz com que agrupamentos razoavelmente diferentes resultem em valores altos de *MoJoSim*. Apesar de demandar uma quantidade pequena de operações, o agrupamento pode ser muito diferente do outro. Com o *EdgeSim*, pretende-se levar em conta, ao analisar os *clusters* dos dois agrupamentos, as mudanças tanto nos vértices quanto nas arestas dos grafos particionados pelo agrupamento.

Dado um grafo  $G = (V, E)$  representando a estrutura de um sistema de software, sendo  $V$  o conjunto de entidades de código fonte (e.g., arquivos ou classes) e  $E$  o conjunto de dependências ponderadas entre as entidades (e.g., chamada de métodos, uso de variáveis). Os pesos das arestas geralmente indicam a força do relacionamento entre duas entidades de código de um sistema.

No cálculo do *EdgeSim*, procura-se mapear os pares de arestas que são *intracluster* ou *intercluster* em ambos os agrupamentos sob comparação. Arestas *intracluster* são arestas que conectam vértices que se localizam em um mesmo *cluster*. Arestas *intercluster* são as que conectam vértices localizados em *clusters* distintos em ambos os agrupamentos. A Figura 2.8 ilustra um grafo de um sistema hipotético no lado esquerdo e dois possíveis agrupamentos resultantes, no centro e à direita. Esta mesma figura será utilizada como exemplo base tanto para esta métrica quanto para a métrica *MeCl* que será descrita na seção 2.7.3.

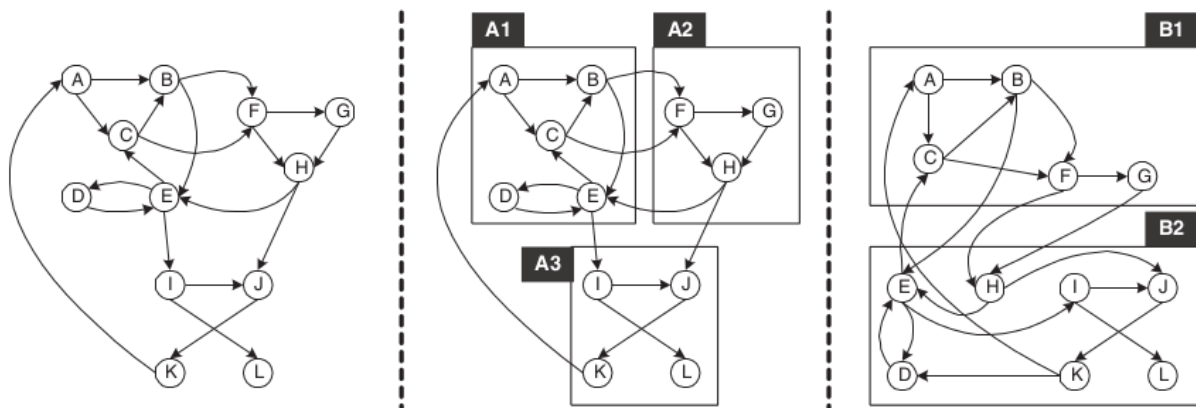


Figura 2.8: Grafo e dois possíveis agrupamentos. Adaptado de [Mitchell e Mancoridis 2001a]

Adaptado de

Ao final do cálculo da similaridade, *EdgeSim* irá dispor de um conjunto de pares de arestas que serão *intracluster* ou *intercluster* tanto em *A* quanto em *B*. Para simplificar, esta coleção de arestas será chamada de conjunto  $\Upsilon$ . A Figura 2.9 ilustra o conjunto  $\Upsilon$  baseado nas partições *A* e *B* da Figura 2.8 através das arestas em negrito.

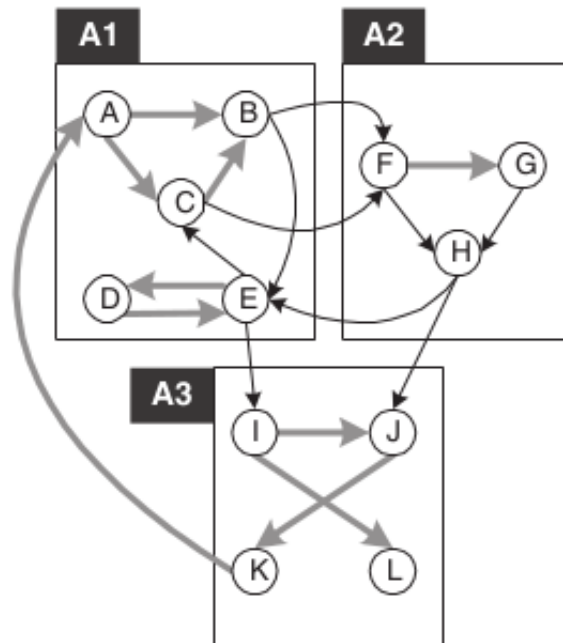


Figura 2.9: Conjuntos intracluster e intercluster [Mitchell e Mancoridis 2001a].

A partir do cálculo do conjunto  $\Upsilon$  (*intraclusters* e *interclusters*),  $EdgeSim(A, B)$  é definido como:

$$EdgeSim(A, B) = \frac{pesos(\Upsilon)}{pesos(E)} \quad (2.14)$$

onde  $pesos(\Upsilon)$  é a soma dos pesos das arestas do conjunto  $\Upsilon$ . No caso de arestas sem peso, basta assumir que elas possuem peso um.

Ao contrário do *MoJo* e do *MeCl* (Seção 2.7.3), o *Edgesim* possui a propriedade de ser reflexiva, logo:  $EdgeSim(A, B) = EdgeSim(B, A)$  [Mitchell e Mancoridis 2001a]. Assim como em *MojoSim*, podemos estabelecer *EdgeSim* tanto para autoridade quanto para a estabilidade seguindo a mesma ideia. A autoridade seria determinada pelo  $EdgeSim(C, C_A)$  e a estabilidade, através do  $EdgeSim(C_i, C_{i+1})$ .

### MeCl

Assim como *EdgeSim*, *MeCl* também foi idealizado por Mitchell e Mancoridis (2001b) para suprir as deficiências de *MoJo*, verificando a similaridade entre dois agrupamentos a partir de uma perspectiva diferente, considerando também tanto os vértices quanto as arestas. A palavra *MeCl*, do inglês (**Me** = *Merge* e **Cl** = *Cluster*), foi usada pois a métrica divide um agrupamento em *clusters* menores e reestrutura-os em seguida a fim de gerar o outro agrupamento.

No caso da Figura 2.8, cada *cluster* de *A* é interseccionado com todas os *clusters* de *B*, gerando um conjunto de subgrafos. Para a partição *A*, tem-se os *clusters*  $\{A_1, A_2 \text{ e } A_3\}$  e, para *B*, tem-se os *clusters*  $\{B_1 \text{ e } B_2\}$ . Assim, seis combinações são necessárias para verificar a intersecções dos *clusters* *A* com cada *cluster* de *B*. No exemplo, a intersecção de  $A_1$  com  $B_1$  é  $A_{1,1}$ , de  $A_1$  com  $B_2$  é  $A_{1,2}$  e assim por diante. Finalmente, cada *cluster* de *B* é união de todas as intersecções dos *clusters* de *A* realizadas com os *clusters* de *B* como pode ser visto na Figura 2.10.

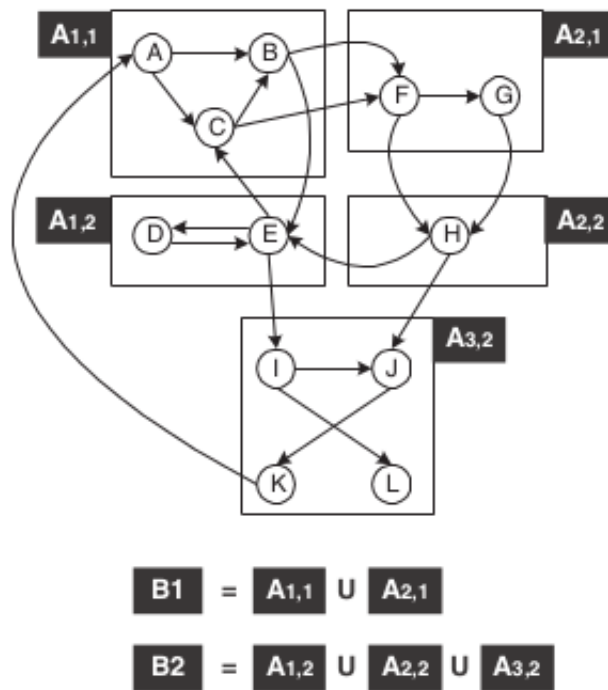


Figura 2.10: Calculando MeCl [Mitchell e Mancoridis 2001a]

Para  $B_1$ , Tem-se a junção das intersecções feitas de *A* com o  $B_1$  e, para  $B_2$ , tem-se a junção das intersecções feitas de *A* com o  $B_2$ . Neste caso, como  $A_{3,1}$  não possui intersecções, então, ele não contribui para a métrica, não sendo necessário expô-lo na equação presente na Figura 2.10.

Por fim, o Cômputo de *MeCl* é realizado a partir da junção de arestas *interclusters*

introduzidas a partir do processo de fusão dos subconjuntos. Neste exemplo, os subconjuntos gerados são  $A_{1,1}$ ,  $A_{1,2}$ ,  $A_{2,1}$ ,  $A_{2,2}$  e  $A_{3,2}$ . Logo:

$$MeCl(A, B) = 1 - \frac{pesos(\Upsilon_B)}{pesos(E)} \quad (2.15)$$

Onde  $pesos(\Upsilon_B)$  é a soma dos pesos do conjunto de arestas que são *intraclusters* em  $A$  mas são *interclusters* em  $B$ , gerando custos ao inserir novas arestas *interclusters*. Da mesma forma que com *EdgeSim*, se as arestas não forem ponderadas, então estas são consideradas com peso um.

$MeCl$  não é reflexivo. Logo  $MeCl(A, B) \neq MeCl(B, A)$ , sendo necessário fazer o cálculo de  $MeCl$  como na Equação 2.16.

$$MeClBidirecional(A, B) = \min(MeCl(A, B), MeCl(B, A)) \quad (2.16)$$

Onde  $MeClBidirecional$  é o valor mínimo do  $MeCl$  entre  $A$  e  $B$  e entre  $B$  e  $A$ . Por uma questão de simplicidade, de agora em diante, far-se-á referência a  $MeClBidirecional$  somente como  $MeCl$ . Analogamente a *MojoSim* e *EdgeSim*, determinamos a autoridade como  $MeCl(C, C_A)$  e a estabilidade, como  $MeCl(C_i, C_{i+1})$ .

## Precision-Recall

Assim como *MojoSim*, *Precision-Recall* não considera as arestas durante o cálculo da similaridade. Definida, para a área de recuperação arquitetural, por Anquetil e Lethbridge (1999a), a similaridade entre dois agrupamentos  $A$  e  $B$  é calculada com base na comparação de pares de entidades, definidos da seguinte forma:

- Precision: Percentual de pares *intra-clusters* no agrupamento  $A$  que também são pares *intra-cluster* no agrupamento  $B$
- Recall: Percentual de pares *intra-clusters* presentes no agrupamento  $B$  que também são pares *intra-cluster* no agrupamento  $A$

Um agrupamento com *clusters* de único elemento tem boa *precision* mas péssimo *recall*. Assim como um grande *cluster* com todos os elementos tem um bom *recall* mas uma baixa *precision*.

Para balancear os valores de *precision* e *recall*, utilizaremos a medida-F, que é definida como a média harmônica de *precision* e *recall*, e é dada por:

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \quad (2.17)$$

Podemos então definir a autoridade como  $F1(C, C_A)$  e a estabilidade, como o  $F1(C_i, C_{i+1})$ .



### B-Cubed Precision-Recall

Definido por Bagga e Baldwin (1998), a versão *B-Cubed* de *Precision-Recall* foi criada associando *precision* e *recall* para cada item no agrupamento.

A métrica B-Cubed Precision-Recall define *precision* a partir da quantidade de itens no mesmo *cluster* pertencentes à sua categoria e *recall* seria quantos itens de sua categoria aparecem no *cluster*. A B-Cubed *precision* de um item é a proporção de itens no seu *cluster* que possui a mesma categoria, inclusive, incluindo ele mesmo. No contexto da engenharia de software, dado dois agrupamentos *A* e *B*, podemos contabilizar a B-Cubed *precision* verificando, para cada item em *A*, quantos pares formados com ele são intra-*clusters* também em *B*. Para obter a B-Cubed *precision* do *cluster*, basta somar as *precisions* de todos os itens do *cluster* e fazer a média pela quantidade de itens. Para obter a B-Cubed *precision* de todo o agrupamento, computa-se a média dos valores de *precision* dos *clusters* pela quantidade de grupos. B-Cubed *recall* é análogo, bastando fazer a análise a partir dos itens de *B* como explanado no *Precision-Recall* definido por Anquetil e Lethbridge (1999a). A Figura 2.11 ilustra como o cálculo é realizado para um item.

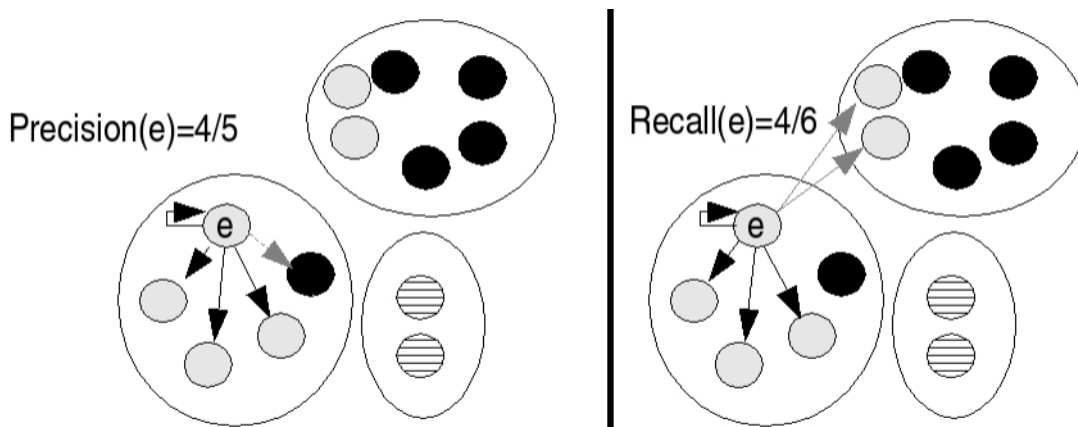


Figura 2.11: Cálculo da B-Cubed *Precision-Recall* para um item [Amigó et al. 2009]

Na Figura 2.11, pode-se verificar que a *precision* do item *e* em destaque é  $\frac{4}{5}$  pois somente um ponto dentro do seu *cluster* pertence à outra categoria. Já o *recall* é calculado como  $\frac{4}{6}$  pois, dos seis itens de sua categoria (incluindo ele mesmo), quatro estão dentro do mesmo *cluster* com ele.

Assim como com *Precision-Recall*, utilizaremos também a medida-F para balancear os valores de *precision* e *recall* computados pela métrica. Para simplificar, chamamos B-Cubed *Precision-Recall* somente como B-Cubed-F1. Dito isto, definimos a autoridade como  $B-Cubed-F1(C, C_A)$  e a estabilidade, como o  $B-Cubed-F1(C_i, C_{i+1})$ .

## Pureza

Zaki e Meira (2014) definem Pureza como uma métrica para quantificar quanto um *cluster*  $C_i$  é “puro”, isto é, quantos itens foram identificados corretamente ao se comparar dois agrupamentos  $A$  e  $B$ . Desta forma, é possível utilizar a Pureza como métrica de similaridade para calcular a proximidade entre dois agrupamentos.

Para calcular a Pureza, primeiramente deve-se montar uma matriz  $M$  onde as linhas são os *clusters* e as colunas são as categorias. Para cada *cluster*  $C_i$ , conta-se quantos objetos foram classificados como  $T_j$ . No contexto da engenharia de software, se um mesmo par de entidades tanto em  $A$  quanto em  $B$  forem *intra-cluster*, podemos então incrementar o valor na posição  $M[i][j]$ , onde  $C_i$  seria um *cluster* do agrupamento  $A$  e  $T_j$  seria um *cluster* do agrupamento  $B$ . A Tabela 2.5 ilustra esta matriz, também denominada de matriz de confusão, preenchida com valores hipotéticos.

Tabela 2.5: Matriz de confusão para o cálculo da pureza

	T1	T2	T3
C1	0	15	22
C2	29	7	0
C3	4	16	1

Por fim, para calcular a pureza, basta, selecionar o maior valor de cada linha, somar e dividir pela quantidade total de entidades do agrupamento. Dada a matriz da Tabela 2.5 e sendo 80 a quantidade de entidades de um agrupamento arbitrário, a pureza resultante pode ser calculada como na Equação 2.18.

$$P = \frac{(22 + 29 + 16)}{80} = 0.8375 \quad (2.18)$$

Podemos então definir a autoridade como  $Pureza(C, C_A)$  e a estabilidade, como  $Pureza(C_i, C_{i+1})$ .

### 2.7.4 Métricas intrínsecas de qualidade para avaliação de agrupamentos

Com o intuito de nos auxiliar na escolha de uma melhor métrica de similaridade para avaliação dos algoritmos, decidimos comparar os valores de autoridade com alguma métrica de avaliação intrínseca de agrupamentos. Diversas métricas existentes na literatura podem ser utilizadas para avaliar internamente os agrupamentos como a *Silhueta*, *Índice Davies-Bouldin* ou *Índice Dunn*. Contudo, escolhemos a silhueta por sua relativa simplicidade e baixo custo computacional.

A Silhueta é um método de avaliação interna de agrupamentos utilizado principalmente quando não há a possibilidade de comparação com uma *ground truth* [Han et al. 2012]. A silhueta retorna valores maiores de qualidade quando os *clusters* são compactos e, ao mesmo tempo, são bem espaçados uns dos outros. Em suma, a silhueta valoriza o agrupamento quando os *clusters* possuem itens que estão bem próximos e a distância entre os *clusters* é grande. A Equação 2.19 ilustra como a Silhueta é calculada.

$$S = \frac{b(i) - a(i)}{\max(a(i), b(i))} \quad (2.19)$$

Onde  $a(i)$  é a distância média da entidade  $i$  max a todas as demais entidades do seu *cluster* e  $b(i)$  é a distância média a todos as entidades do *cluster* mais próximo da entidade  $i$ . A Silhueta de um grupo é a média das Silhuetas dos itens do grupo. A Silhueta do agrupamento é a média das Silhuetas dos *clusters*. O valor da Silhueta varia entre -1 e 1, indicando um agrupamento de boa qualidade quanto mais o coeficiente se aproximar de um.

## 2.8 Trabalhos Relacionados

Trabalhos como os de Anquetil e Lethbridge (1997), Anquetil e Lethbridge (1999), Tzerpos e Holt (1999), Tzerpos e Holt (2000b), Mitchell e Mancoridis (2001b), Mitchell e Mancoridis (2001a), Wu et al. (2005), Bittencourt e Guerrero (2009), Garcia et al. (2013) e Lutellier et al. (2015) avaliaram distintos algoritmos de agrupamento a partir de um conjunto variado de sistemas desenvolvidos em C/C++ ou Java.

Anquetil e Lethbridge (1997) avaliaram a criação de agrupamentos baseados nos nomes dos arquivos a partir da decomposição dos nomes dos arquivos em todas as substrings de um determinado tamanho denominado  $n$ -grama. Os melhores resultados foram obtidos através do 3-gramas com 73.1% de precisão quando comparado com os agrupamentos determinados por engenheiros de software de um sistema de telecomunicações desenvolvido em C e composto por mais de 1500 arquivos. Pode-se perceber neste trabalho o esforço realizado para tentar avaliar os agrupamentos a partir de algum critério. Neste caso, o critério foi através da comparação com o modelo desenvolvido por especialistas.

Anquetil e Lethbridge (1999) avaliaram, mais tarde, a aplicação dos algoritmos hierárquicos aglomerativos a partir de quatro critérios de qualidade para agrupar arquivos: *design*, decomposição por especialistas, redundância e tamanho. O *design* representaria a ideia de um sistema com boa coesão e acoplamento. A decomposição por especialistas seria a comparação com modelos reais desenvolvidos por *experts*. A redundância significa que diferentes características estão propensas a fornecerem resultados parecidos, logo, é necessário computar as menores distâncias entre duas

entidades e quantos pares de características elas têm em comum. O tamanho leva em conta se os tamanhos dos *clusters* são similares.

Tzerpos e Holt (1999) apresentaram uma métrica para cálculo de similaridade entre dois agrupamentos denominada *MoJo*, como descrito na Seção 2.7.3. Este trabalho apresentou resultados experimentais relevantes com o intuito de provar a eficácia do novo método e sua aplicabilidade no processo de avaliação de agrupamentos. Diversos outros estudos são baseados no cálculo desta medida de similaridade.

Posteriormente, Tzerpos e Holt (2000b) definiram a dimensão de avaliação dos algoritmos de agrupamento denominada *estabilidade* e avaliaram os algoritmos hierárquicos aglomerativos utilizando o coeficiente de Jaccard para calcular as similaridades. Mais especificamente, foram avaliadas a estabilidade de algoritmos aglomerativos com regras de ligação simples e completa como definido na Seção 2.6), variando o ponto de corte.

