Análise de Grandes Grafos
2024/25 – 2º Semestre

Relatório Trabalho Final
Dependências do JUnit

Trabalho realizado por:
Eduardo Vieira - 65361
Gustavo Feio - 65714
Ivo Lopes - 65511

Com apoio dos docentes:
Pedro Mota
Maria Isabel Gomes

Departamento Matemática
NOVA FCT
Portugal
Junho de 2025

1 Introdução

Este trabalho aborda a construção de um grafo e análise de algumas das suas características relevantes. Para esse fim, escolhemos como rede as dependências internas do JUnit [4], uma framework bastante útil para testes unitários para aplicações Java [3].

Esta análise é importante por diversas razões. Primeiramente permite-nos visualizar mais claramente a arquitetura das classes desta framework e a forma como se relacionam, e permite-nos identificar, por exemplo, ciclos que podem dificultar a manutenção e/ou evolução da framework.

Além disso, o grafo de dependências possibilita a identificação de módulos altamente acoplados, que podem representar pontos críticos do sistema. Módulos com muitas dependências de entrada tornam-se mais difíceis de modificar, pois qualquer alteração pode ter impactos significativos em outras partes da framework. Por outro lado, módulos com muitas dependências de saída revelam um elevado grau de dependência de outros componentes, o que reduz a sua autonomia e reutilização.

Os dados usados na construção do grafo e subsequente análise foram retirados de [1] e podem ser obtidos tanto em lista de incidências de arcos de entrada, como em lista de incidências de arcos de saída. Estes foram convertidos para ficarem sob a forma de matriz de adjacências com a ajuda de um script, que foi posteriormente passada para o código fonte `R` [2], no qual toda a análise do grafo foi feita.

2 Contexto

O JUnit é a ferramenta de bateria de testes mais popular da linguagem de programação Java e é considerada largamente como o standard da indústria.

Na prática são chamado métodos singulares e comparado o resultado com um valor esperado préviamente, que terá de ser calculado à mão ou por uma ferramenta similar ao programa. São executado diversos testes e no final é indicado quais dos métodos retornaram valores diferentes do esperado. Um exemplo prático do JUnit pode ser encontrado na Figura 1.

3 Grafo de Dependências

O grafo obtido encontra-se na Figura 2.

Este é um grafo direcionado, visto que se trata de uma rede de dependências, em que cada nó representa uma classe e cada aresta uma dependência entre estas, nomeadamente por utilização de funções implementadas nelas.

