Trabalho Pr atico
Ferramenta de Processamento de Dados com Grafos

Davi Augusto Dias Soares[1], Pedro H. Morais Marques[2], Luisa C. de Paula Lara Silva[3], Luiz Filipe Nery[4] , Victor Lucas Tornelli[5]

[1]Instituto de Ciencias Exatas – Pontificia Universidade Católica (PUC-MG)

{dadsoares, pedro.marques.1265848}@sga.pucminas.br

{lcplsilva, luiznery, victor.tornelli}@sga.pucminas.br

Abstract. *Sistemas modernos de software, como gerenciadores de pacotes, orquestradores de serviços e pipelines de integração contínua, frequentemente lidam com tarefas interdependentes cuja ordem de execução precisa ser rigorosamente respeitada. A presença de ciclos de dependência entre essas tarefas representa um risco à execução correta, podendo causar falhas ou comportamentos indesejados. Este trabalho apresenta uma ferramenta computacional desenvolvida em Java, baseada na teoria dos grafos, que detecta conflitos de dependência e gera uma ordenação segura de execução por meio de algoritmos clássicos, como detecção de ciclos (Tarjan) e ordenação topológica (Kahn). O sistema inclui uma interface gráfica interativa, suporte à leitura de arquivos JSON, e visualização do grafo com destaque para ciclos detectados. Experimentos com dados reais do dataset "Services Dependency Graphs for Web Services Composition" demonstram a eficácia e desempenho da ferramenta em diferentes cenários. A solução proposta contribui para o diagnóstico e resolução de conflitos em sistemas de execução automatizada, promovendo confiabilidade e escalabilidade.*

1. Introdução

A disciplina de Teoria dos Grafos é uma das bases fundamentais da ciência da computação e da engenharia de software. Sua aplicação permite modelar, analisar e resolver problemas complexos por meio de estruturas abstratas compostas por vértices e arestas, representando entidades e relações entre elas. Essa abordagem é amplamente utilizada em áreas como redes de computadores, análise de dados, logística, inteligência artificial, entre outras.

O presente trabalho prático foi proposto com o objetivo de consolidar os conhecimentos adquiridos ao longo da disciplina, por meio do desenvolvimento de uma ferramenta computacional que emprega grafos como estrutura central para a modelagem e o processamento de dados. A atividade envolve todas as etapas de um projeto completo: desde a definição do problema e a modelagem do grafo, até a implementação dos algoritmos e a construção de uma interface funcional para visualização e interação com os resultados.

Essa iniciativa não apenas reforça a aplicação prática dos conceitos teóricos estudados, como também estimula o desenvolvimento de competências técnicas e de engenharia de software, incluindo organização de código, documentação, uso de bibliotecas especializadas e integração entre componentes. Ao final, espera-se que a ferramenta desenvolvida seja capaz de resolver um problema real ou simulado de forma eficiente, demonstrando na prática a importância e a versatilidade dos grafos como ferramenta computacional.

1.1. Escolha do Tema

Tema 9. Gestão de Conflitos em Sistemas de Dependência: esse tema envolve detecção de ciclos, ordenação topológica e execução segura de tarefas em sistemas com dependências, como em gerenciadores de pacotes, fluxos de trabalho ou pipelines de dados.

A escolha do Tema 9 — *Gestão de Conflitos em Sistemas de Dependência* — foi motivada pelo seu alinhamento direto com os conceitos centrais estudados na disciplina de Teoria dos Grafos. A representação de tarefas como vértices e dependências como arestas dirigidas permite a modelagem precisa de problemas recorrentes em ambientes computacionais modernos, como sistemas de build, orquestração de microsserviços e gerenciadores de pacotes. Além disso, o estudo e aplicação de algoritmos clássicos como detecção de ciclos (Tarjan) e ordenação topológica (Kahn ou DFS pós-ordem) fornecem uma oportunidade prática de consolidar os conhecimentos teóricos adquiridos ao longo da disciplina.

Outro fator decisivo para a escolha do tema foi sua relevância real no contexto do desenvolvimento de software. Ciclos de dependência são um problema frequente que pode comprometer a execução correta de sistemas distribuídos ou modulares. Ao implementar uma ferramenta capaz de identificar esses conflitos e sugerir uma ordem segura de execução, o projeto propõe não apenas uma aplicação direta da teoria dos grafos, mas também uma solução com valor prático, reutilizável e extensível. Essa abordagem oferece uma ponte concreta entre teoria e prática, tornando o tema não só didático, mas também tecnicamente desafiador e relevante para o cenário atual da computação.

1.2. Problema a Ser Resolvido

Sistemas que possuem tarefas interdependentes (como instalação de pacotes, workflows de CI/CD ou módulos de software) frequentemente enfrentam conflitos de dependência ou ciclos que impedem a execução segura. Em sistemas computacionais que envolvem tarefas interdependentes — como pipelines de integração contínua, gerenciadores de pacotes e ferramentas de automação — é comum que uma tarefa dependa da execução bem-sucedida de outra. Essas relações de dependência podem ser modeladas como um grafo dirigido, no qual os vértices representam as tarefas e as arestas direcionadas indicam as dependências. O problema surge quando esse grafo contém ciclos, o que impede a definição de uma ordem válida de execução, resultando em falhas, travamentos ou comportamentos indeterminados. Sendo assim, o desafio central deste trabalho é desenvolver uma ferramenta que, a partir de um conjunto de tarefas e suas dependências, seja capaz de construir o grafo correspondente, detectar automaticamente a presença de ciclos e, se possível, realizar uma ordenação topológica que garanta a execução segura e eficiente do sistema.