Mitchell e Mancoridis (2001a) desenvolveram um sistema em Java denominado CRAFT para a produção automática de modelos de referência na ausência de modelos reais de especialistas. A abordagem utilizada gera agrupamentos de um mesmo sistema a partir da execução de diversas técnicas de agrupamento. Os módulos que se encontram no mesmo agrupamento são guardados. Um dos maiores dificuldades da recuperação arquitetural é obter modelos de referência para comparação. Neste trabalho, os autores propuseram uma forma automatizada de gerar modelos “artificiais” para simular os agrupamentos desenvolvidas por engenheiros de software.

Mais tarde, Mitchell e Mancoridis (2001b) propuseram e avaliaram duas novas métricas de similaridade denominadas *EdgeSim* e *MeCl* conforme exposto nas Seções 2.7.3 e 2.7.3. Eles aplicaram o algoritmo Bunch cem vezes e cada dois agrupamentos foram comparados utilizando Precision-Recall, MoJo, EdgeSim e MeCl. Além da criação destas duas novas métricas, eles também realizaram um estudo comparativo analisando componentes de código-fonte que geralmente contribuem negativamente nos resultados de similaridade como os módulos onipresentes e as bibliotecas. Neste estudo, as duas novas métricas visam produzir valores mais fidedignos de similaridade entre agrupamentos pois MoJoSim e Precision-Recall podem apresentar valores muito próximos de similaridade para agrupamentos muito distintos por não considerar as arestas do grafo de sistema durante o cálculo.

O trabalho de Wu et al. (2005) avaliou os algoritmos hierárquicos (Seção 2.6), ACDC e Bunch através de cinco sistemas *open source* desenvolvidos em C/C++ e utilizando três dimensões de avaliação: estabilidade, autoridade e não-extremidade (ver Seção 2.7). O cálculo das dissimilaridades foi realizado através de MoJo. Os modelos de referência utilizados para a avaliação da dimensão de autoridade foram gerados através da estrutura de diretórios assim como em trabalhos anteriores como o de Anquetil e Lethbridge (1999).

Bittencourt e Guerrero (2009) avaliaram a recuperação arquitetural de versões de quatro sistemas a partir das métricas definidas por Wu et. al. (2005) analisando as

*releases* de quatro sistemas utilizando a visão de pacotes como modelo de referência. A visão de pacotes seria o equivalente à estrutura de diretórios adotado por Wu et al. (2005) e Anquetil e Lethbridge (1999) em seus trabalhos. As similaridades entre agrupamentos foram calculadas a partir do MoJoSim como mostrado na Seção 2.7.3, calculando a similaridade ao invés da dissimilaridade. Os autores apontam, em sua pesquisa, a insuficiência dos algoritmos de promoverem a recuperação arquitetural sozinhos. Todavia, eles podem ser um primeiro passo para aprimorar e agilizar o processo de recuperação assistida por um especialista. Isto se deve aos baixos valores de autoridade encontrados, não só neste trabalho, mas em outros anteriores.

Garcia et al. (2013) avaliaram oito algoritmos de agrupamento a partir de seis sistemas *open source* comparando com os modelos de referência chamados também de *ground-truth architectures*. O cálculo de similaridade foi realizado a partir de três diferentes métricas. A primeira é relativa a uma versão aprimorada de MoJo denominado MoJoFM [Wen e Tzerpos 2003]. MoJoFM, assim como MoJoSim, é uma medida de similaridade. A segunda compara agrupamentos *cluster a cluster (c2c)* com o modelo *ground-truth* para determinar o quanto um agrupamento gerado por um algoritmo reflete os modelos de referência. A terceira foi realizada a partir de três critérios: estabelecer uma métrica de similaridade para o caso dos algoritmos aglomerativos, maximizar a função objetivo para o caso do *Bunch* e identificar um padrão particular nos módulos dos sistemas para o caso do ACDC. Dois dos algoritmos obtiveram melhores resultados apesar de todos apresentarem baixa precisão ao mapearem as *ground-truth architectures*.

Lutellier et al. (2015) avaliaram os mesmos algoritmos presentes no trabalho de Garcia et al. e usando cinco sistemas *open-source* desenvolvidos tanto em C/C++ quanto em Java. Eles fizeram a comparação com as *ground-truth architectures* extraídas, refletindo como a qualidade das entradas pode afetar os resultados. Uma maior ênfase foi dada ao sistema *Chromium* pois este foi o maior sistema analisado, com 9,7 milhões de linhas de código. Eles adicionaram duas novas métricas para o cálculo de similaridade, denominadas arquitetura para arquitetura (*a2a*) e qualidade de modularização turbo (*TurboMQ*), totalizando quatro métricas de similaridade. Os autores também afirmam que, apesar das técnicas de recuperação arquitetural serem avaliadas em diversos trabalhos, os resultados dos diferentes trabalhos nem sempre são consistentes.

# Capítulo 3

## Metodologia

Este trabalho utiliza uma metodologia de pesquisa quantitativa de engenharia de software experimental [Wohlin et al. 2012]. Neste contexto, técnicas, ferramentas ou métodos podem ser avaliados comparativamente através da definição de critérios de avaliação e sua aplicação. No contexto de engenharia de software experimental, é comum utilizar artefatos de software (e.g., documento, modelos, código) como entrada para a avaliação. Em nosso caso, código-fonte de projetos *open source* são usados como entradas para algoritmos de agrupamento automático e os modelos arquiteturais resultantes são comparados entre si ou contra modelos propostos pelos próprios desenvolvedores.

Usando o método GQM (*Goal-Question-Metric*) para estudos empíricos no contexto da engenharia de software [Wohlin et al. 2012], o objetivo deste estudo é **analisar** as métricas de similaridade e os algoritmos de agrupamento **com o propósito de** avaliar **com respeito à** a qualidade das métricas e técnicas de agrupamento **do ponto de vista** de pesquisadores **no contexto** de técnicas quase-automáticas de recuperação arquitetural de visões modulares de software.

Dessa forma, as atividades a serem desenvolvidas neste trabalho são:

1. Revisão bibliográfica;
2. Desenvolvimento de ferramenta;
3. Seleção de modelos para avaliação;
4. Extração de versões de software para avaliação;
5. Avaliação das métricas de similaridade;
6. Avaliação dos algoritmos de agrupamento;
7. Escrita dos resultados através da escrita de artigo e dissertação;

**Revisão bibliográfica.** Inicialmente, foi feito um estudo do estado da arte da área da engenharia reversa, principalmente publicações sobre sua origem e evolução.

Após definido o objetivo principal da pesquisa, foram revisados alguns trabalhos que avaliam algoritmos de agrupamento e as métricas de similaridade. Com o avanço das leituras e debates sobre estes trabalhos revisados, foi possível determinar o *design experimental*, assim como os algoritmos de agrupamento e métricas utilizados neste estudo.

**Desenvolvimento de ferramental.** Para que o estudo pudesse ser efetivamente posto em prática, foi essencial preparar o ferramental necessário. A *Design Suite* [Bittencourt et al. 2009], ferramental usado como base para este trabalho, possui diversas funcionalidades disponíveis. Contudo, foi necessário realizar alterações na suíte de ferramentas, tais como a implementação de algoritmos de agrupamento e métricas de similaridade de agrupamentos, assim como a correção de *bugs* que surgiram, com o intuito de prover um meio de coleta dos dados que foram processados e analisados. Assim, as métricas EdgeSim, MeCl, F1 e a B-Cubed F1 além dos quatro algoritmos hierárquicos aglomerativos foram implementados ao longo do estudo. A métrica MoJoSim já estava presente na ferramenta antes do início deste trabalho.

**Seleção de modelos para avaliação.** Para que a avaliação pudesse ser feita da melhor forma, foi necessário encontrar modelos de referência para comparação. Devido à dificuldade de obter modelos de referência, como retratado na Seção 2.8, utilizamos os modelos disponibilizados por Bittencourt (2012). Estes modelos foram de fundamental importância para a análise das métricas de similaridade de agrupamentos e dos algoritmos de agrupamento a partir dos sistemas alvo, como ilustrado na Tabela 3.4. A versão completa das *ground truth architectures* utilizadas se encontram no Apêndice A. Estes modelos são arquivos de texto com expressões regulares que classificam as classes presentes no código-fonte em grupos pré-determinados. Por exemplo, para o *SweetHome3D*, o *cluster sweetHome3DModel* deve conter todas as classes dentro do pacote *com.eteks.sweethome3d.model*, o *cluster sweetHome3DTools* deve conter todas as classes dentro do pacote *com.eteks.sweethome3d.tools* e assim por diante. Estes mapeamentos se encontram no trecho *#mapping* de cada modelo.

**Extração de versões de software para avaliação.** A extração automática é uma alternativa para acelerar o acesso a maiores quantidades de versões de software. A *Design Suite* fornece essa extração de forma automatizada. A partir da extração das versões de software, da geração de grafos de design de baixo nível de cada versão e após aplicar os algoritmos de agrupamento nas versões dos sistemas, foi possível realizar a avaliação das dimensões de estabilidade e autoridade. As entidades e relações de design de baixo nível extraídas pela ferramenta estão listadas na Tabela 3.1. A coluna intitulada *Source Entity* indica as entidades que podem ser extraídas do código-fonte. A coluna *Relation* indica as dependências estruturais que as entidades podem estabelecer umas com as outras.

Entidade Fonte	Relação
pacote	pacote contém classe
	pacote contém interface
interface	interface contém campo
	interface contém método
	interface contém interface
Classe	Classe contém campo
	Classe contém método
	Classe contém classe
	Classe estende classe
	Classe implementa interface
campo	Campo é uma classe
método	Método recebe classe
	Método retorna classe
	Método acessa campo
	Método chama método
	Método lança classe
	Método captura classe

Tabela 3.1: Entidades de código e relações de dependência extraídos pela *Design Suite* [Bittencourt et al. 2009]

Essencialmente, a *Design Suite* consegue extrair, métodos, atributos, classes, interfaces, pacotes e suas relações tais como herança, implementação, contenção, retorno, chamada, uso, dentre outras. Na *Design Suite*, as relações possuem peso um no cálculo das similaridades, isto é, o peso de todas as relações são iguais. Por exemplo, uma classe estender outra não é considerado mais importante do que um método retornar uma mesma classe. A partir da extração das entidades, é então montado um grafo com as entidade e seus relacionamentos. Estes grafos são gravados em arquivos no formato *GXL* para utilização posterior como entrada dos algoritmos de agrupamento.

**Avaliação das métricas de similaridade.** As métricas de qualidade pressupõem a comparação dos agrupamentos para saber quão similares eles são. Contudo, qual delas é a mais apropriada para tal tarefa? Para nos auxiliar nesta questão, realizamos i) uma avaliação de estatísticas descritivas destas métricas; ii) computamos a correlação dos valores de autoridade computados por cada métrica com uma métrica de avaliação intrínseca de agrupamento e iii) correlacionamos também os valores de estabilidade computados por cada métrica com as variações de código-fonte oriundas da própria evolução do software. Para estas variações, daremos o nome de deltas.

Utilizamos a silhueta como métrica de avaliação interna de comparação com os valores de autoridade e verificamos a correlação dos valores. Neste caso, se os



valores de autoridade forem altos, é desejável que os valores de silhueta sejam altos, indicando uma concordância entre as métricas baseadas em modelos e a silhueta.

Para capturar a evolução semanal dos sistemas, coletamos os deltas por linhas de código (LOC – do inglês *Lines Of Code*) e número de classes entre duas versões consecutivas e fizemos a correlação deles para com os valores de estabilidade. O  $\Delta LOC$  considera a quantidade de linhas removidas e adicionadas entre duas versões. O  $\Delta Classes$  calcula o módulo da diferença da quantidade de classes Java entre duas versões. Neste caso, espera-se que, se os valores de estabilidade forem altos, então os deltas serão baixos, pois, se o software sofreu uma grande modificação em algum momento, então a estabilidade deve cair devido aos agrupamentos serem mais diferentes.

**Avaliação dos algoritmos de agrupamento.** As métricas de similaridade utilizadas foram a base para realizar os cálculos das semelhança entre dois agrupamentos, seja entre o agrupamento produzido por cada algoritmo e um modelo de referência (autoridade), seja entre dois agrupamentos produzidos por cada algoritmo sobre, respectivamente, duas versões consecutivas de um dado sistema (estabilidade).

**Escrita de artigos e dissertação.** Com os resultados obtidos através deste estudo, foi produzido um artigo completo para um evento especializado na área de engenharia de software e, finalmente, a presente dissertação.

## 3.1 Design Experimental

A Figura 3.1 ilustra o design experimental desse estudo. Basicamente, o estudo está dividido em oito passos. No primeiro passo, foram extraídas versões semanais de quatro sistemas *open source* alvo no intervalo de um ano. No segundo passo, estas versões foram compiladas para que os grafos de design de baixo nível pudessem ser extraídos através da *Design Suite* a partir dos arquivos de bytecodes *jar* gerados pela compilação. O terceiro passo foi aplicar os algoritmos de agrupamento nos *designs* extraídos. No quarto e quinto passos, os agrupamentos foram submetidos à avaliação através das mensuração de autoridade e estabilidade para todas as métricas de similaridade de agrupamentos de interesse. No sexto passo, os valores de autoridade foram correlacionados com os valores de silhueta. No sétimo passo, os valores de estabilidade foram correlacionados com  $\Delta LOC$  e  $\Delta Classes$ . Por fim, as estatísticas descritivas dos resultados das métricas de qualidade e os valores das correlações resultantes desse processo foram analisados no oitavo passo com o intuito de avaliar, primeiramente, as métricas de similaridade e, em seguida, os algoritmos de agrupamento propriamente ditos.

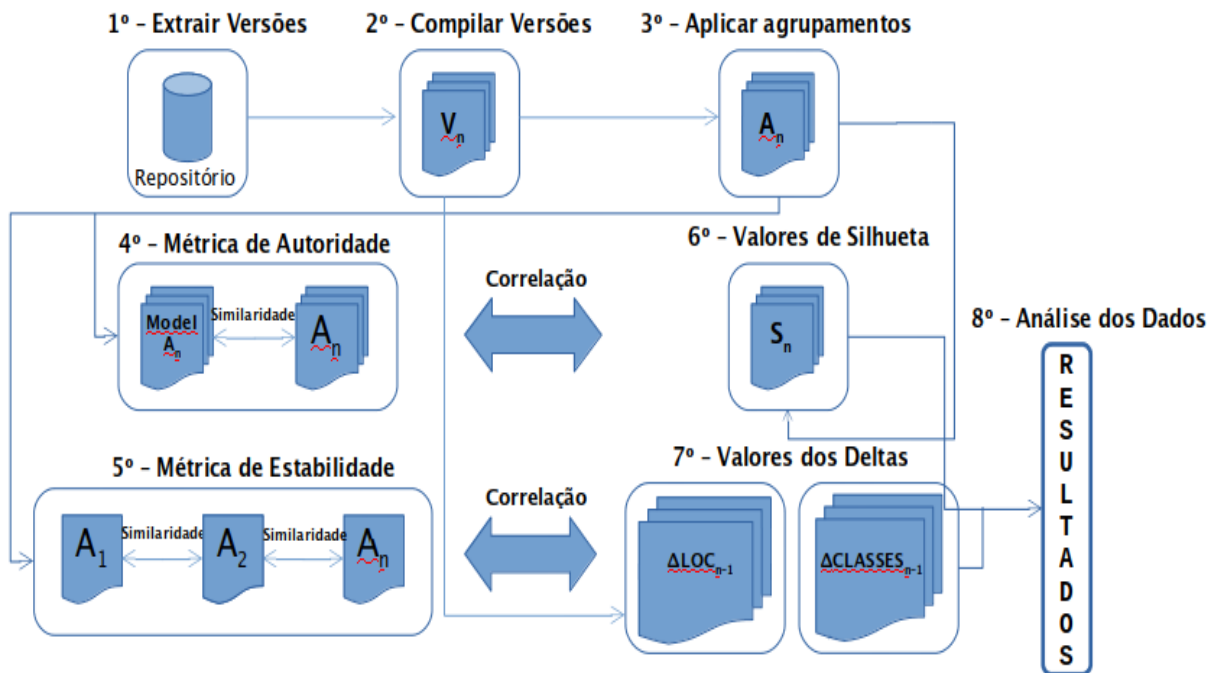


Figura 3.1: Design experimental do estudo

A Tabela 3.2 lista os algoritmos que avaliamos. Os algoritmos aglomerativos usados computam similaridade entre *clusters* através de técnicas de recuperação de informação, embora dependências estruturais também possam ser utilizadas. A recuperação de informação aplicada realiza as etapas de tokenização, remoção de *stop words*, normalização e *stemming* e o vetor de pesos é montado utilizando *tf-idf* reduzido através de *LSI* (Ver Seção 2.4). Utilizamos os pontos de corte 75 e 90 para os algoritmos aglomerativos assim como realizado nos trabalhos de Wu et al. (2005) e Bittencourt e Guerrero (2009).

Tabela 3.2: Algoritmos para análise

Nome	Tipo	Regra de Ligação
SL75	Hierárquico	Simples
SL90	Hierárquico	Simples
CL75	Hierárquico	Completa
CL90	Hierárquico	Completa

A Tabela 3.3 apresenta as métricas utilizadas para computar a estabilidade e a autoridade. No caso da autoridade, a comparação é realizada entre o agrupamento resultante do algoritmo e a *ground truth architecture* respectiva. Assim,  $C$  seria o agrupamento gerado automaticamente e  $C_A$  seria a arquitetura de referência respectiva gerada por especialistas. Já na estabilidade, são sempre analisadas duas versões consecutivas de agrupamentos gerados pelos algoritmos. Desta forma,  $A_i$  seria um

agrupamento de uma dada versão e  $A_{i+1}$  seria o agrupamento consecutivo, da versão uma semana posterior, a serem comparados, gerando  $i - 1$  valores de estabilidade.

Tabela 3.3: Métricas de similaridade de agrupamentos aplicadas na avaliação

	Autoridade	Estabilidade
MoJoSim	$MoJoSim(C, C_A)$	$MoJoSim(A_i, A_{i+1})$
EdgeSim	$EdgeSim(C, C_A)$	$EdgeSim(A_i, A_{i+1})$
MeCl	$MeCl(C, C_A)$	$MeCl(A_i, A_{i+1})$
F1	$F1(C, C_A)$	$F1(A_i, A_{i+1})$
B-Cubed-F1	$B-Cubed-F1(C, C_A)$	$B-Cubed-F1(A_i, A_{i+1})$
Purity	$Purity(C, C_A)$	$Purity(A_i, A_{i+1})$

Analizamos os sistemas listados na Tabela 3.4. Todos são *softwares* de código aberto desenvolvidos em linguagem Java.

Tabela 3.4: Sistemas alvo com *ground truth architectures*

Sistema	Intervalo	# Classes	# Módulos	Tamanho (KLOC)
SweetHome3D	08/03/2009 a 28/02/2010	142 a 165	9	42 a 55
Ant	29/10/2006 a 21/10/2007	487 a 511	16	122 a 123
Lucene	21/03/2010 a 13/03/2011	473 a 513	7	152 a 154
ArgoUML	19/11/2006 a 11/11/2007	1388 a 1524	19	151 a 177

## 3.2 Análise dos Resultados

A avaliação teve duas etapas: primeiro, a avaliação das métricas; em seguida, a avaliação dos algoritmos de agrupamento.

### 3.2.1 Avaliação das métricas de similaridade

Como cada métrica produz valores distintos de similaridade. Assim, observamos a concentração e a dispersão de cada métrica aliadas às correlações com a silhueta e deltas para determinar a métrica mais apropriada na tarefa de recuperar uma visão arquitetural modular de software. No caso da autoridade, correlacionamos os valores mensurados das métricas com os valores de silhueta e, no caso da estabilidade, correlacionamos os valores medidos com os valores dos deltas de *LOC* e classes.

### 3.2.2 Avaliação dos algoritmos de agrupamento

Para avaliar os algoritmos de agrupamento, utilizamos algumas medidas absolutas e relativas de classificação de séries apresentadas por Wu et al. (2005) com base nas métricas de similaridade utilizadas.

A métrica relativa *Above* compara uma série de dados de um algoritmo com relação aos outros para cada sistema analisado. A métrica *Above* é dada por:

$$Above(DS_i, DS_j) = \frac{|\{n | DS_i[n] > DS_j[n], 1 \leq n \leq |DS_i|\}|}{|DS_i|} \quad (3.1)$$

onde  $DS_i$  e  $DS_j$  são duas séries de dados a serem comparadas. Para  $k$  séries, a medida relativa de  $DS_i$  em relação às outras séries é:

$$Above(DS_i) = \sum_{j=1}^k Above(DS_i, DS_j) \quad (3.2)$$

Já a métrica absoluta utilizada compara as séries de dados com patamares bem definidos chamados de  $H$ ,  $M$  e  $L$ . Esta métrica é denominada de avaliação ordinal *HML* [Wu et al. 2005]. A Figura 3.2 ilustra uma série de dados sendo comparada com esses limiares.