A direcionalidade do grafo reflete a relação de uso/importação entre as classes, isto é, se a classe A depende da classe B, então existe uma aresta orientada de A para B. Desta forma, é possível mapear o fluxo de dependências e compreender melhor a organização interna da framework.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MathPowTest {
    @Test
    void testPowerPositiveExponent() {
        assertEquals(8.0, Math.pow(2, 3));
    }
}
```

Figura 1: Exemplo Java

Figura 2: JUnit - Grafo de Dependências

4 Análise do Grafo

4.1 Grau (Degree)

A primeira medida analisada do grafo, e talvez uma das mais importantes, é o número de dependências diretas de cada classe, ou seja, no contexto de grafos, o seu grau. A representação visual resultante na Figura 3 facilita a identificação imediata de módulos críticos, permitindo distinguir entre os pacotes mais isolados e aqueles que se encontram fortemente conectados dentro da rede.

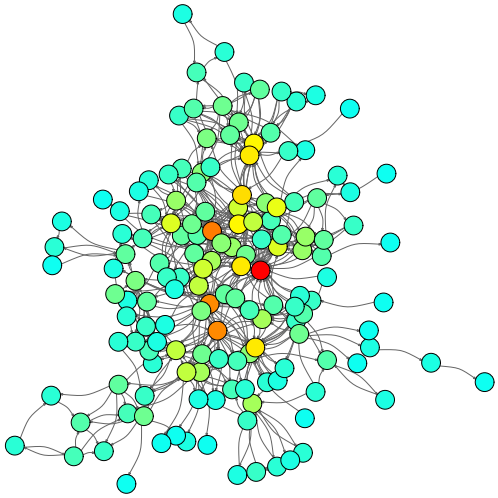

Figura 3: Visualização dos Graus

Nós com grau elevado foram destacados com cores mais intensas, uma vez que representam pontos centrais e críticos da arquitetura. Alterações nestes pacotes podem ter impacto em cascata sobre outros, o que aumenta o risco e a complexidade da manutenção e atualização. Por outro lado, nós com grau reduzido indicam módulos mais independentes, geralmente mais fáceis de testar e reutilizar.

No contexto do JUnit, esta abordagem é particularmente relevante, pois permite identificar quais partes da framework têm um papel fundamental e quais atuam como extensões específicas. Tal distinção é essencial para garantir a modularidade da framework, evitando que funcionalidades de uso específico dependam fortemente de componentes centrais ou vice-versa.

Assim, a coloração por grau não só facilita a leitura visual do grafo, como também fornece dados sobre a dependência e a importância relativa de cada

pacote dentro da rede.

4.2 Centralidade

A análise de centralidade é especialmente importante, pois permite identificar classes que, embora possam não ser centrais em termos de dependência direta, são importantes para manter a coerência e a integração entre diferentes componentes.

Além disso, esta métrica ajuda a distinguir entre nós "populares"(com alto grau) e nós "estratégicos", contribuindo para uma compreensão mais refinada da arquitetura. Em projetos de larga escala como o JUnit, esta distinção pode ser crucial para decisões de refatoração, modularização ou otimização do desempenho da framework.

4.2.1 Eigenvalue (Valor Próprio)

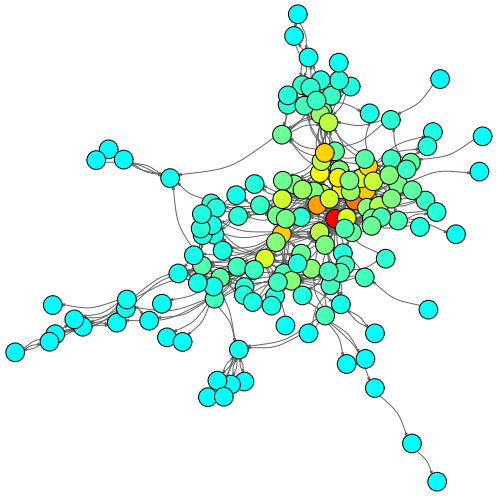

Figura 4: Visualização de Eigenvalue

O Eigenvalue, cujo grafo está na Figura 4, vai indentificar a importância estrutural de cada classe. Maior eigenvalue significará classes que serão mais amplamente importadas e importada ela também por outras classes importantes. Poderá ser util para identificar classes críticas e saber onde focar mais os testes.

4.2.2 Katz

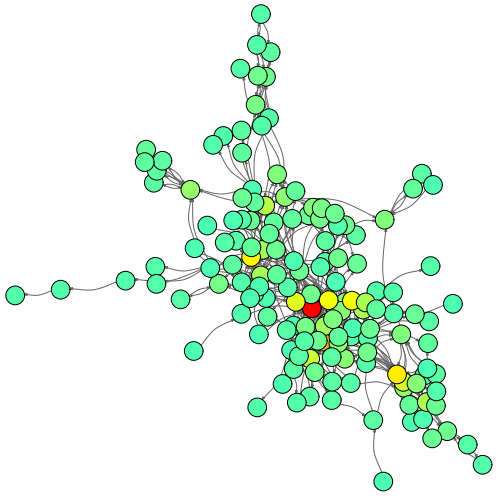

Figura 5: Visualização de Katz

A medida Katz, representada na Figura 5, seguirá o exemplo do EigenValue e indentificará classes importantes.