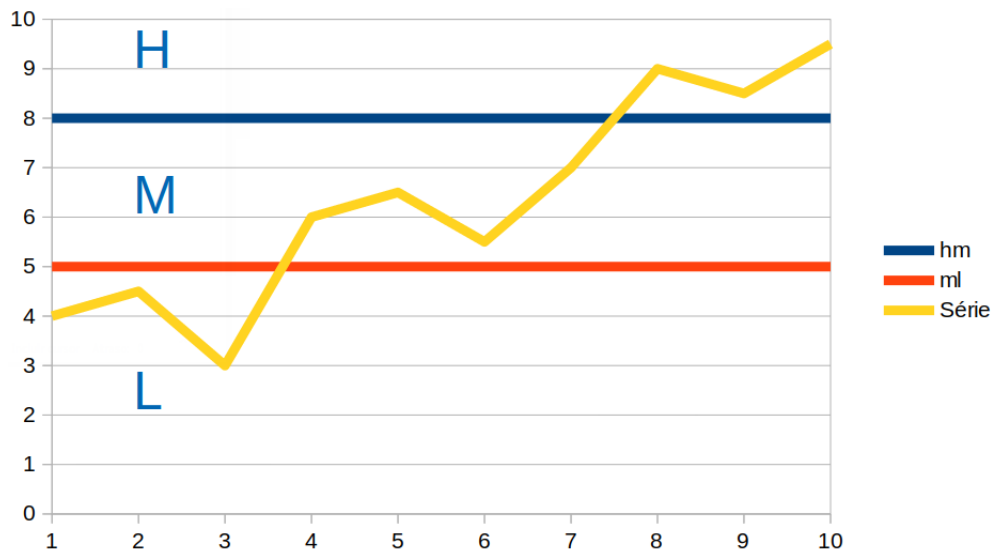


Figura 3.2: Comparação através da métrica absoluta HML

Para o uso do *HML*, os parâmetros foram utilizados segundo a Equação 3.3 definida por Bittencourt e Guerrero (2009) como:

$$HML(DS_i) = \begin{cases} H, & \text{se } Above(DS_i, hm) \geq 0.8 \\ M, & \text{se } Above(DS_i, ml) \geq 0.8 \\ L, & \text{Caso contrário} \end{cases} \quad (3.3)$$

Onde  $hm$  e  $ml$  são valores constantes que dividem o gráfico em três regiões. O valor de  $hm$  separa a região de valores elevados da região de valores médios, enquanto o  $ml$  separa a região de valores médios da região de valores baixos da métrica em análise. Com isto, é possível mensurar o quanto um algoritmo se aproxima dos modelos ou o quanto é estável em relação a um sistema em particular.

No exemplo da equação 3.3, caso 80% dos pontos de uma série de dados fiquem localizados na região  $H$ , então a série possui *score*  $H$ . Caso 80% dos pontos de uma série de dados fiquem localizados na região contendo  $H$  e  $M$ , então a série possui *score*  $M$ . O *score* será  $L$  caso contrário. Quanto mais *scores*  $H$  um algoritmo obter, melhor é o algoritmo de maneira absoluta.

# Capítulo 4

## Resultados

Aplicados os algoritmos de agrupamento, computamos a autoridade e estabilidade utilizando as métricas de similaridade *MojoSim*, *EdgeSim*, *MeCl*, *F1*, *B-Cubed-F1* e *Pureza* para a comparação dos agrupamentos. Primeiramente, avaliamos as métricas de similaridade e, em seguida, partimos para os resultados da avaliação dos algoritmos de agrupamento.

### 4.1 Análise das métricas de similaridade

Em primeiro lugar, avaliamos as métricas através do critério de autoridade e, posteriormente, através da estabilidade.

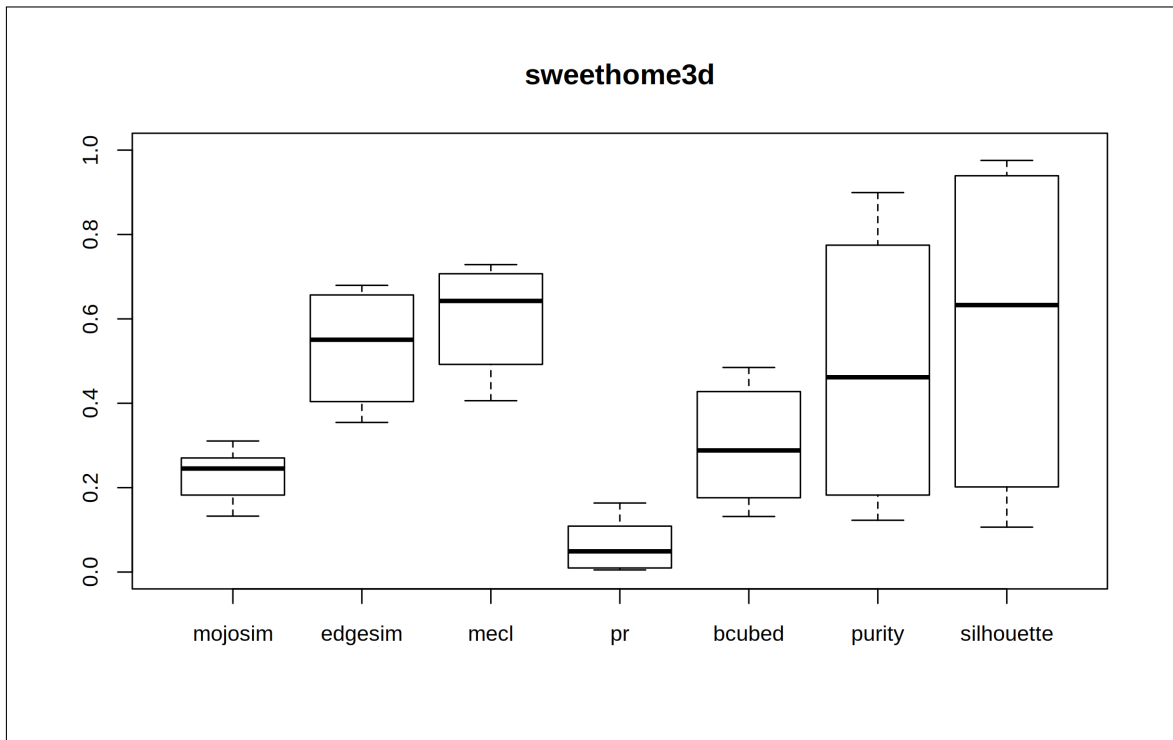
#### 4.1.1 Avaliação da autoridade

Espera-se que uma métrica de similaridade que mensure autoridade possua poder discriminatório suficiente para identificar diferenças entre a versão gerada pelo agrupamento e pela versão gerada por especialistas. Computamos então os valores de autoridade gerados pelas métricas e estudamos a concentração e a dispersão dos pontos de cada métrica com o propósito de analisar o posicionamento e as faixas de valores gerados.

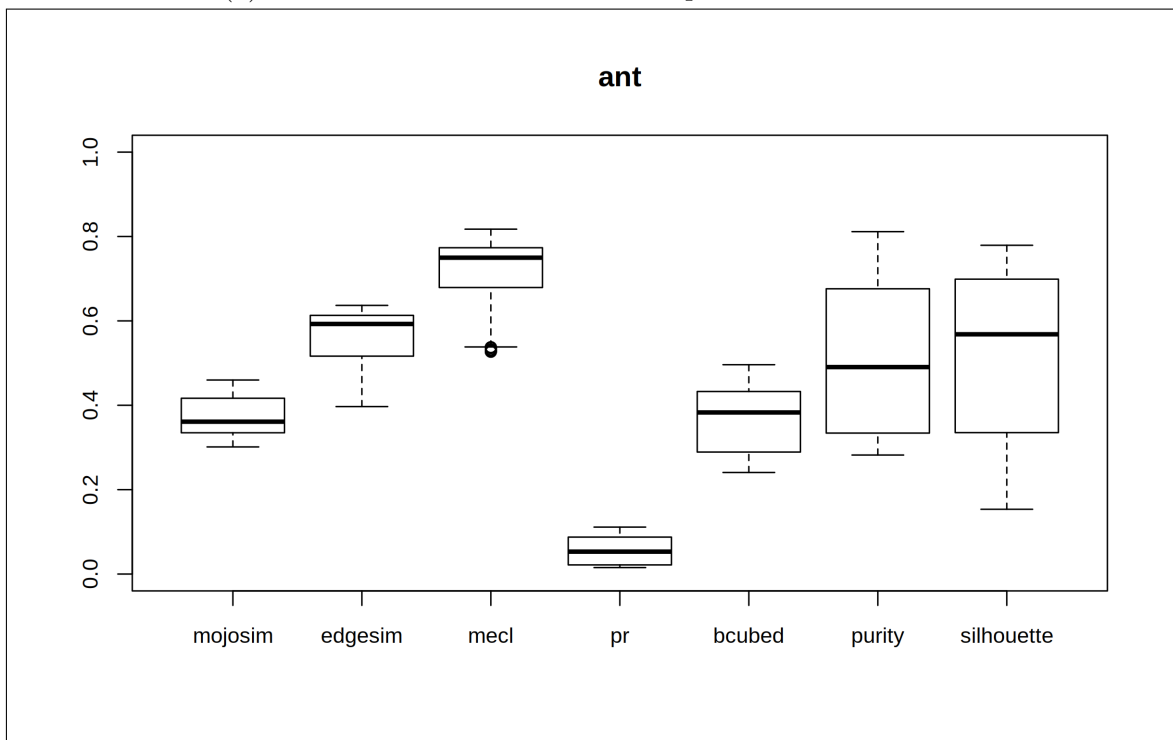
Aliado a isso, realizamos uma comparação dos valores de autoridade resultantes de todas as métricas com os valores de Silhueta. Calculamos o valor de Silhueta nos agrupamentos gerados pelos algoritmos em cada sistema e aplicamos a correlação de *Pearson* entre os valores de autoridade com a Silhueta. A partir deste comparativo, podemos então verificar se há alguma relação entre uma métrica que considera modelos *ground truth* com alguma medida de avaliação intrínseca que não considera modelos.

Como a silhueta gera valores de -1 a 1, normalizamos os valores de silhueta através da expressão  $S_{normalizada} = \frac{S+1}{2}$  com o intuito de trazer a silhueta para a mesma escala das métricas (entre 0 e 1), facilitando a análise dos gráficos de caixas e bigodes (*box plots*).

As Figuras 4.1 e 4.2 ilustram os dados gerados com os valores de autoridade obtidos neste estudo. Observando os *box plots*, ao comparar as métricas baseadas em modelos entre si, o MeCl apresenta uma mediana mais alta e valores mais altos de autoridade, seguida por EdgeSim (com exceção do ArgoUML), depois por Pureza (dispersão maior), MojoSim, B-Cubed-F1 (dispersão maior) e F1. Para todos os gráficos de dispersão por métrica, o eixo nomeado como *pr* representa os valores da medida-F de *Precision-Recall*, enquanto o eixo nomeado como *bcubed* representa a medida-F da B-Cubed *Precision-Recall*.

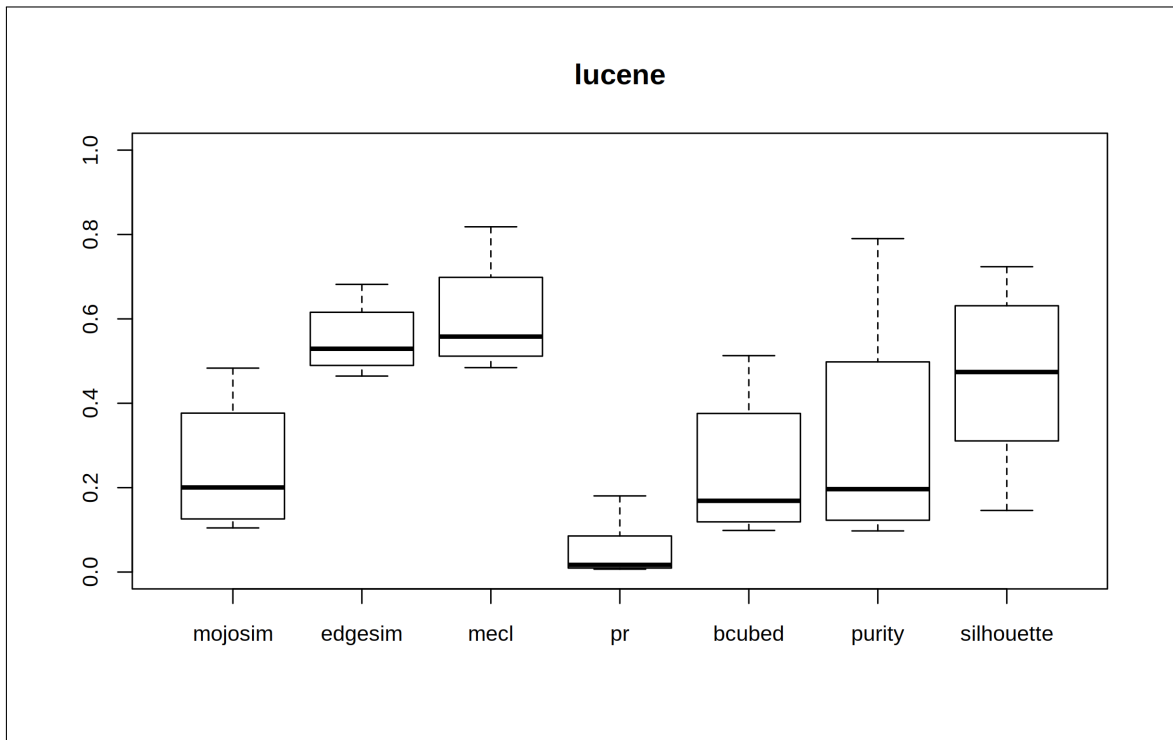


(a) Valores de autoridade e silhueta para o SweetHome3D

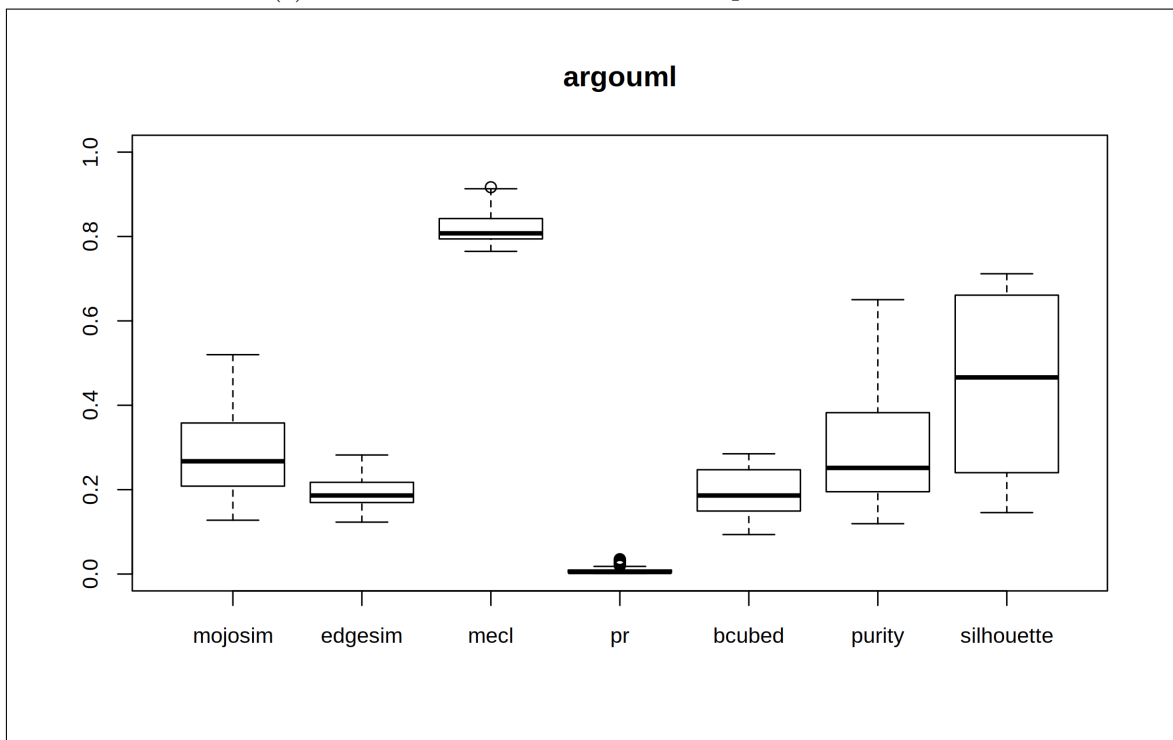


(b) Valores de autoridade e silhueta para o Ant



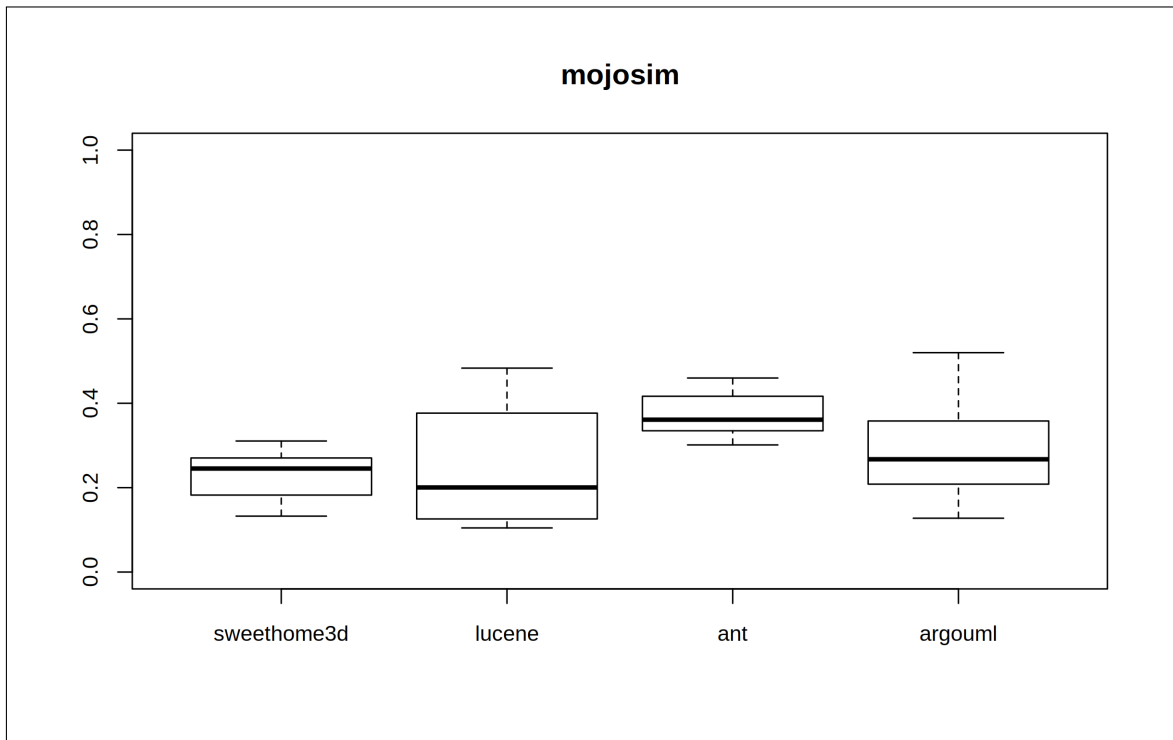


(c) Valores de autoridade e silhueta para o Lucene

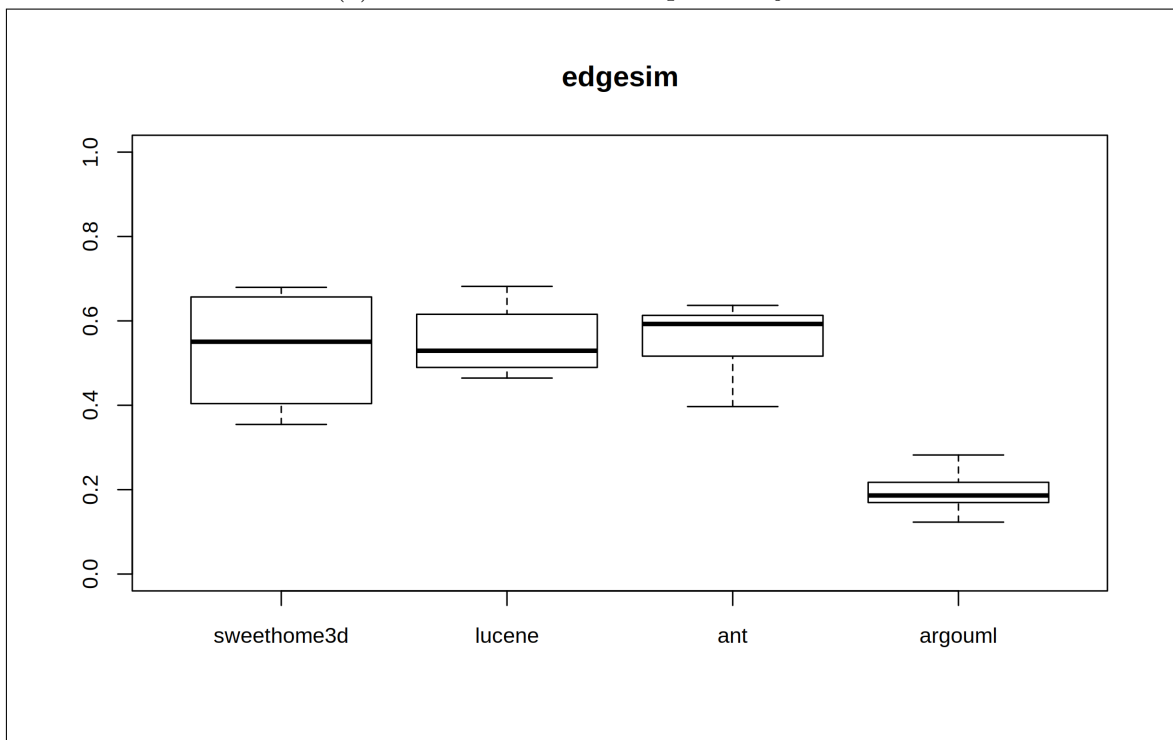


(d) Valores de autoridade e silhueta para o ArgoUML

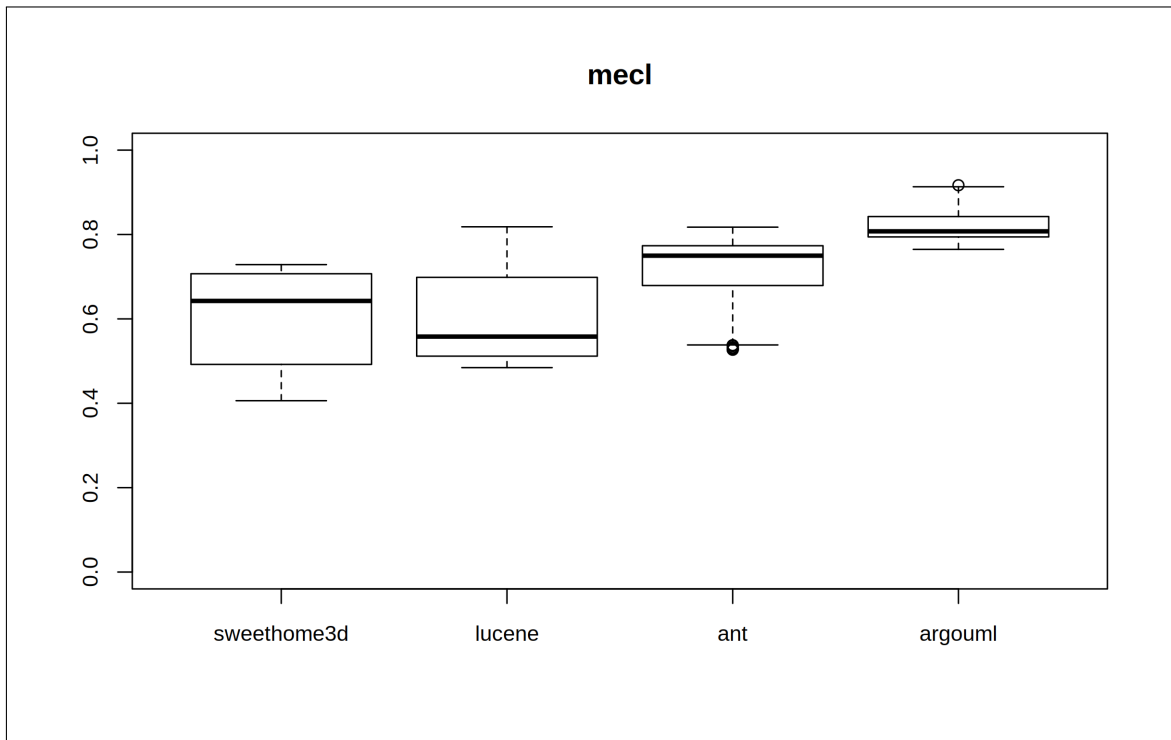
Figura 4.1: Distribuição dos valores de autoridade e silhueta por sistema



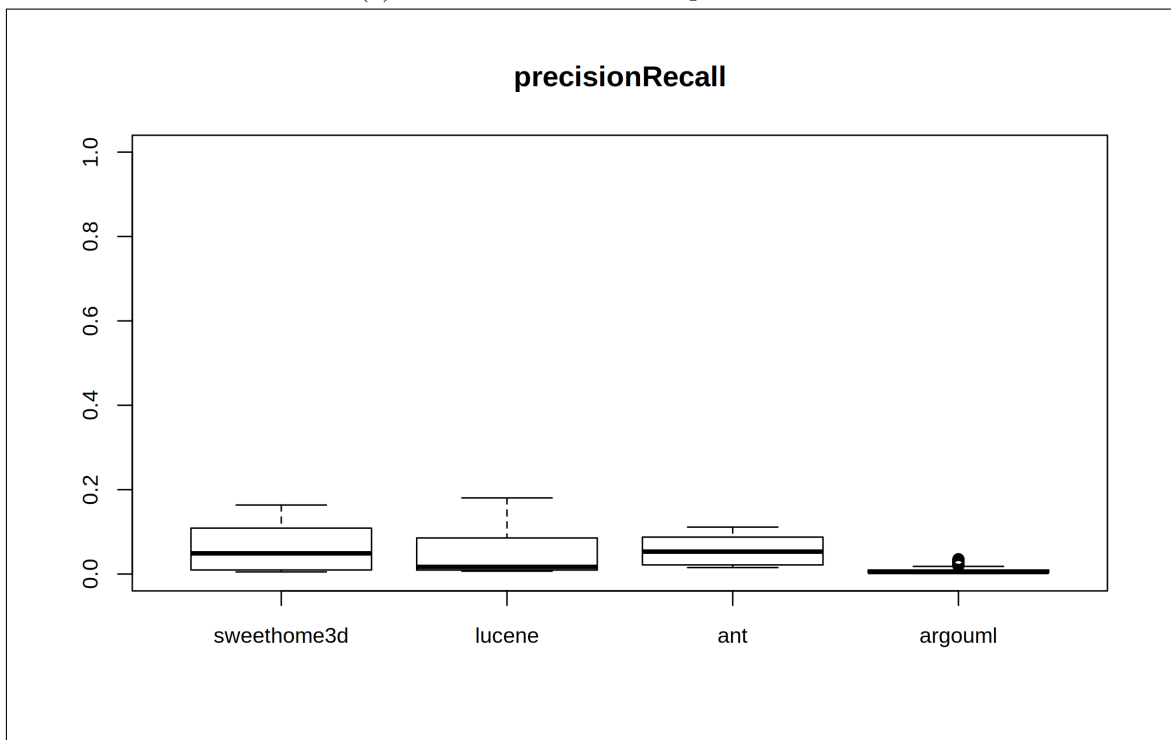
(a) Valores de autoridade para MojoSim



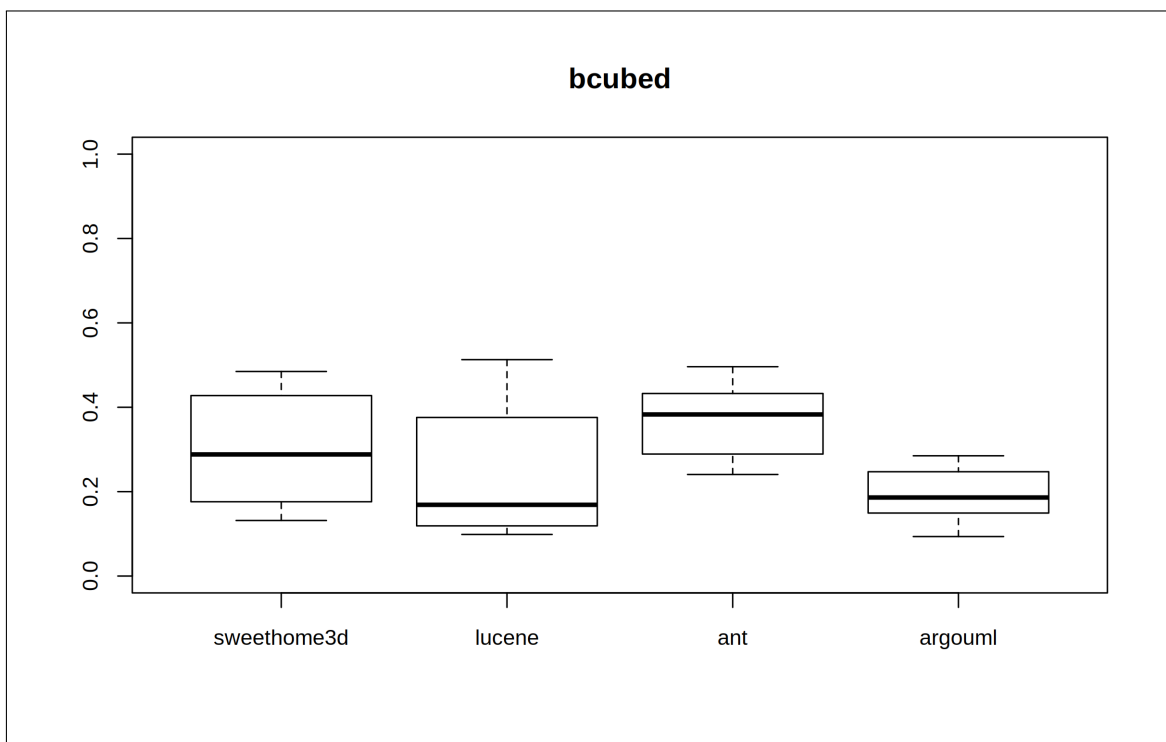
(b) Valores de autoridade para EdgeSim



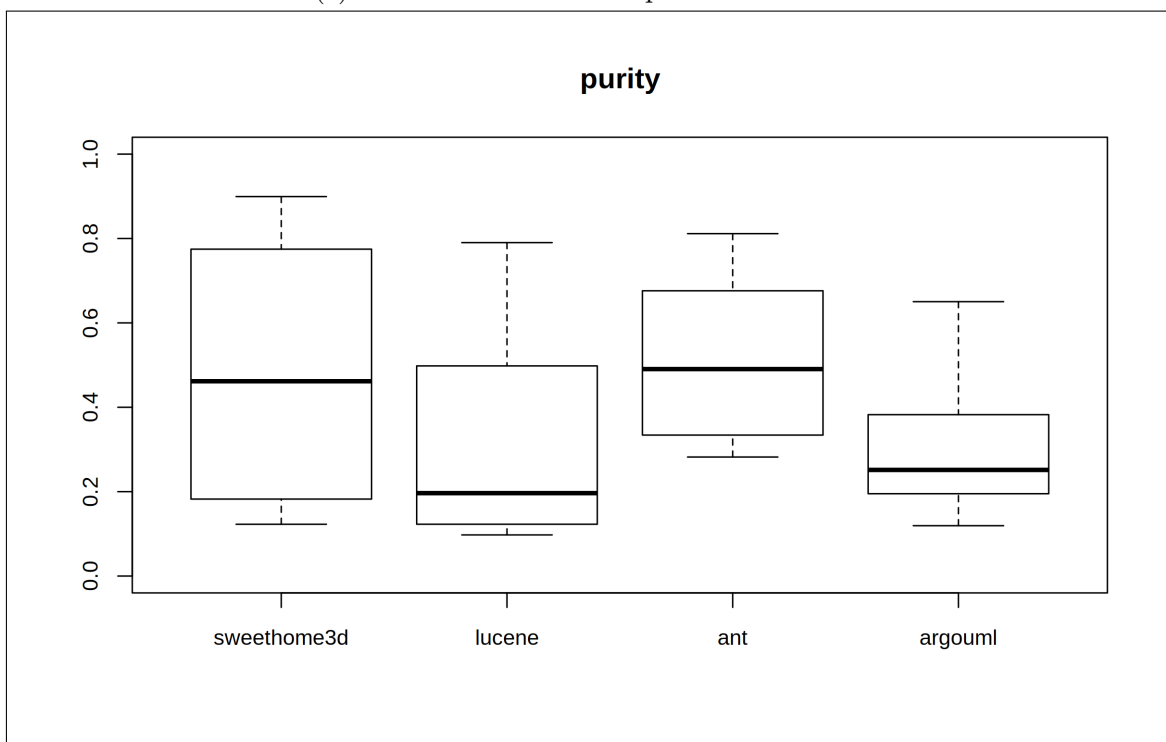
(c) Valores de autoridade para MeCl



(d) Valores de autoridade para F1



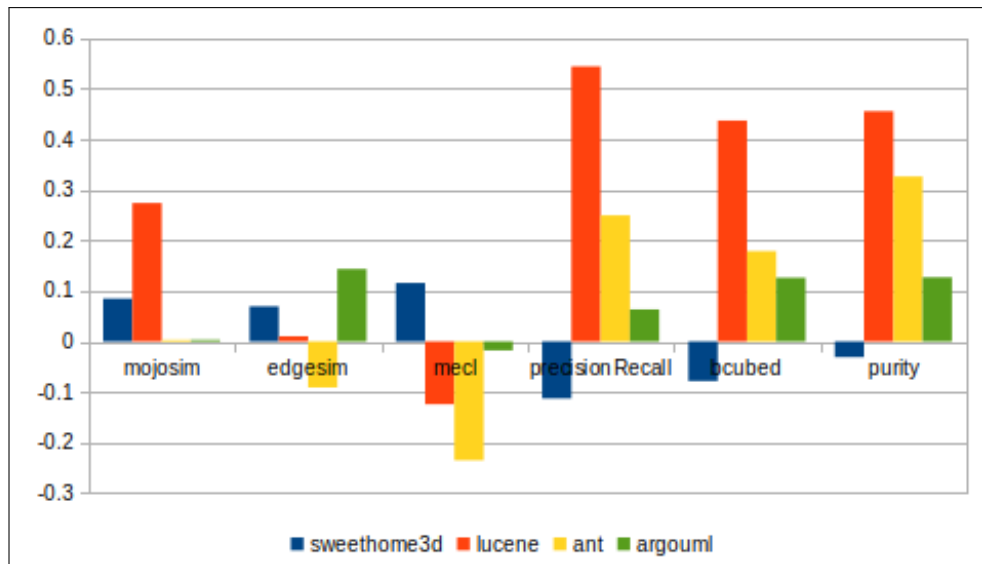
(e) Valores de autoridade para B-Cubed F1



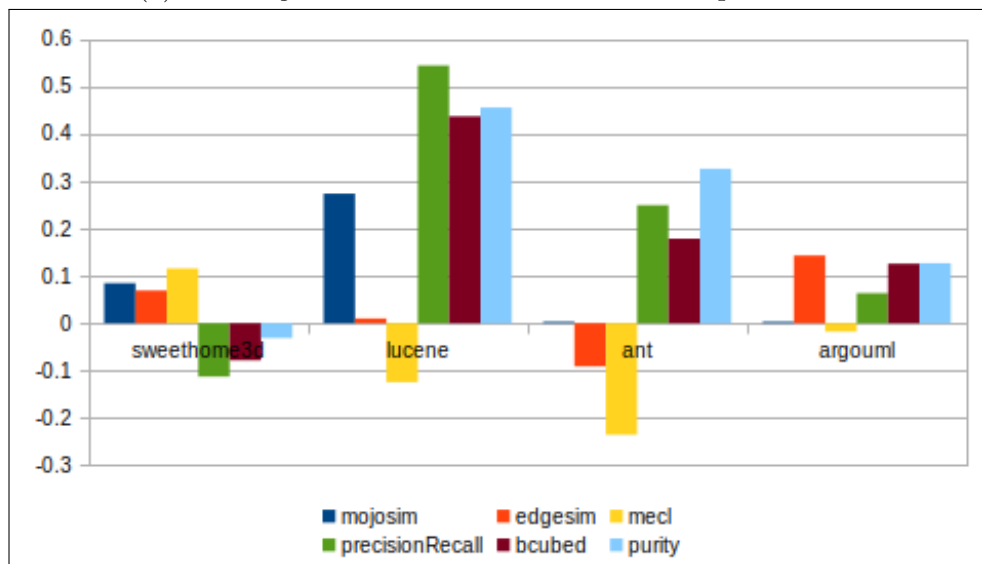
(f) Valores de autoridade para Pureza

Figura 4.2: Distribuição dos valores de autoridade por métrica

A Figura 4.3 ilustra os gráficos das correlações de Pearson das métricas com a silhueta. A partir das correlações das métricas com a silhueta, podemos verificar que, no gráfico por sistema, as medidas de pureza, B-Cubed-F1 e F1 correlacionam mais fortemente com a silhueta. Já no gráfico por métrica, há uma tendência de redução das correlações com o aumento do tamanho dos sistemas quando as medidas baseadas em modelos são comparadas com a Silhueta.



(a) Correlações da autoridade com a silhueta por sistema



(b) Correlações da autoridade com a silhueta por métrica

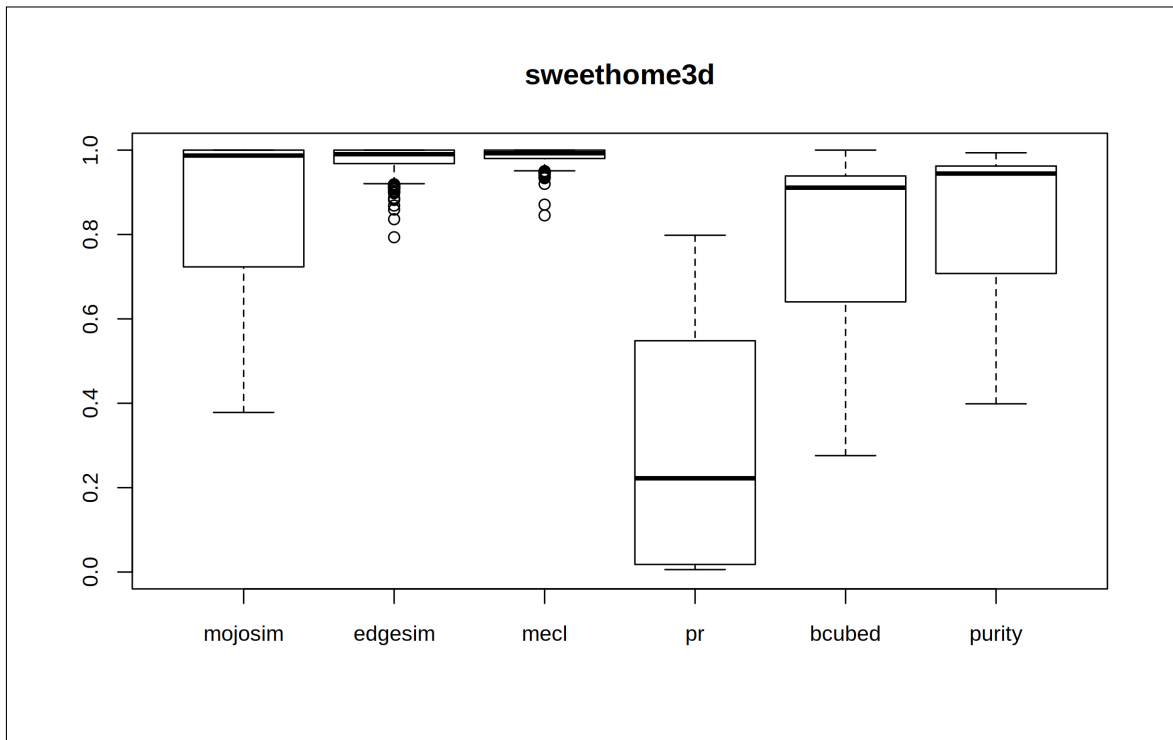
Figura 4.3: Correlações médias com a silhueta

### 4.1.2 Avaliação da estabilidade

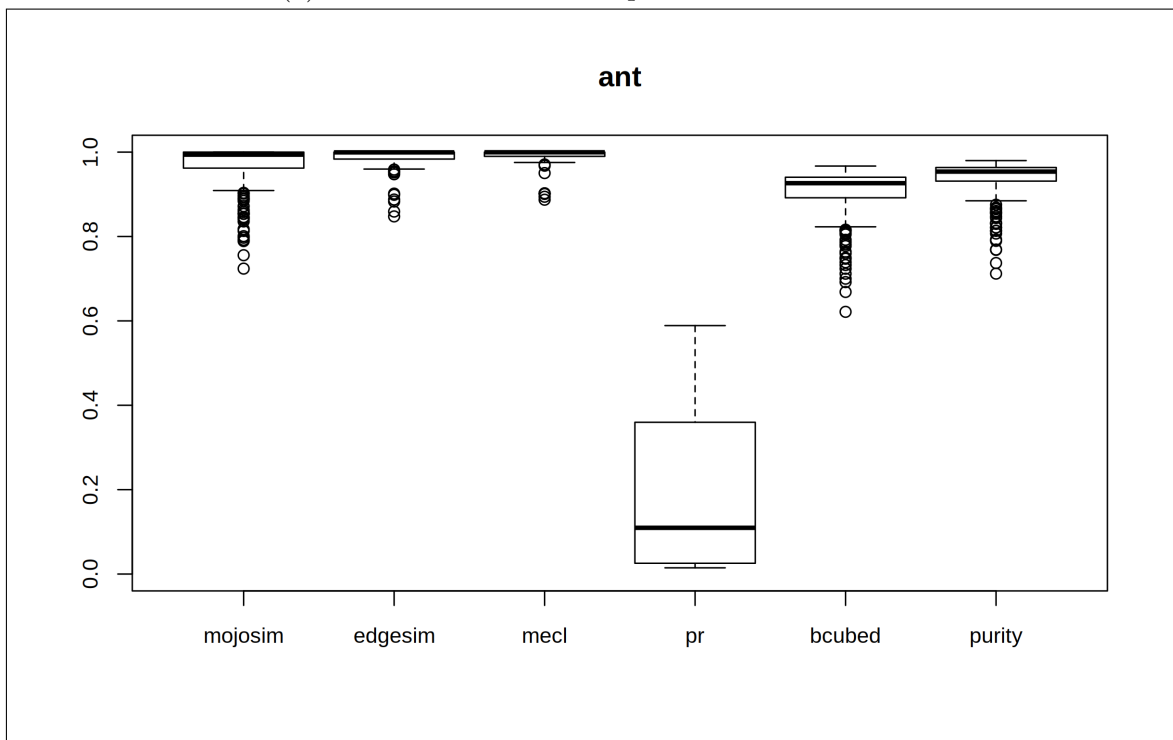
Espera-se que uma métrica de similaridade que mensure estabilidade tenha poder discriminatório suficiente para identificar diferenças entre as versões consecutivas dos sistemas. Computamos então os valores de estabilidade das métricas de similaridade e estudamos a concentração e a dispersão dos pontos para averiguar o posicionamento e as faixas dos dados de similaridade gerados pelas métricas.