4.2.3 PageRank

Por fim, a medida Page rank, representado na Figura 6, identifica também por si classes importantes, tal como Katz 4.2.2 e Eigenvalue 4.2.1.

4.2.4 Intermediação (Betweenness)

Outra métrica de centralidade considerada na análise do grafo de dependências foi a centralidade de intermediação. Esta medida avalia a frequência com que um nó aparece nos caminhos mínimos entre pares de outros nós. Isto é, indica o quão intermediário um pacote é na comunicação ou ligação entre diferentes partes da framework.

Na Figura 7, os nós foram coloridos de acordo com o valor da sua intermediação, com cores mais intensas indicando valores mais elevados, ou seja, módulos que funcionam como pontes críticas entre diferentes partes da rede. Estes módulos não têm necessariamente o maior número de conexões ou grau, mas desempenham um papel estratégico na conectividade global da estrutura.

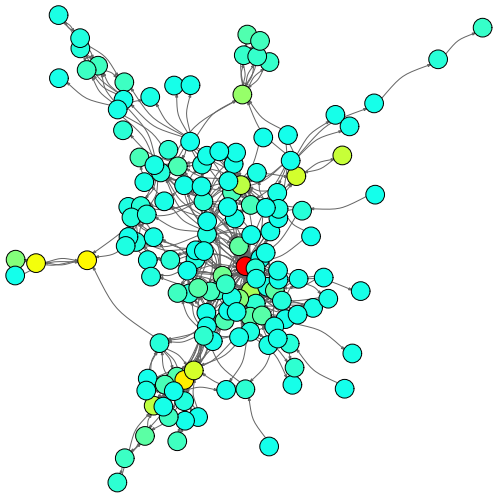

Figura 6: Visualização de PageRank

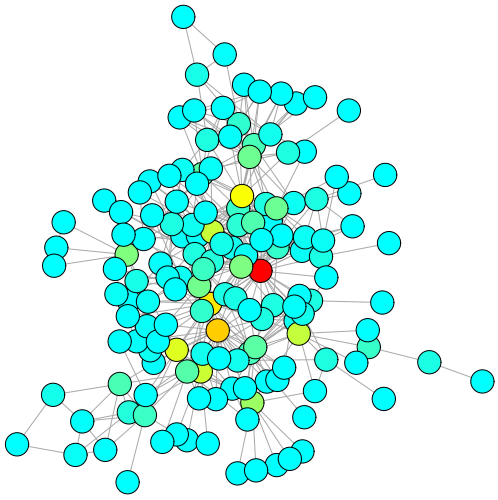

Figura 7: Visualização da Intermediação

Uma classe com alta centralidade de intermediação, se modificada ou removida, pode eliminar caminhos importantes entre outros módulos, afetando a integridade da framework como um todo. Assim, a visualização baseada na intermediação complementa a análise de grau, permitindo uma visão mais completa dos pontos estruturais sensíveis da rede de dependências.

4.3 Diâmetro (Diameter)

O diâmetro da rede ilustra a maior distância entre quaisquer dois nós no grafo de dependências do JUnit. O diâmetro representa o caminho mais longo dentro da rede entre dois módulos conectados, sendo um indicador da profundidade estrutural da framework.

No JUnit, o maior diâmetro encontrado foi de 9. Esta informação permite identificar módulos periféricos, que se encontram mais distantes do núcleo do sistema, e também compreender o grau de propagação de uma alteração — ou seja, até onde uma dependência pode afetar o restante da rede.

No contexto desta análise, um diâmetro muito elevado pode sinalizar complexidade excessiva ou baixa coesão entre partes da framework, enquanto um diâmetro mais curto tende a indicar melhor integração e modularidade.

4.4 Modularidade (Modularity)

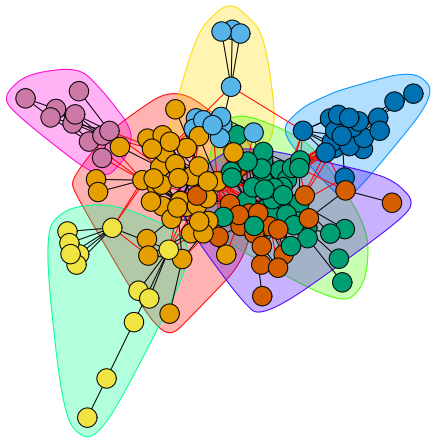

Figura 8: Visualização da Modularidade

A imagem na Figura 8 ilustra a divisão do grafo em comunidades ou subgrupos coesos, onde os nós dentro de um mesmo grupo estão mais densamente

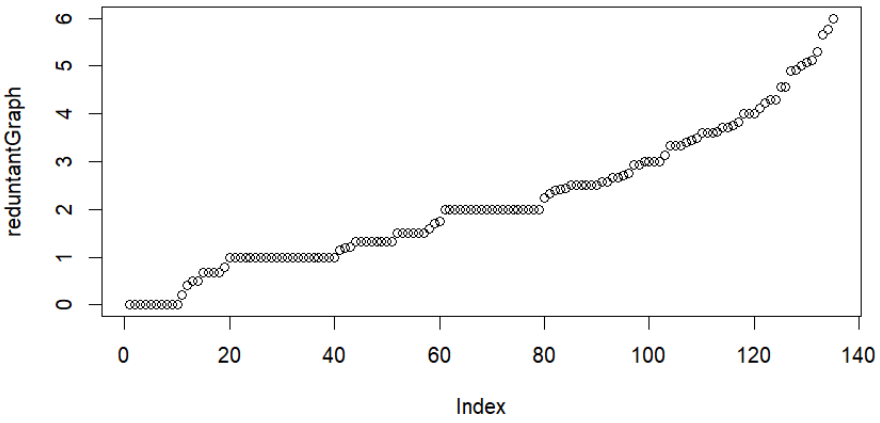

Figura 9: Gráfico de Redundância

conectados entre si do que com nós de outros grupos. Cada comunidade é representada por uma cor distinta, permitindo observar como a arquitetura do JUnit se organiza em termos de blocos funcionais ou temáticos.

A modularidade é uma métrica fundamental em redes complexas, pois reflete o grau de separação funcional entre os componentes. Uma modularidade elevada sugere que a framework está bem estruturada em termos de coesão interna e baixa acoplagem externa, características desejáveis numa arquitetura eficiente, visto que tornam o desenvolvimento de funcionalidades atuais e novas mais fácil de implementar.