Aliado a isso, obtivemos os valores dos deltas de linhas de código e classes de cada versão dos sistemas e computamos a correlação de *Pearson* entre os valores de estabilidade gerados pelas métricas e cada delta das versões. A partir deste comparativo, podemos verificar se há alguma relação entre os valores de estabilidade gerados por meio das métricas e a evolução do código-fonte do software.

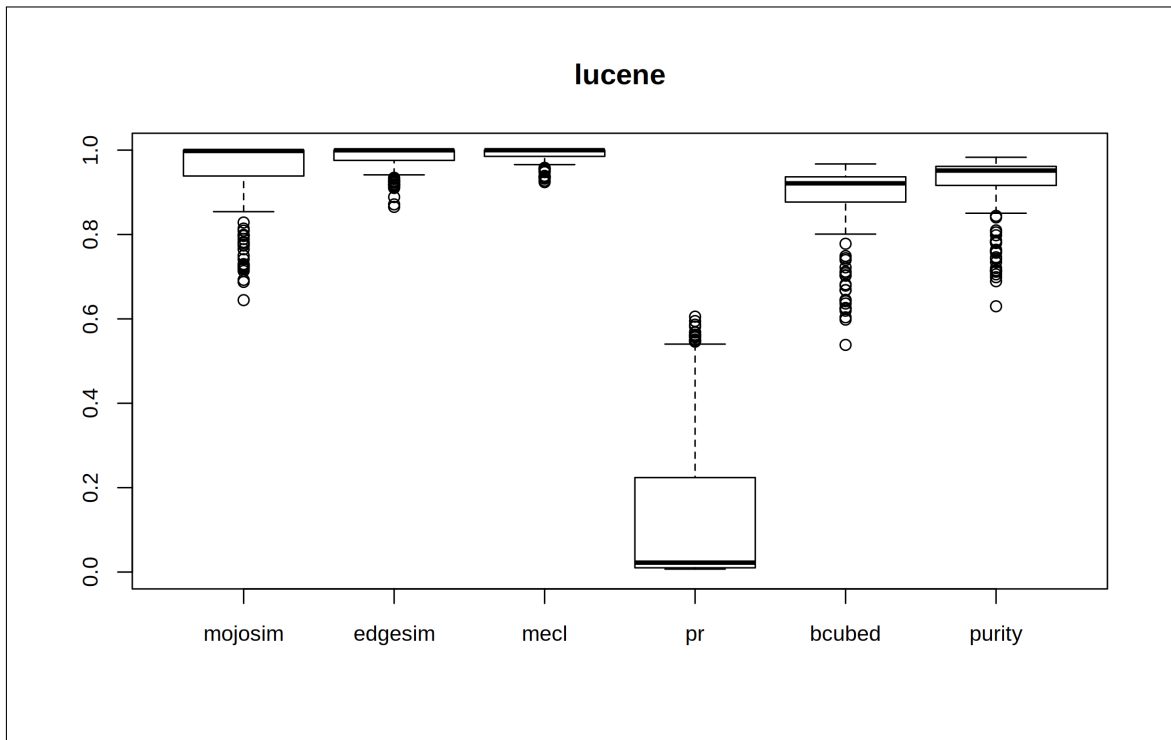
As Figuras 4.4 a 4.5 ilustram os dados de dispersão gerados. Observa-se que a maioria das métricas mantém valores altos de estabilidade, o que condiz com a maioria das pequenas mudanças semanais nos sistemas. Por outro lado, as medidas de estabilidade da maioria das métricas apresentam *outliers* com menor estabilidade, o que provavelmente captura as situações em que ocorreram mudanças um pouco maiores nos sistemas. Comparando as métricas entre si, MeCl possui uma concentração maior com valores altos de estabilidade, seguida por EdgeSim (com exceção do ArgoUML), MojoSim, Pureza e B-Cubed-F1. Somente F1 teve valores muito baixos de estabilidade.



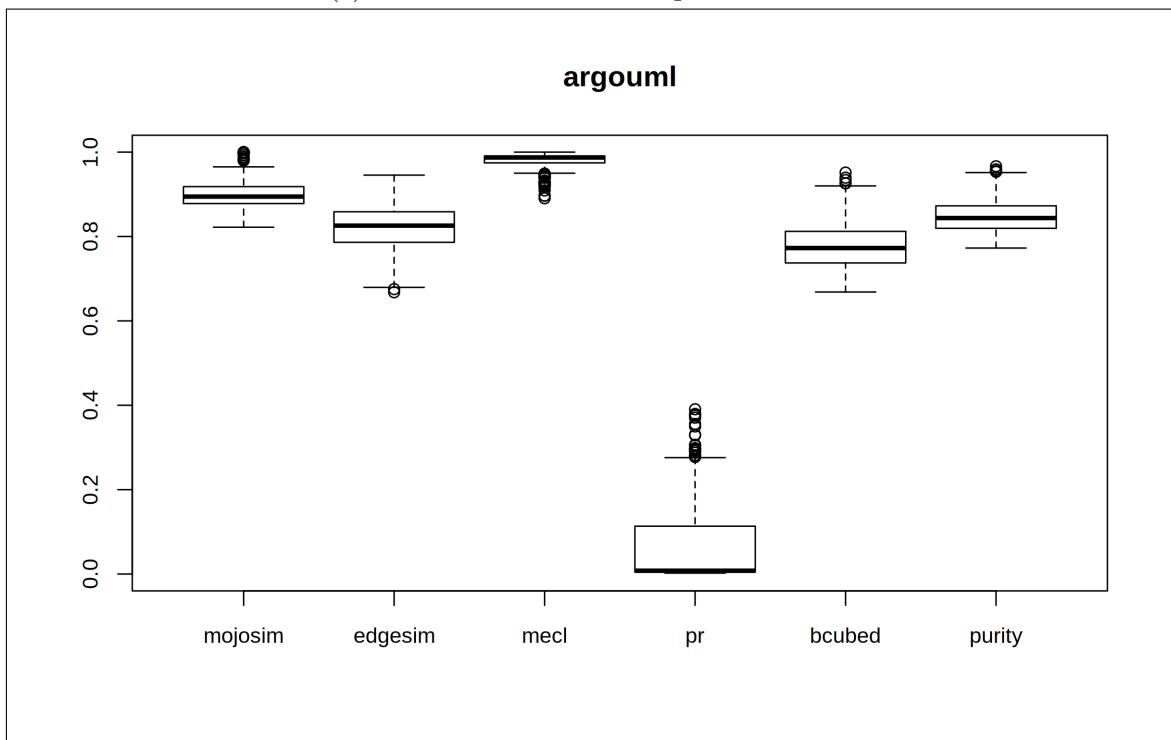
(a) Valores de estabilidade para o SweetHome3D



(b) Valores de estabilidade para o Ant



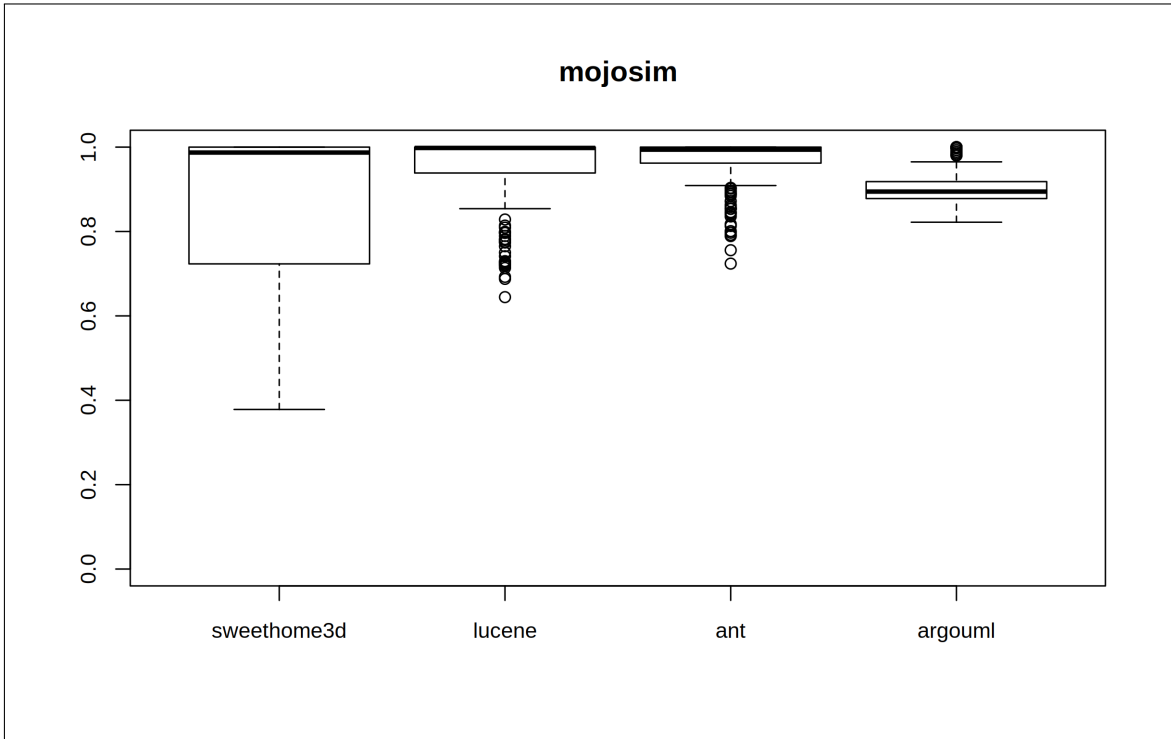
(c) Valores de estabilidade para o Lucene



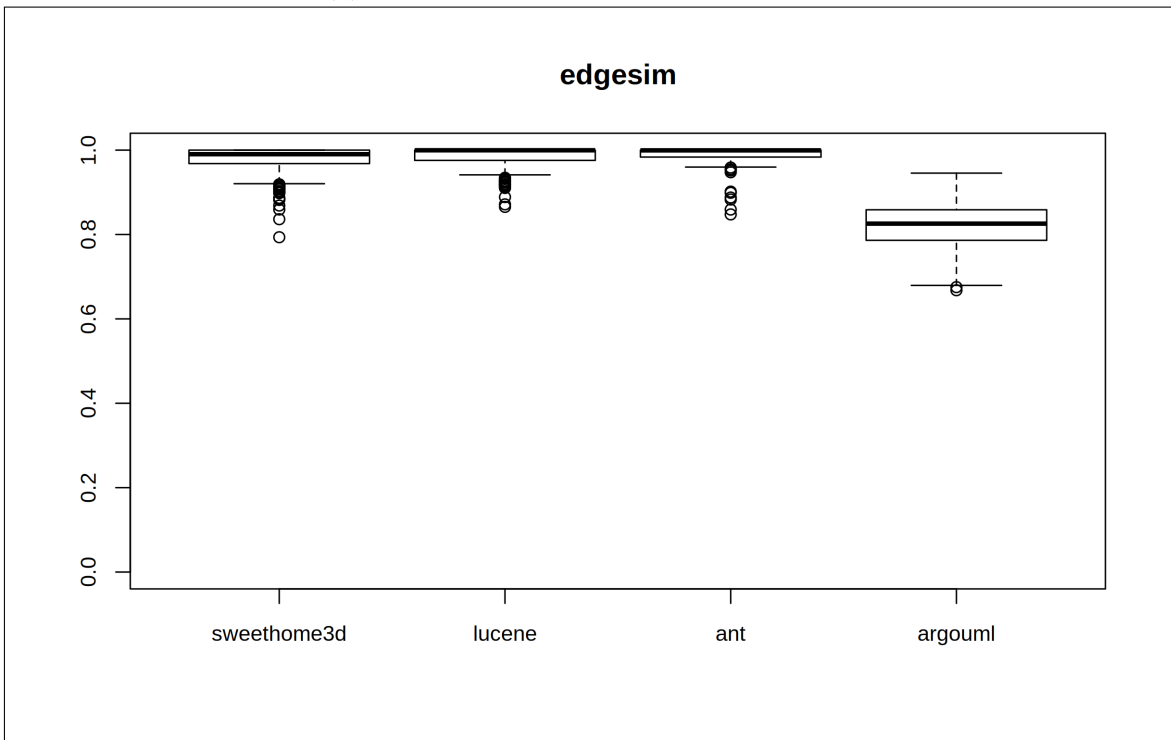
(d) Valores de estabilidade para o ArgoUML

Figura 4.4: Distribuição dos valores de estabilidade por sistema

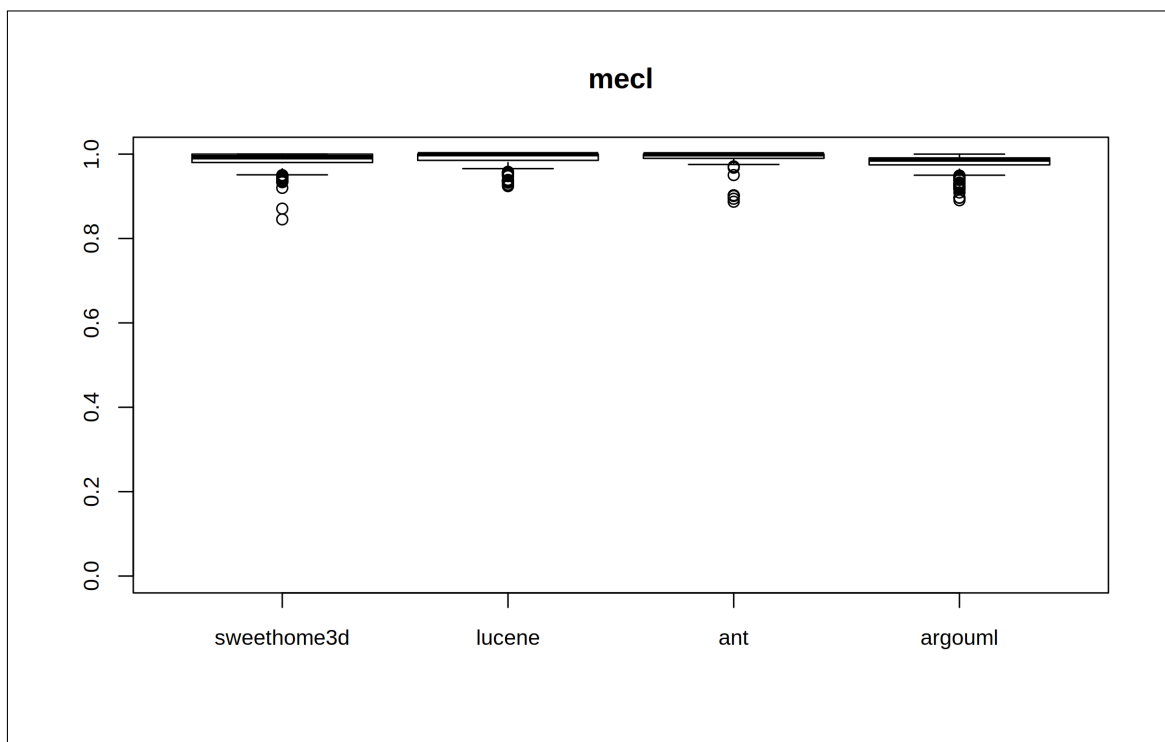




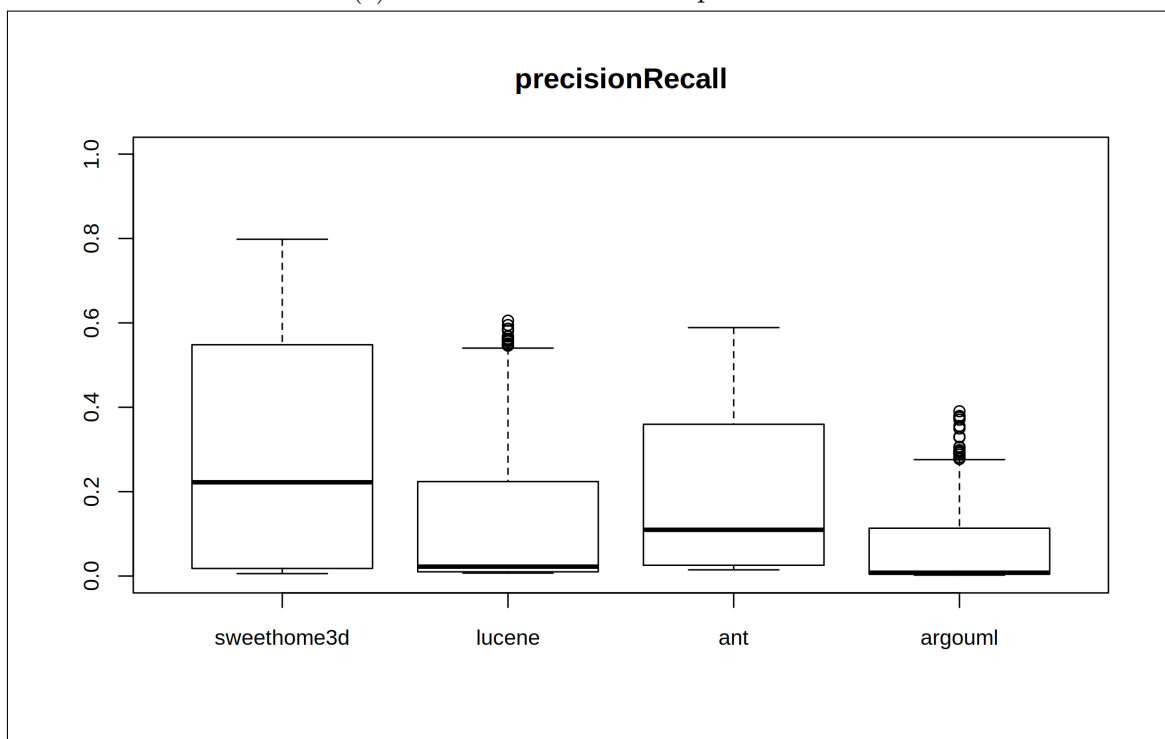
(a) Valores de estabilidad para MojoSim



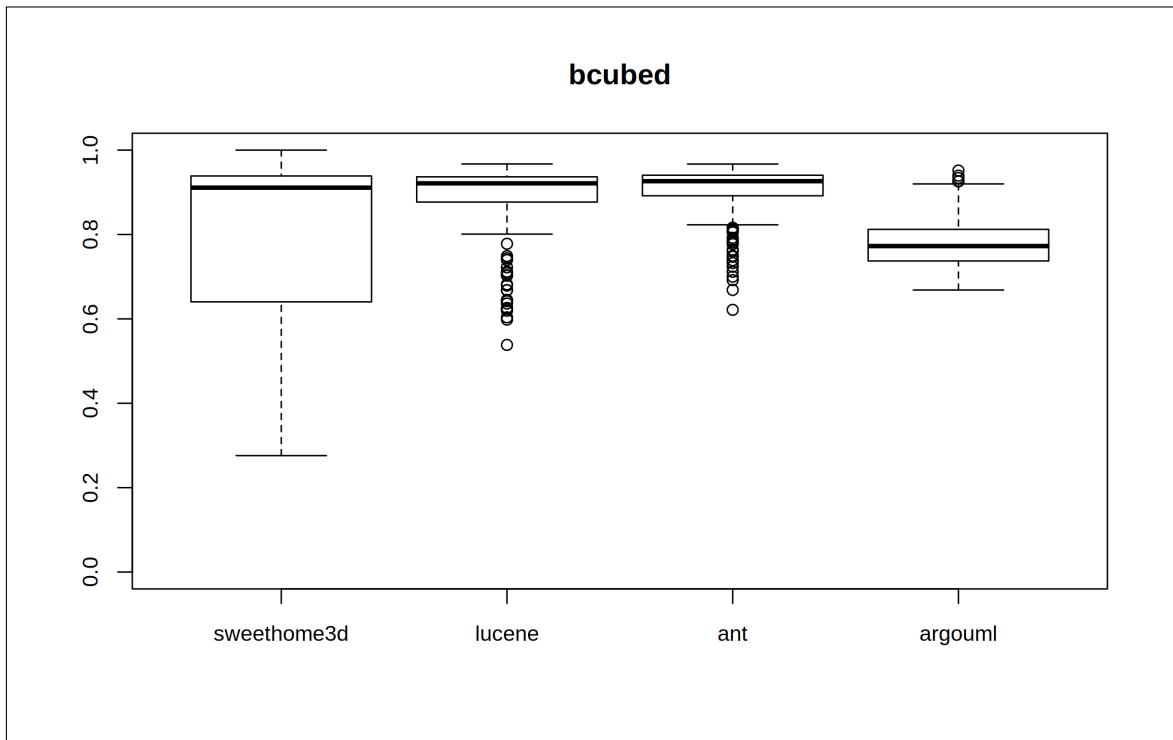
(b) Valores de estabilidad para EdgeSim



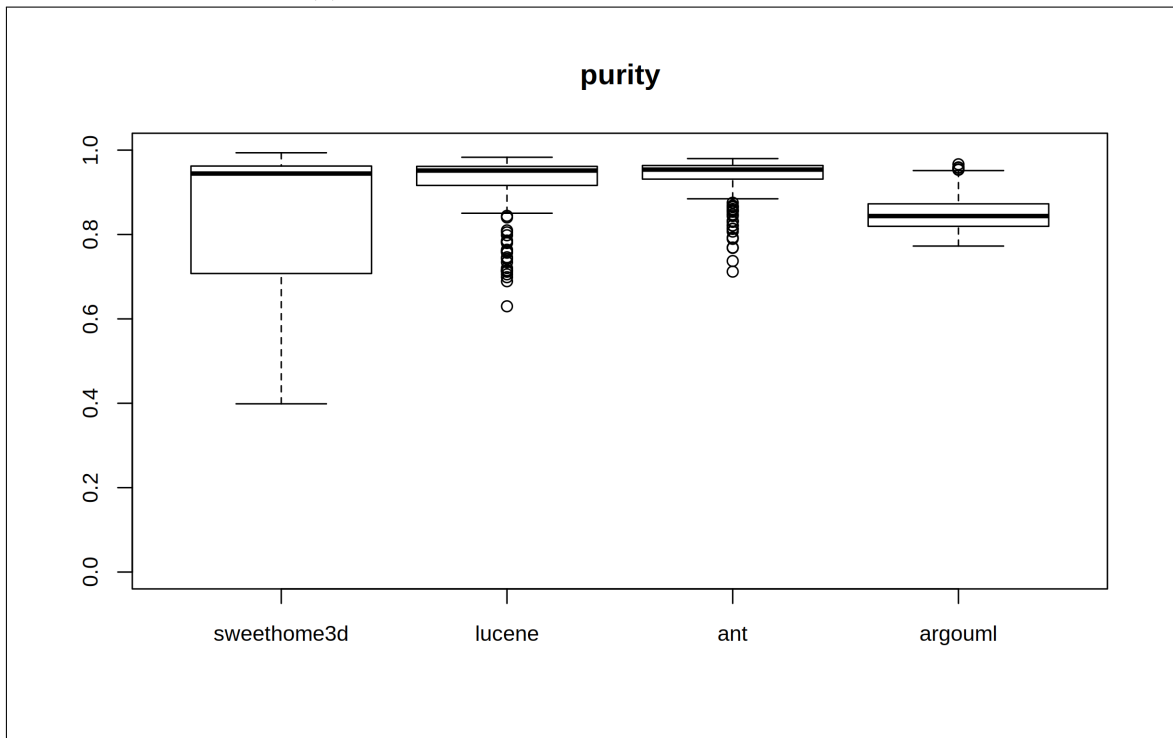
(c) Valores de estabilidade para MeCl



(d) Valores de estabilidade para F1



(e) Valores de estabilidade para B-Cubed-F1

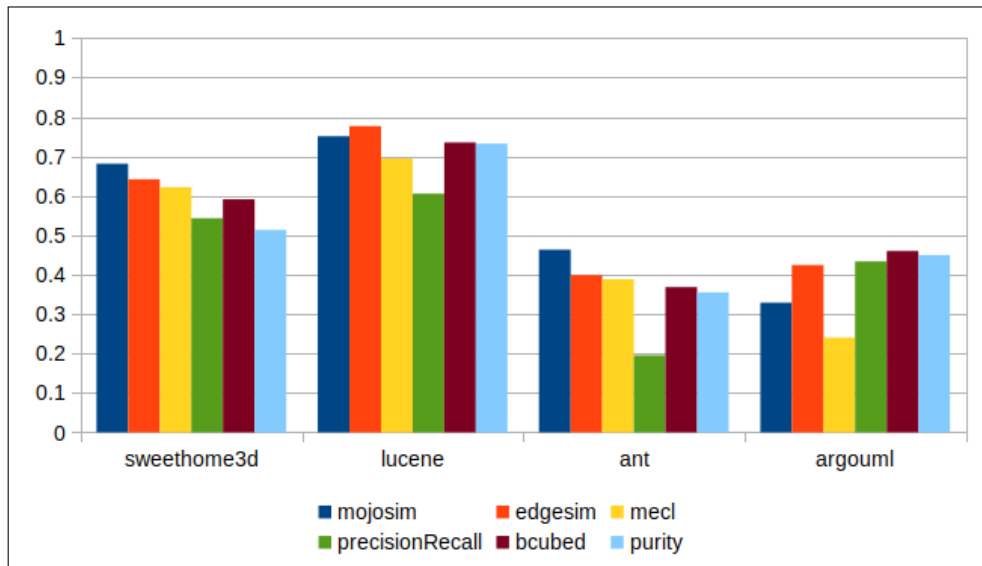


(f) Valores de estabilidade para Pureza

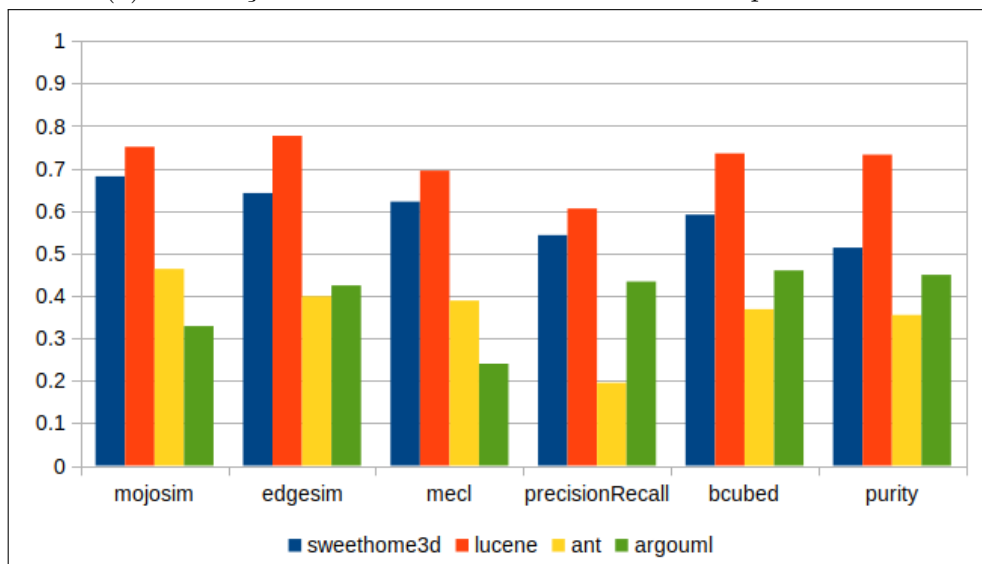
Figura 4.5: Distribuição dos valores de estabilidade por métrica

Por outro lado, é importante comparar também as oscilações da estabilidade com as variações do código-fonte do próprio software em evolução. Neste caso, medimos o delta LOC e o delta de classes. É esperado que a correlação da estabilidade com os deltas seja inversa, ou seja, quanto maiores as modificações sofridas pelos sistemas entre uma semana e outra, menor a estabilidade.

As Figuras 4.6 e 4.7 ilustram os gráficos das correlações das métricas com os valores de estabilidade em relação aos deltas de LOC e classes. Como todos os valores de correlação foram negativos, realizamos a inversão do sinal dos dados de correlação com o intuito de facilitar a compreensão. As correlações de estabilidade mostram que, exceto F1, todas as métricas tiveram correlações altas com ambos os deltas no gráfico por sistema. Contudo, com sistemas menores, as correlações são mais altas com as métricas da área da engenharia de software, e com sistemas maiores com as métricas tradicionais da área de classificação. Já no gráfico por métrica, percebe-se uma tendência de aumento da correlação de sistemas pequenos (SweetHome3D) para sistemas médios (Lucene), com queda posterior já nos médios (Ant) e também nos grandes (ArgoUML).

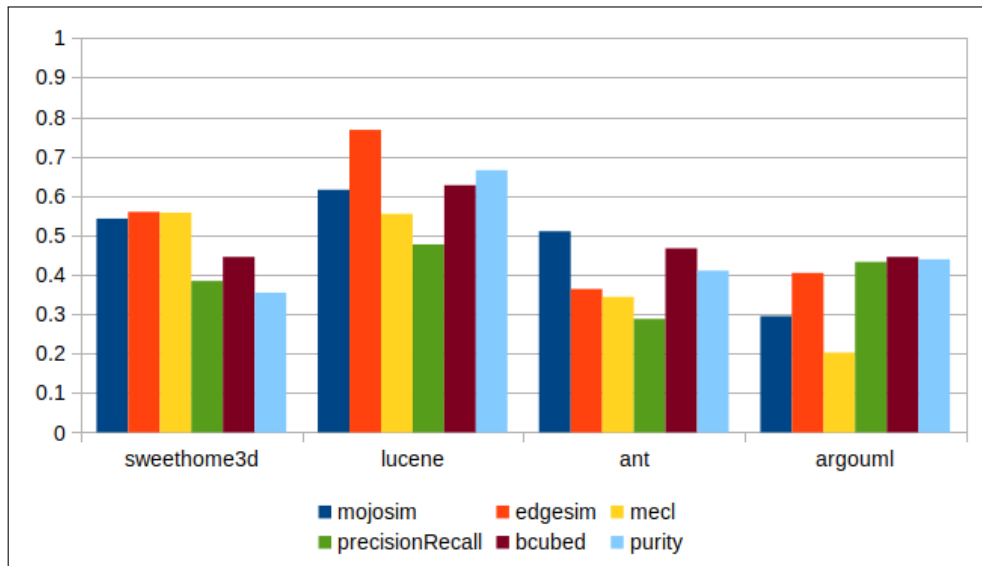


(a) Correlações da estabilidade com delta de LOC por sistema

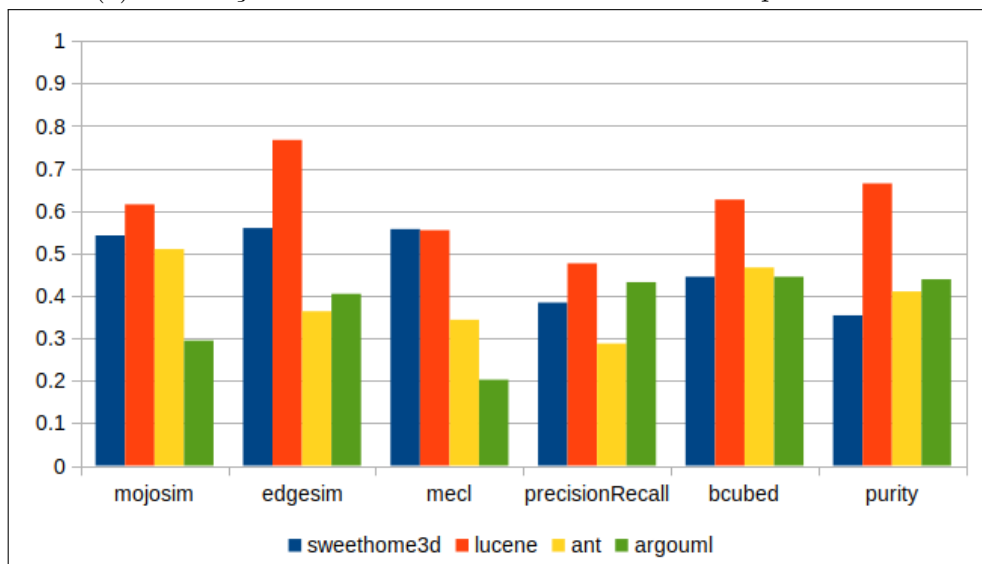


(b) Correlações da estabilidade com delta de LOC por métrica

Figura 4.6: Correlações de estabilidade com delta de LOC



(a) Correlações da estabilidade com delta de classes por sistema



(b) Correlações da estabilidade com delta de classes por métrica

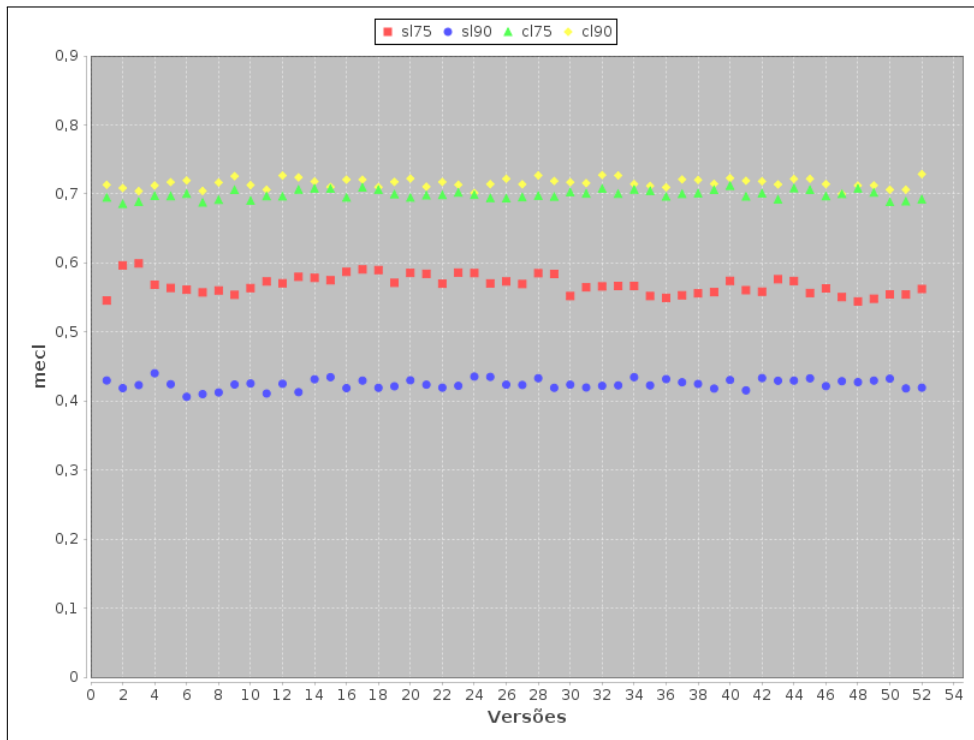
Figura 4.7: Correlações de estabilidade com delta de classes

## 4.2 Resultados dos algoritmos de agrupamento

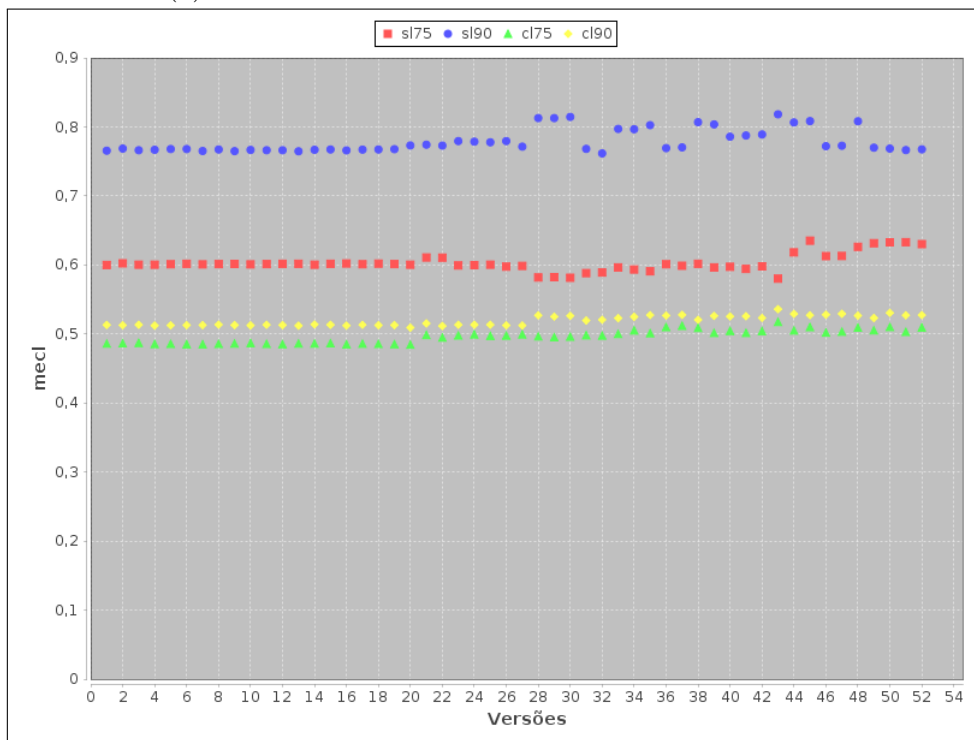
No capítulo de discussão, será feita uma avaliação mais detalhada dos resultados. Mas, em poucas palavras, a métrica MeCl obteve melhores resultados para autoridade e estabilidade. Como este trabalho utilizou modelos de referência para a autoridade e MeCl gerou os melhores resultados, utilizamos a métrica MeCl como base para a avaliação dos algoritmos de agrupamento nesta seção.

### 4.2.1 Autoridade

A Figura 4.8 ilustra os gráficos de autoridade para os sistemas avaliados por meio de *MeCl*. Quanto maior o valor, maior autoridade o algoritmo possui para o dado sistema.

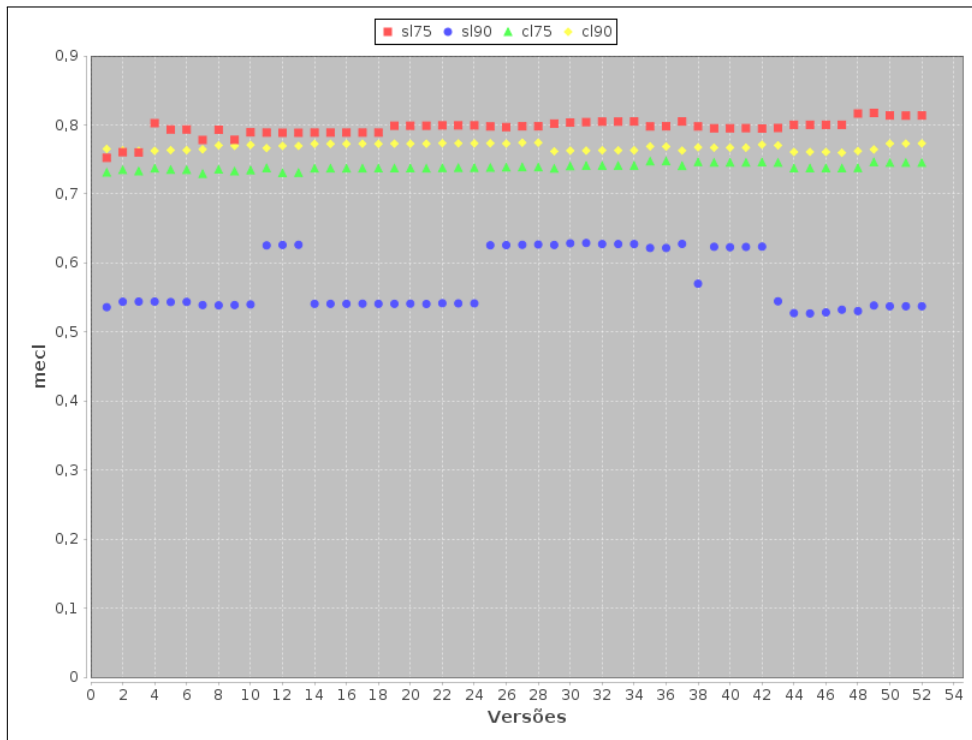


(a) Autoridade do SweetHome3D baseada em MeCl

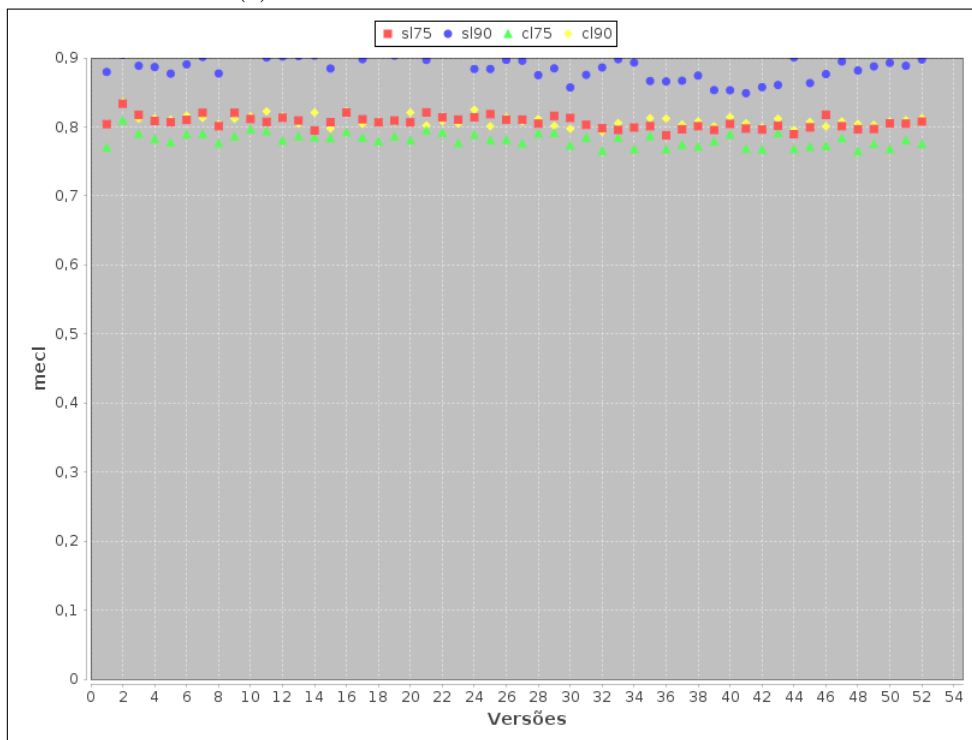


(b) Autoridade do Lucene baseada em MeCl





(c) Autoridade do Ant baseada em MeCl



(d) Autoridade do ArgoUml baseada em MeCl

Figura 4.8: Autoridade dos algoritmos de agrupamento em relação aos sistemas estudados

A Tabela 4.1 mostra os resultados obtidos na aplicação da métrica relativa *Above* para avaliação da autoridade, o que permite detectar os algoritmos com séries de dados com maiores valores. Quanto maior o valor da métrica, maior a autoridade do algoritmo em relação aos demais. Observa-se que o SL90 teve autoridade mais alta para o Lucene e o ArgoUML, CL90, para o Sweethome3D, e o SL75, para o Ant.