Além disso, quantifica o nível de transitividade. Deste modo, a modularidade representa o quanto determinados pacotes dependem uns dos outros.

4.5 Redundância (Redundancy)

A análise de redundância, representada na Figura 9, mostra a existência de caminhos alternativos entre os módulos no grafo, ou seja, em que medida a rede mantém conectividade mesmo na ausência de determinados nós ou arestas.

Em termos de arquitetura de software, a redundância pode ter duas interpretações:

- Quando moderada, é positiva, pois garante resiliência da estrutura — a remoção de um módulo não quebra a comunicação entre os restantes.
- Quando excessiva, pode indicar dependências duplicadas ou desnecessárias, que aumentam a complexidade sem aumentar valor.

No caso do JUnit, observamos que há um conjunto de nós — por volta de um quarto da rede — que apresentam um grau de redundância mais elevado. Isto significa que apresentam dependências potencialmente redundantes que poderiam ser simplificadas.

5 Conclusões

A análise do grafo de dependências do JUnit revelou-se uma abordagem eficaz para compreender a arquitetura interna desta framework muito utilizada. A representação em grafo permitiu visualizar com clareza as relações entre classes e identificar padrões estruturais que, de outro modo, poderiam passar despercebidos numa inspeção apenas textual ou documental.

Através do estudo das diferentes de grafos — como o grau, a centralidade de intermediação, o diâmetro, a modularidade e a redundância — foi possível avaliar a importância relativa das classes dentro da rede, tanto em termos de conectividade direta quanto de papel estratégico na comunicação entre os demais componentes.

Estas análises forneceram informações importantes, como:

- A identificação de módulos centrais e críticos, cuja modificação pode impactar grande parte da framework;
- A deteção de estruturas modulares, que favorecem a escalabilidade e a manutenção;
- A evidência de módulos de intermediação, cuja posição na rede os torna fundamentais para a coesão do sistema;
- A avaliação da robustez estrutural da rede, com base na existência ou não de caminhos alternativos (redundância);
- E a compreensão da profundidade e dispersão das dependências, por meio do diâmetro da rede.

Estas métricas não só permitem avaliar a qualidade da arquitetura do JUnit como a de qualquer outro sistema. A aplicação destes conceitos ao desenvolvimento de software dá-nos uma maior clareza estrutural, permitindo assim garantir uma arquitetura de software robusta.

Num todo, consideramos que as técnicas de análise de grafos descritas neste relatório revelaram-se úteis para a avaliação de um caso prático e real de uma rede complexa.

Referências

[1] Dan Amlund Thomsen: Adjacency matrix code visualizer - junit 4.11. `https://danamlund.dk/adjmatrix/junit-4.11.html` (2019), accessed: 2025-05-30

[2] Eduardo Vieira and Gustavo Feio and Ivo Lopes: Github repository. https://github.com/Ivo-Lopes-65511/AGG-PROJ.git (2025)

[3] Oracle Corporation: Java. https://www.java.com (1995), accessed on: 2025-06-05

[4] The JUnit Team: JUnit 4. https://junit.org/junit4/ (2021), accessed: 2025-05-30