Tabela 4.1: Autoridade relativa baseada em MeCl

	<b>Sweethome3D</b>	<b>Lucene</b>	<b>Ant</b>	<b>ArgoUML</b>
<b>SL75</b>	1.00	2.00	<b>2.94</b>	1.33
<b>SL90</b>	0.00	<b>3.00</b>	0.00	<b>3.00</b>
<b>CL75</b>	2.02	0.00	1.00	0.00
<b>CL90</b>	<b>2.98</b>	1.00	2.06	1.67

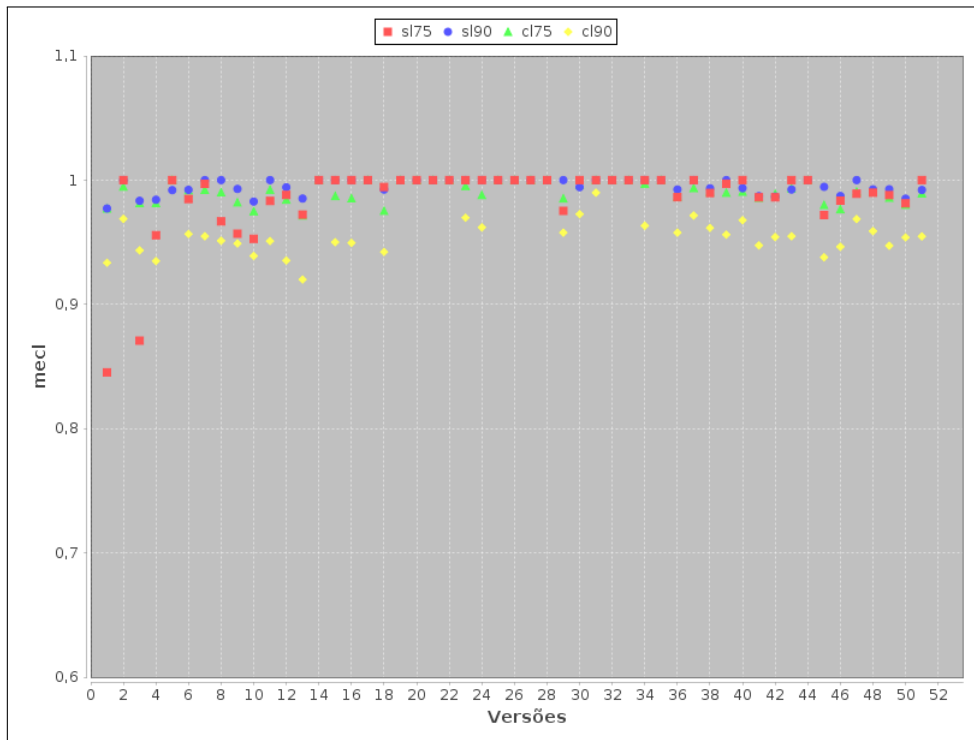
A Tabela 4.2 mostra os resultados obtidos na aplicação da métrica ordinal *HML* para avaliação da autoridade. Para autoridade, o *hm* foi definido como 0, 8 e o *ml* como 0, 5, mesmos valores utilizados por Wu et al. (2005) e por Bittencourt e Guerrero (2009). Pode-se perceber que a boa parte dos scores são *M*, tendo dois scores *L* para o SL90 e CL75 e dois scores *H* para o SL90 e o CL90. Em um cenário ideal, o algoritmo deve conseguir a maior quantidade possível de scores *H*, indicando valores altos e, conseqüentemente, maior autoridade.

Tabela 4.2: Autoridade absoluta baseada em MeCl

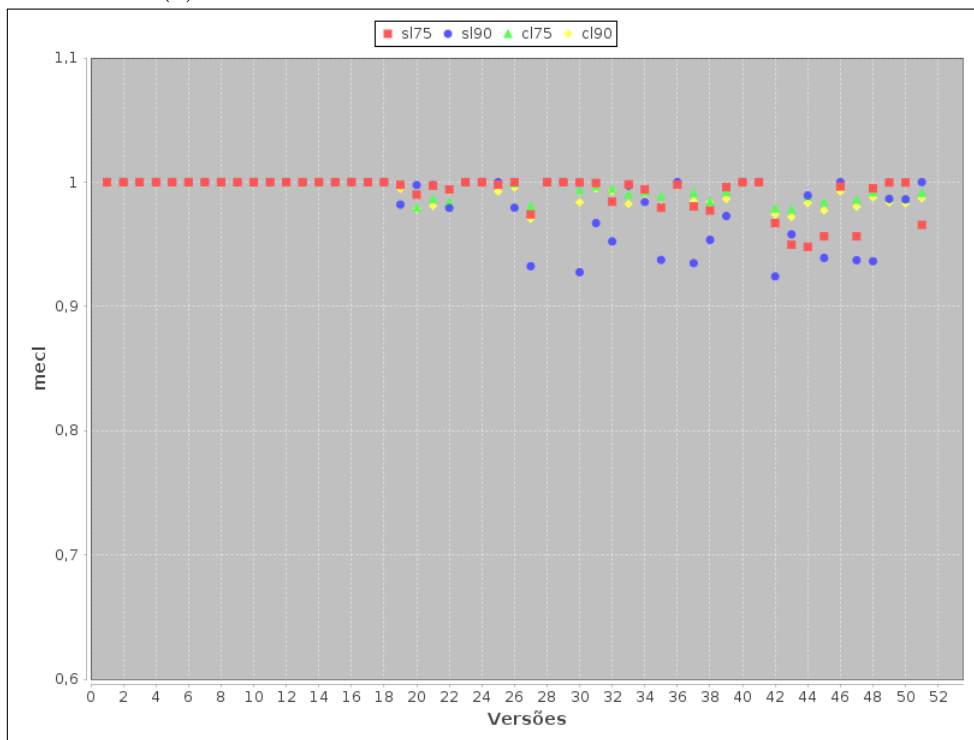
	<b>Sweethome3d</b>	<b>Lucene</b>	<b>Ant</b>	<b>ArgoUML</b>
<b>SL75</b>	M	M	M	M
<b>SL90</b>	L	M	M	H
<b>CL75</b>	M	L	M	M
<b>CL90</b>	M	M	M	H

### 4.2.2 Estabilidade

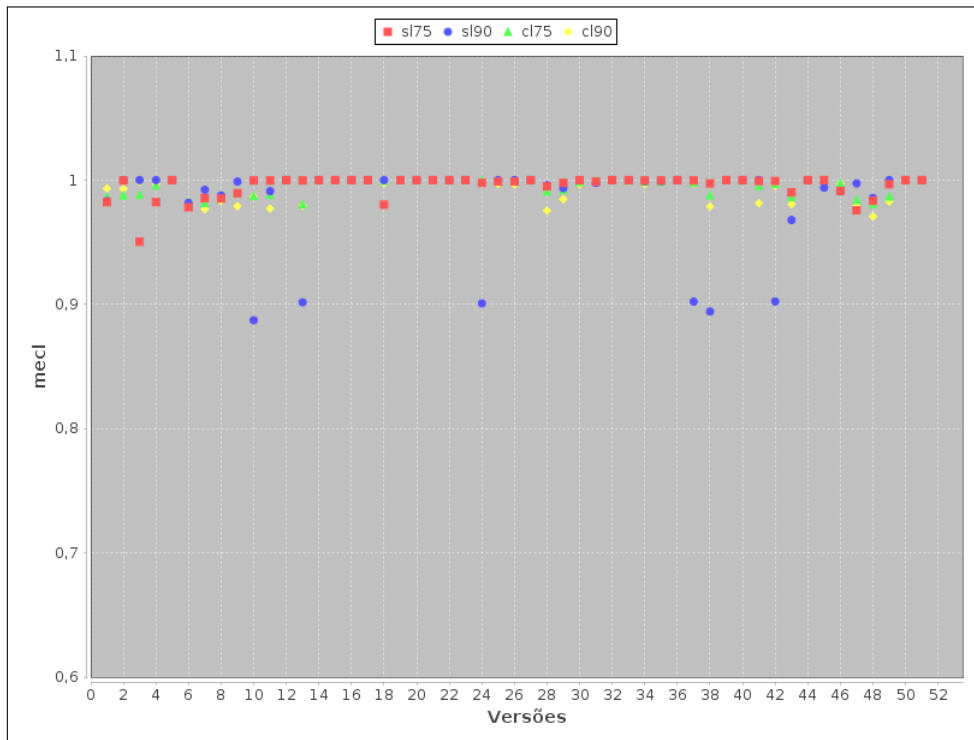
A Figura 4.9 ilustra os gráficos de estabilidade para os sistemas avaliados por meio de *MeCl*. Quanto mais alto forem os valores, melhor a estabilidade do algoritmo para o sistema.



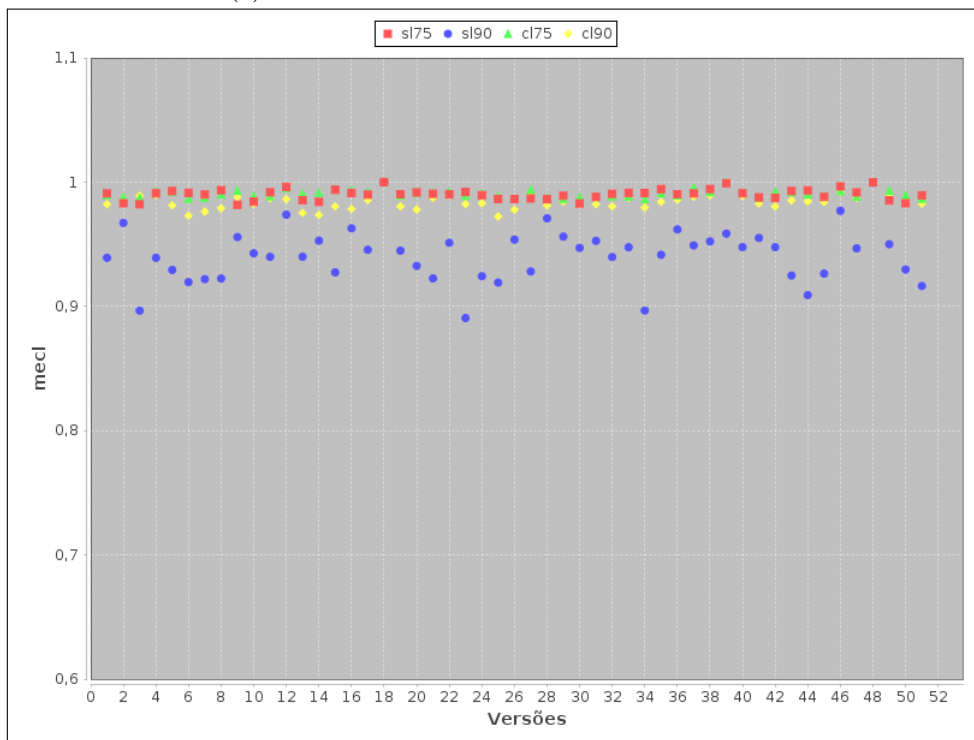
(a) Estabilidade do SweetHome3D baseada em MeCl



(b) Estabilidade do Lucene baseada em MeCl



(c) Estabilidade do Ant baseada em MeCl



(d) Estabilidade do ArgoUML baseada em MeCl

Figura 4.9: Estabilidade dos algoritmos de agrupamento em relação aos sistemas estudados

A Tabela 4.3 mostra os resultados obtidos na aplicação da métrica relativa *Above* para avaliação da estabilidade, o que permite mensurar as diferenças entre os algoritmos. Quanto maior o valor da métrica, maior a estabilidade do algoritmo em relação aos demais. Observa-se que o SL90 obteve estabilidade mais alta para o Sweethome3D e e Ant. Já para o Lucene e ArgoUML, o CL75 apresentou maior estabilidade. Vale ressaltar que, para o Ant, o SL75, SL90 e CL75 apresentaram valores de estabilidade próximos, indicando comportamento similar.

Tabela 4.3: Estabilidade relativa baseada em MeCl

	<b>Sweethome3d</b>	<b>Lucene</b>	<b>Ant</b>	<b>ArgoUML</b>
<b>SL75</b>	1.18	0.94	<b>1.10</b>	2.27
<b>SL90</b>	<b>1.76</b>	0.51	<b>1.12</b>	0.00
<b>CL75</b>	1.10	<b>1.18</b>	<b>1.02</b>	<b>2.41</b>
<b>CL90</b>	0.06	0.53	0.45	1.18

A Tabela 4.4 mostra os resultados obtidos na aplicação da métrica ordinal *HML* para avaliação da estabilidade. Neste caso, o *hm* foi definido como 0,9 e o *ml* como 0,7, mesmos valores utilizados por Wu et al. (2005) e por Bittencourt e Guerrero (2009). Pode-se perceber que todos os algoritmos apresentaram altos valores absolutos de estabilidade.

Tabela 4.4: Estabilidade absoluta baseada em MeCl

	<b>Sweethome3d</b>	<b>Lucene</b>	<b>Ant</b>	<b>ArgoUML</b>
<b>SL75</b>	H	H	H	H
<b>SL90</b>	H	H	H	H
<b>CL75</b>	H	H	H	H
<b>CL90</b>	H	H	H	H

# Capítulo 5

## Discussão

Este capítulo apresenta uma discussão dos resultados obtidos em relação às métricas de similaridade e aos algoritmos de agrupamento aglomerativos.

### 5.1 Sobre as métricas de similaridade

A partir de nossos experimentos, a maioria das métricas de similaridade de agrupamentos, em geral, apresentam bons resultados tanto de autoridade quanto de estabilidade exceto pela F1. Contudo, ao analisarmos de perto a concentração e a dispersão de estabilidade e autoridade, verificamos que os valores de MeCl são melhores ao compararmos as métricas entre si, tanto para autoridade quanto para a estabilidade.

No caso da autoridade, MeCl apresentou concentração de valores maiores ficando entre 0,6 e 0,8 se considerarmos somente as métricas de similaridade. Isto leva a crer que, na existência de modelos de referência, MeCl seria a melhor métrica a ser utilizada. Contudo, se considerarmos a silhueta como a melhor referência de avaliação de agrupamentos, Pureza, B-Cubed-F1 e F1 apresentam uma correlação mais forte entre os dados de autoridade e silhueta. Caso o estudo não tivesse modelos para comparação, a Pureza poderia ser considerada como uma métrica mais adequada.

Para a estabilidade, todas as métricas apresentam valores altos de estabilidade mas o MeCl mostrou concentração alta de pontos perto de 1 em todos os sistemas além da presença dos *outliers*. Neste caso, MeCl seria o melhor candidato para o estudo para a estabilidade.

Ao verificarmos as correlações com os deltas de LOC e classes, identificamos uma possível tendência de maiores correlações para as métricas B-Cubed-F1, Pureza e F1 à medida que os sistemas crescem. Inicialmente todas as métricas mantêm uma alta correlação em torno de 0,7 mas, no ArgoUML, apesar da redução das correlações, as métricas tradicionais da área de classificação conseguiram manter valores um pouco

maiores e próximos entre si quando comparados com as métricas oriundas da área da engenharia de *software* (MojoSim, EdgeSim e MeCl).

Como utilizamos modelos de referência para comparação e MeCl apresentou bons valores de dispersão, iremos discutir os resultados dos algoritmos de agrupamento baseados na métrica MeCl.

Como a silhueta não considera os modelos criados pelos desenvolvedores, e sim a estrutura interna dos *clusters*, a avaliação da qualidade do agrupamento pode não refletir a arquitetura projetada pelos *experts*. A baixa correlação dos valores de autoridade com a silhueta indicam isto. Neste caso, caso hajam modelos para comparação, utilizar MeCl parece ser a melhor opção para capturar a qualidade do agrupamento em relação ao critério de autoridade.

## 5.2 Sobre os algoritmos de agrupamento

De modo geral, os algoritmos hierárquicos aglomerativos apresentam resultados razoáveis em relação a autoridade e estabilidade. Todavia, a escolha de um ponto de corte para o caso dos algoritmos hierárquicos aglomerativos pode influenciar diretamente nos resultados dos agrupamentos gerados, tanto em relação à autoridade quanto para a estabilidade. Caso um modelo arquitetural indique uma quantidade pequena de módulos, o ponto 90 pode se sair melhor nos resultados. No caso da estabilidade, o ponto de corte 75 seria melhor devido às pequenas mudanças que ficam mais evidentes somente quando se aglomera mais as entidades.

Com relação à autoridade, os agrupamentos gerados pela maioria dos algoritmos estabelecem valores médios, produzindo mais scores 'M' em sua maioria para o *HML*. Contudo vale ressaltar dois *scores* 'H' obtido pelos algoritmos SL90 e CL90 para o ArgoUML. Além disto, os dados de autoridade relativa indicam o SL90 como melhor algoritmo para o Lucene e o ArgoUML. Como o *Single Linkage* gera grupos mais isolados, isto pode ter favorecido as características dos modelos do Lucene e do ArgoUML, conseguindo o algoritmo mapear melhor os modelos, gerando valores altos de autoridade. Apesar do SL90 e do CL90 conseguirem *scores* 'H', a série de dados do SL90 gera valores muito mais próximos de 1 do que do CL90 como pode ser visto na Figura 4.8. Todavia, para os outros dois sistemas, o SL90 se torna o pior com valores comparativamente bem mais baixos em relação aos outros. Alguns fatores podem ter influenciado do SL90 ter saído melhor para o Lucene e ArgoUML mas aparecer como pior nos outros dois sistemas no caso da autoridade. Acreditamos que um motivo provável pode ter relação com os modelos utilizados, favorecendo o método *single-linkage* nos pontos de corte mais altos.

Já com relação a estabilidade, os agrupamentos gerados pelos algoritmos aglomerativos estabelecem altos valores de estabilidade, como a métrica de estabilidade absoluta permite inferir, pois obtiveram escores todos 'H' para o *HML*. Além disto,

a estabilidade relativa nos confirma que realmente estes algoritmos estão próximos uns dos outros, apresentando valores de estabilidade próximos. Comparativamente, o algoritmo *SL90* foi mais estável para dois sistemas enquanto o *CL75* foi mais estável para os outros dois. Vale ressaltar a proximidade do *SL75* e *CL75* para o caso do sistema *Ant* com uma mínima diferença de 0,02, do *CL75* com diferença de 0,1 em relação ao *SL90*, e para o *ArgoUML*, distantes entre si por 0,14. Para o Lucene e *ArgoUML*, o *SL90* acabou oscilando bastante ao longo do período de análise e muitas vezes gerando valores comparativamente mais baixos em relação aos outros algoritmos, se saindo pior nestes dois casos.

Geralmente outros trabalhos tomam como base para avaliação o MoJo ou sua versão modificada chamada de MoJoFM. Como tomamos o MeCl como métrica, geralmente nossas conclusões não concordam com os resultados de outros estudos. A avaliação de Wu et. al. (2005) por exemplo utilizou o MoJo como base e apresentou o *SL75* e *SL90* como piores no quesito autoridade, apesar de apresentarem valores altos de estabilidade. Contudo, a visão de diretórios foi utilizada como modelo arquitetural ao invés de uma visão dos arquitetos de *software* como utilizado neste estudo.

### 5.3 Ameaças à validade

Realizamos a correlação dos valores de autoridade com a silhueta. Isto foi feito para que pudéssemos obter uma outra visão sobre a qualidade dos agrupamentos gerados. Realizamos também a correlação dos valores de estabilidade das métricas de similaridade com os deltas. Desta forma, conseguimos uma visão mais fidedigna sobre a captura das mudanças sofridas pelos *softwares* através das métricas de similaridade. Usamos os algoritmos para gerar os dados para ver concentração e dispersão. O ideal seria ter todos os algoritmos possíveis. Porém, devido as limitações de tempo, tomamos algoritmos mais usados como sucedidos na tarefa de recuperação arquitetural de visões modulares.

Não pretendemos que os resultados sejam válidos para outros sistemas. Porém, buscamos utilizar *softwares open-source* e algoritmos conhecidos com o intuito de tornar este estudo o mais reproduzível quanto possível pela comunidade científica.

As métricas foram implementadas para *designs* de sistemas em Java, de acordo com os conceitos contido na literatura. Os deltas representam as mudanças de código sofridas pelas versões semanais dos sistemas ao longo de um ano. Contudo, isto não ocorre para o caso da silhueta em relação a autoridade. Para a avaliação da autoridade, utilizamos um modelo para cada sistema em períodos nos quais os sistemas não sofreram mudanças arquiteturais que comprometessem o uso dos modelos dos desenvolvedores.



# Capítulo 6

## Considerações Finais

Este trabalho avaliou experimentalmente métricas de similaridade de agrupamentos e algoritmos de agrupamento analisados a partir da análise de versões semanais e consecutivas de quatro sistemas de software *open source* no intervalo de um ano de desenvolvimento. Seis métricas de similaridade foram avaliadas como candidatas para medir a qualidade dos agrupamentos gerados pelos algoritmos: *MoJoSim*, *EdgeSim*, *MeCl*, *Pureza*, *F1* e *B-Cubed-F1*. Calculamos as medidas de qualidade dos agrupamentos gerados, tanto em relação a agrupamentos de referência (autoridade) quanto em relação às mudanças nos agrupamentos para versões consecutivas (estabilidade) e analisamos sua concentração e dispersão. Foram extraídos também os dados de variação dos sistemas estudados através dos deltas de linhas de código e classes para correlacionar com a estabilidade. Além disso, correlacionamos os valores de autoridade com os dados da métrica intrínseca de silhueta para melhor fundamentar a escolha da uma métrica de similaridade mais adequada para avaliação dos algoritmos de agrupamento. Finalmente, escolhemos a melhor métrica (no caso, *MeCl*) e a utilizamos para avaliar quatro algoritmos de agrupamento hierárquicos aglomerativos (*SL75*, *SL90*, *CL75* e *CL90*). Utilizamos medidas relativas e absolutas baseadas em séries de dados para avaliar a qualidade dos agrupamentos gerados pelos algoritmos.

Para a autoridade, *MeCl* se destaca das outras métricas seguidas pelo *EdgeSim* (exceto para o *ArgoUML*), depois por *Pureza* (dispersão maior), *MojoSim*, *B-Cubed-F1* (dispersão maior) e *F1* quando as métricas são comparadas entre si. Contudo, na ausência de modelos de referência, a correlação com a medida intrínseca silhueta é mais forte com as métricas *Pureza*, *B-Cubed-F1* e *F1*.

Para a estabilidade, *MeCl* também se destaca com uma concentração de valores altos em todos os casos, mas com a geração de alguns *outliers*, provavelmente capturando mudanças mais bruscas no *software*. Além disto, as correlações com os deltas no código-fonte aumentam do *Sweethome3D* para o *Lucene* e depois ficaram mais baixos tanto para o *Ant* quanto o *ArgoUML*.

Com os melhores resultados de autoridade e estabilidade, utilizamos o MeCl como métrica base para avaliação dos algoritmos.

Em relação aos algoritmos de agrupamento, apesar de apresentarem resultados distintos, nenhum algoritmo supera os outros em todos os aspectos de maneira contundente. Acreditamos que isto ocorre devido aos algoritmos serem da mesma natureza (hierárquicos aglomerativos). Apesar disto, observamos que, tanto para a autoridade quanto para a estabilidade, o SL90 apresentou melhores resultados para dois dos quatro sistemas alvo, sendo melhor no Lucene e ArgoUML no quesito autoridade enquanto que, no critério de estabilidade, apresentou melhores resultados para o SweetHome3D e Ant. Por outro lado, em relação aos valores absolutos de autoridade e estabilidade, observamos que os quatro algoritmos resultam em boa estabilidade (valores próximos de um e com alguns *outliers*) embora a autoridade apresente, de modo geral, valores médios, com exceção do SL90 e do CL90 para o ArgoUML. Isto se explica, de certa forma, pela característica dos algoritmos em impor uma decomposição arquetural e não necessariamente recuperar um modelo proposto pelos desenvolvedores.

Alguns trabalhos usando algoritmos de agrupamento para recuperação arquetural já foram produzidos pela comunidade acadêmica. Entretanto, fazia-se necessária uma análise mais apurada destas técnicas através de trabalhos de avaliação experimental. A verificação da qualidade das métricas de similaridade de agrupamentos realizada nesta dissertação permitiu compreender melhor os pontos fortes e fracos de cada métrica, descrevendo melhor seu comportamento e auxiliando na redução das inconsistências dos resultados encontrados em diversos outros estudos. Além disso, foi realizada ainda uma avaliação dos algoritmos de agrupamento propriamente ditos à luz da métrica MeCl, cujos resultados foram melhores na presença de modelos arquiteturais de referência como uma primeira avaliação mais apurada a partir dos resultados da avaliação das métricas.

Este trabalho contribui para a área de recuperação arquetural de software, através da avaliação de técnicas de agrupamento de software, caracterizando melhor a qualidade das métricas de similaridade de agrupamentos e os algoritmos de agrupamento existentes na literatura. Este trabalho é um passo nesta direção e, com os trabalhos futuros, pode-se avançar no conhecimento sobre o potencial dos algoritmos de agrupamento para recuperação arquetural. Com a melhor avaliação dos algoritmos de agrupamento para recuperação arquetural de software, pode-se implementar ferramentas que auxiliem os profissionais de área, seja na redocumentação de sistemas ou para checar a conformidade entre a arquitetura planejada e a implementação, entre outras possibilidades.

## 6.1 Trabalhos Futuros

Como trabalhos futuros, compreende-se a necessidade de avaliar os algoritmos aglomerativos baseados em dependências estruturais e compará-los com os baseados em recuperação de informação utilizados neste estudo. Além disso, faz-se necessário aumentar a variedade dos sistemas alvo e dos algoritmos de agrupamento, além de adicionar novas métricas de similaridade como a *c2c* e a *a2a*.

# Referências Bibliográficas

- [Amigó et al. 2009] Amigó, E., Gonzalo, J., Artiles, J., e Verdejo, F. (2009). A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information Retrieval*.
- [Anquetil e Lethbridge 1997] Anquetil, N. e Lethbridge, T. (1997). File clustering using naming conventions for legacy systems. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, pp. 2.
- [Anquetil e Lethbridge 1999] Anquetil, N. e Lethbridge, T. C. (1999). Experiments with clustering as a software modularization method. In *Proc. Sixth Working Conf. Reverse Eng.*, pp. 235–255.
- [Bagga e Baldwin 1998] Bagga, A. e Baldwin, B. (1998). Entity-based cross-document coreferencing using the Vector Space Model. In *Proceedings of the 36th annual meeting on Association for Computational Linguistics -*.
- [Bass et al. 2003] Bass, L., Clements, P., e Kazman, R. (2003). *Software Architecture in Practice*, volume 2nd. Addison-Wesley Professional.
- [Bittencourt 2012] Bittencourt, R. A. (2012). *Habilitando a Checagem Estática de Conformidade Arquitetural de Software em Evolução*. PhD thesis, Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande, Campina Grande.
- [Bittencourt et al. 2009] Bittencourt, R. A., Damásio, J. F., Jansen, G., de Almeida Filho, A. T., da Nóbrega Filho, J. M., de Figueiredo, J. C. A., e Guerrero, D. D. S. (2009). Design suite: Towards an open scientific investigation environment for software architecture recovery. *SBQS 2009: Anais do VIII Simpósio Brasileiro de Qualidade de Software*.
- [Bittencourt e Guerrero 2009] Bittencourt, R. A. e Guerrero, D. D. S. (2009). Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pp. 251–254, Washington, DC, USA. IEEE Computer Society.
- [Bowman et al. 1999] Bowman, I. T., Holt, R. C., e Brewster, N. V. (1999). Linux as a Case Study: Its Extracted Software Architecture. *ACM International Conference on Software Engineering*, pp. 555–563.

- [De Lucia et al. 2012] De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., e Panichella, S. (2012). Using IR methods for labeling source code artifacts: Is it worthwhile? In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pp. 193–202.
- [Garcia et al. 2013] Garcia, J., Ivkovic, I., e Medvidovic, N. (2013). A Comparative Analysis of Software Architecture Recovery Techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 486–496.
- [Han et al. 2012] Han, J., Kamber, M., e Pei, J. (2012). *Data Mining: Concepts and Techniques*. Elsevier.
- [Korn e Koutsofios 1999] Korn, J. e Koutsofios, E. (1999). Chava: reverse engineering and tracking of Java applets. *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pp. 314–325.
- [Koschke e Eisenbarth 2000] Koschke, R. e Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. In *Proceedings - IEEE Workshop on Program Comprehension*, volume 2000-January, pp. 201–210.
- [Lutellier et al. 2015] Lutellier, T., Chollak, D., Garcia, J., Rayside, D., Kroeger, R., Tan, L., Rayside, D., Medvidovic, N., e Kroeger, R. (2015). Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 69–78.
- [Mancoridis et al. 1998] Mancoridis, S., Mitchell, B., Rorres, C., Chen, Y., e Gansner, E. (1998). Using automatic clustering to produce high-level system organizations of source code. *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pp. 45–52.
- [Manning et al. 2008] Manning, C. D., Raghavan, P., e Schütze, H. (2008). *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge.
- [Maqbool e Babri 2004] Maqbool, O. e Babri, H. (2004). The weighted combined algorithm: a linkage algorithm for software clustering. *Eighth European Conference on Software Maintenance and Reengineering*, pp. 15–24.
- [Maqbool e Babri 2007] Maqbool, O. e Babri, H. (2007). Hierarchical clustering for software architecture recovery.
- [Mitchell e Mancoridis 2001a] Mitchell, B. S. e Mancoridis, S. (2001a). Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *IEEE International Conference on Software Maintenance, ICSM*, pp. 744–753.
- [Mitchell e Mancoridis 2001b] Mitchell, B. S. e Mancoridis, S. (2001b). Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pp. 93–, Washington, DC, USA. IEEE Computer Society.

- [Pollet et al. 2007] Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cîmpan, S., e Verjus, H. (2007). Towards a process-oriented software architecture reconstruction taxonomy. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 137–148.
- [Terceiro et al. 2010] Terceiro, A., Costa, J., Miranda, J., Meirelles, P., Rios, L. R., Almeida, L., Chavez, C., e 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 e Holt 1999] Tzerpos, V. e Holt, R. C. (1999). MoJo: a distance metric for software clusterings. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pp. 187–193.
- [Tzerpos e Holt 2000] Tzerpos, V. e Holt, R. C. (2000). On the stability of software clustering algorithms. In *Proceedings - IEEE Workshop on Program Comprehension*, volume 2000-January, pp. 211–218.
- [Wen e Tzerpos 2003] Wen, Z. e Tzerpos, V. (2003). An optimal algorithm for MoJo distance. In *Proceedings - IEEE Workshop on Program Comprehension*, volume 2003-May, pp. 227–235.
- [Wiggerts 1997] Wiggerts, T. A. (1997). Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pp. 33, Washington, DC, USA. IEEE Computer Society.
- [Wohlin et al. 2012] Wohlin, C., Runeson, P., Höst, M., Magnus C., O., Regnell, B., e Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg, 1 edition.
- [Wu et al. 2005] Wu, J., Hassan, A. E., e Holt, R. C. (2005). Comparison of Clustering Algorithms in the Context of Software Evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 525–535, Washington, DC, USA. IEEE Computer Society.
- [Zaki e Meira 2014] Zaki, M. J. e Meira, W. J. (2014). *Data Mining and Analysis: fundamental concepts and algorithms*. Cambridge University Press.

# Apêndice A

## Modelos

Neste Apêndice, as visões modulares de alto nível e o mapeamento das entidades de código-fonte para os módulos nestas visões são detalhados. Para cada sistema, os módulos são descritos, seguidos pelo mapeamento na forma de expressões regulares. Finalmente, as relações autorizadas entre módulos são expressas como relações binárias entre os módulos fonte e destino.

## A.1 SweetHome3D

```

# modules
sweetHome3DModel
sweetHome3DTools
sweetHome3DPlugin
sweetHome3DViewController
sweetHome3DSwing
sweetHome3DJava3D
sweetHome3DIO
sweetHome3DApplet
sweetHome3DApplication

# mapping
# <high_level_module> <regular_expression>
sweetHome3DModel com.eteks.sweethome3d.model.*
sweetHome3DTools com.eteks.sweethome3d.tools.*
sweetHome3DPlugin com.eteks.sweethome3d.plugin.*
sweetHome3DViewController com.eteks.sweethome3d.viewcontroller.*
sweetHome3DSwing com.eteks.sweethome3d.swing.*
sweetHome3DJava3D com.eteks.sweethome3d.j3d.*
sweetHome3DIO com.eteks.sweethome3d.io.*
sweetHome3DApplet com.eteks.sweethome3d.applet.*
sweetHome3DApplication com.eteks.sweethome3d(?!(.model|.tools|.plugin|.viewController|.j3d|.io|.applet)).*

# relations
# <source_module> <target_module>
sweetHome3DTools sweetHome3DModel
sweetHome3DPlugin sweetHome3DModel
sweetHome3DPlugin sweetHome3DTools
sweetHome3DViewController sweetHome3DModel
sweetHome3DViewController sweetHome3DTools
sweetHome3DViewController sweetHome3DPlugin
sweetHome3DJava3D sweetHome3DModel
sweetHome3DJava3D sweetHome3DTools
sweetHome3DSwing sweetHome3DModel
sweetHome3DSwing sweetHome3DTools
sweetHome3DSwing sweetHome3DPlugin
sweetHome3DSwing sweetHome3DViewController
sweetHome3DSwing sweetHome3DJava3D
sweetHome3DIO sweetHome3DModel
sweetHome3DIO sweetHome3DTools
sweetHome3DApplet sweetHome3DModel
sweetHome3DApplet sweetHome3DTools
sweetHome3DApplet sweetHome3DPlugin
sweetHome3DApplet sweetHome3DViewController
sweetHome3DApplet sweetHome3DJava3D
sweetHome3DApplet sweetHome3DSwing
sweetHome3DApplet sweetHome3DIO
sweetHome3DApplication sweetHome3DModel
sweetHome3DApplication sweetHome3DTools
sweetHome3DApplication sweetHome3DPlugin
sweetHome3DApplication sweetHome3DViewController
sweetHome3DApplication sweetHome3DJava3D
sweetHome3DApplication sweetHome3DSwing
sweetHome3DApplication sweetHome3DIO

```



## A.2 Ant

```

# modules
optional
compilers
condition
rmic
cvslib
email
repository
taskdefs
listener
types
ant
ant.util
zip
tar
mail
bzip2

# mapping
# <high_level_module> <regular_expression>
optional org.apache.tools.ant.taskdefs.optional.*
compilers org.apache.tools.ant.taskdefs.compilers.*
condition org.apache.tools.ant.taskdefs.condition.*
rmic org.apache.tools.ant.taskdefs.rmic.*
cvslib org.apache.tools.ant.taskdefs.cvslib.*
email org.apache.tools.ant.taskdefs.email.*
repository org.apache.tools.ant.taskdefs.repository.*
taskdefs org.apache.tools.ant.taskdefs?(!(.optional|.compilers|.condition|.rmic|.cvslib|.email|.repository)).*
listener org.apache.tools.ant.listener.*
types org.apache.tools.ant.types.*
ant org.apache.tools.ant?(!(.taskdefs|.listener|.types|.util|.filters|.helper|.input|.launch|.loader|.dispatch|.property)).*
ant.util org.apache.tools.ant.util.*
zip org.apache.tools.zip.*
tar org.apache.tools.tar.*
mail org.apache.tools.mail.*
bzip2 org.apache.tools.bzip2.*

# relations
# <source_module> <target_module>
optional taskdefs
optional listener
optional types
optional ant
optional ant.util
optional zip
optional tar
optional mail
optional bzip2
compilers taskdefs
compilers listener
compilers types
compilers ant
compilers ant.util
compilers zip
compilers tar
compilers mail
compilers bzip2
condition taskdefs
condition listener
condition types
condition ant
condition ant.util
condition zip
condition tar
condition mail
condition bzip2
rmic taskdefs
rmic listener
rmic types
rmic ant
rmic ant.util
rmic zip
rmic tar
rmic mail
rmic bzip2
cvslib taskdefs
cvslib listener
cvslib types
cvslib ant
cvslib ant.util
cvslib zip
cvslib tar
cvslib mail
cvslib bzip2
email taskdefs
email listener
email types
email ant

```

---

```
email ant.util
email zip
email tar
email mail
email bzip2
repository taskdefs
repository listener
repository types
repository ant
repository ant.util
repository zip
repository tar
repository mail
repository bzip2
taskdefs listener
taskdefs types
taskdefs ant
taskdefs ant.util
taskdefs zip
taskdefs tar
taskdefs mail
taskdefs bzip2
listener ant
listener ant.util
listener zip
listener tar
listener mail
listener bzip2
types ant
types ant.util
types zip
types tar
types mail
types bzip2
ant zip
ant tar
ant mail
ant bzip2
ant.util zip
ant.util tar
ant.util mail
ant.util bzip2
```

## A.3 Lucene

```
# modules
queryparser
search
index
store
analysis
util
document

# mapping
# <high_level_module> <regular_expression>
queryparser org.apache.lucene.queryParser.*
search org.apache.lucene.search.*
index org.apache.lucene.index.*
store org.apache.lucene.store.*
analysis org.apache.lucene.(analysis|collation).*
util org.apache.lucene.(util|message).*
document org.apache.lucene.document.*

# relations
# <source_module> <target_module>
queryparser search
queryparser index
queryparser document
queryparser util
search index
search analysis
search document
search util
index store
index analysis
index document
index util
analysis document
analysis util
document util
util document
```

## A.4 ArgoUML

```

# modules
application
diagrams
notation
explorer
codeGeneration
reverseEngineering
persistence
profile
help
moduleLoader
gui
model
internationalization
taskManagement
configuration
swingExtensions
ocl
critics
javaCodeGeneration

# mapping
# <high_level_module> <regular_expression>
application org.argouml.application.*
diagrams org.argouml.uml.diagram.*
notation org.argouml.notation.*
explorer org.argouml.ui.explorer.*
codeGeneration org.argouml.language.*
reverseEngineering org.argouml.uml.reveng.*
persistence org.argouml.persistence.*
profile org.argouml.profile.*
help org.argouml.help.*
moduleLoader org.argouml.moduleloader|org.argouml.application.modules|org.argouml.application.api
gui org.argouml.ui.*
model org.argouml.model.*
internationalization org.argouml.i18n.*
taskManagement org.argouml.taskgmt.*
configuration org.argouml.configuration.*
swingExtensions org.argouml.swingext.*
ocl org.argouml.ocl.*
critics org.argouml.cognitive.*
javaCodeGeneration org.argouml.language.java.*

# relations
# <source_module> <target_module>
application diagrams
application notation
application explorer
application codeGeneration
application reverseEngineering
application persistence
application profile
application help
application moduleLoader
application gui
application model
application internationalization
application taskManagement
application configuration
application swingExtensions
application ocl
application critics
application javaCodeGeneration
diagrams notation
diagrams gui
diagrams model
diagrams internationalization
diagrams taskManagement
diagrams configuration
diagrams swingExtensions
notation model
notation internationalization
notation taskManagement
notation configuration
notation swingExtensions
explorer gui
explorer model
explorer internationalization
explorer taskManagement
explorer configuration
explorer swingExtensions
codeGeneration moduleLoader
codeGeneration model
codeGeneration internationalization
codeGeneration taskManagement
codeGeneration configuration
codeGeneration swingExtensions

```

---

```
reverseEngineering model
reverseEngineering internationalization
reverseEngineering taskManagement
reverseEngineering configuration
reverseEngineering swingExtensions
persistence model
persistence internationalization
persistence taskManagement
persistence configuration
persistence swingExtensions
profile model
profile internationalization
profile taskManagement
profile configuration
profile swingExtensions
help model
help internationalization
help taskManagement
help configuration
help swingExtensions
moduleLoader model
moduleLoader internationalization
moduleLoader taskManagement
moduleLoader configuration
moduleLoader swingExtensions
gui internationalization
gui taskManagement
gui configuration
gui swingExtensions
javaCodeGeneration codeGeneration
javaCodeGeneration reverseEngineering
javaCodeGeneration moduleLoader
javaCodeGeneration model
ocl moduleLoader
ocl model
critics moduleLoader
critics model
```