Em resumo, a proposta é criar uma ferramenta que:

- Modele o sistema como um grafo dirigido
- Detecte ciclos (conflitos)
- Forneça uma execução segura (ordenada topologicamente)

2. Modelagem e Planejamento da Solução

Nesta seção, são apresentados os aspectos relacionados à modelagem do problema escolhido e ao planejamento da solução proposta. A estrutura adotada visa representar o sistema de forma clara, permitindo a aplicação eficiente dos algoritmos e técnicas necessários para o processamento e análise dos dados.

2.1. Justificativa da Modelagem

A utilização de grafos para modelar sistemas com tarefas interdependentes é justificada pela capacidade natural dessa estrutura em representar relações de dependência. Cada tarefa pode ser representada como um vértice, enquanto as dependências são expressas como arestas direcionadas, permitindo visualizar e manipular o fluxo de execução de forma clara e objetiva.

Um dos principais desafios nesses sistemas é a ocorrência de ciclos no grafo, que tornam impossível determinar uma ordem de execução válida. A detecção e resolução desses ciclos são essenciais para garantir a execução segura e previsível das tarefas. Para isso, a modelagem com grafos permite a aplicação direta de algoritmos clássicos e bem estabelecidos, como o algoritmo de Tarjan, utilizado para identificar componentes fortemente conectados (indicadores de ciclos), e o algoritmo de Kahn, empregado para gerar uma ordenação topológica eficiente.

Além disso, a implementação desse modelo é amplamente facilitada por bibliotecas especializadas para a vizualização desses grafos de forma grafica. Dessa forma, a escolha da modelagem com grafos não apenas se alinha com a teoria estudada, mas também permite o desenvolvimento prático e robusto da solução.

2.2. Diagrama de Casos de uso

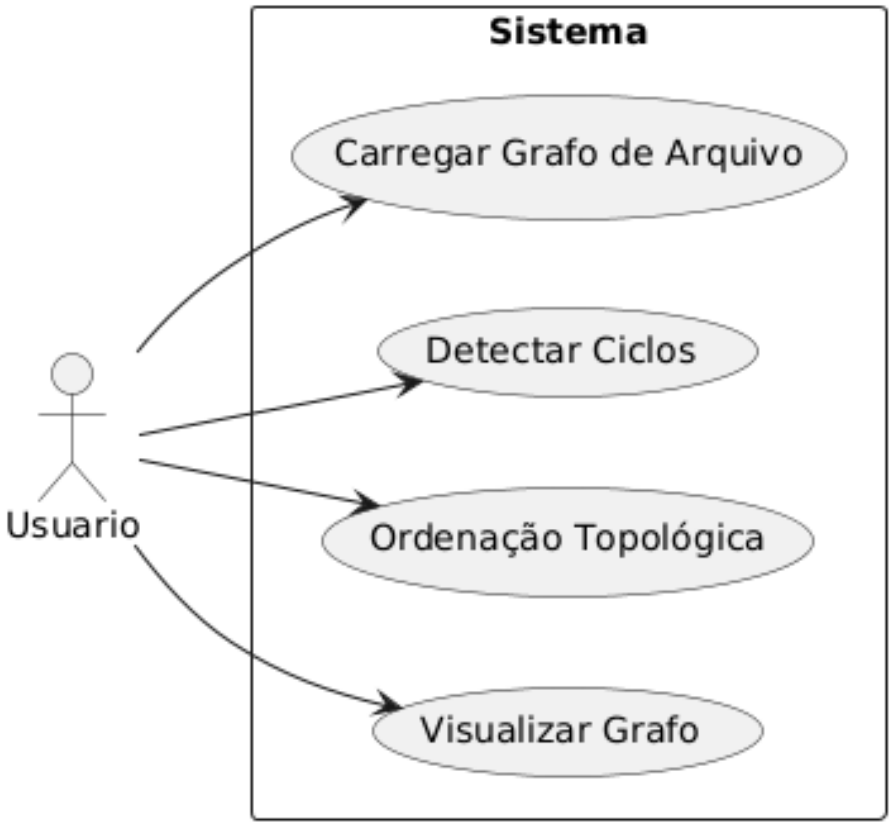

Figure 1. Diagrama de caso de uso do sistema

O diagrama de casos de uso ilustra as principais interações entre o usuário e a ferramenta de gestão de dependências. Ele apresenta as funcionalidades centrais do sistema a partir da perspectiva do usuário, que assume o papel de ator primário. Dentre os casos de uso modelados, destacam-se: o carregamento de arquivos de entrada (no formato `.xml`), a detecção de ciclos de dependência no grafo, a execução da ordenação topológica para garantir a sequência segura de execução das tarefas, e a visualização gráfica do grafo construído. Esse diagrama permite entender claramente o escopo funcional do sistema e as ações que podem ser executadas por um usuário final.

2.3. Diagrama de Classes Uml

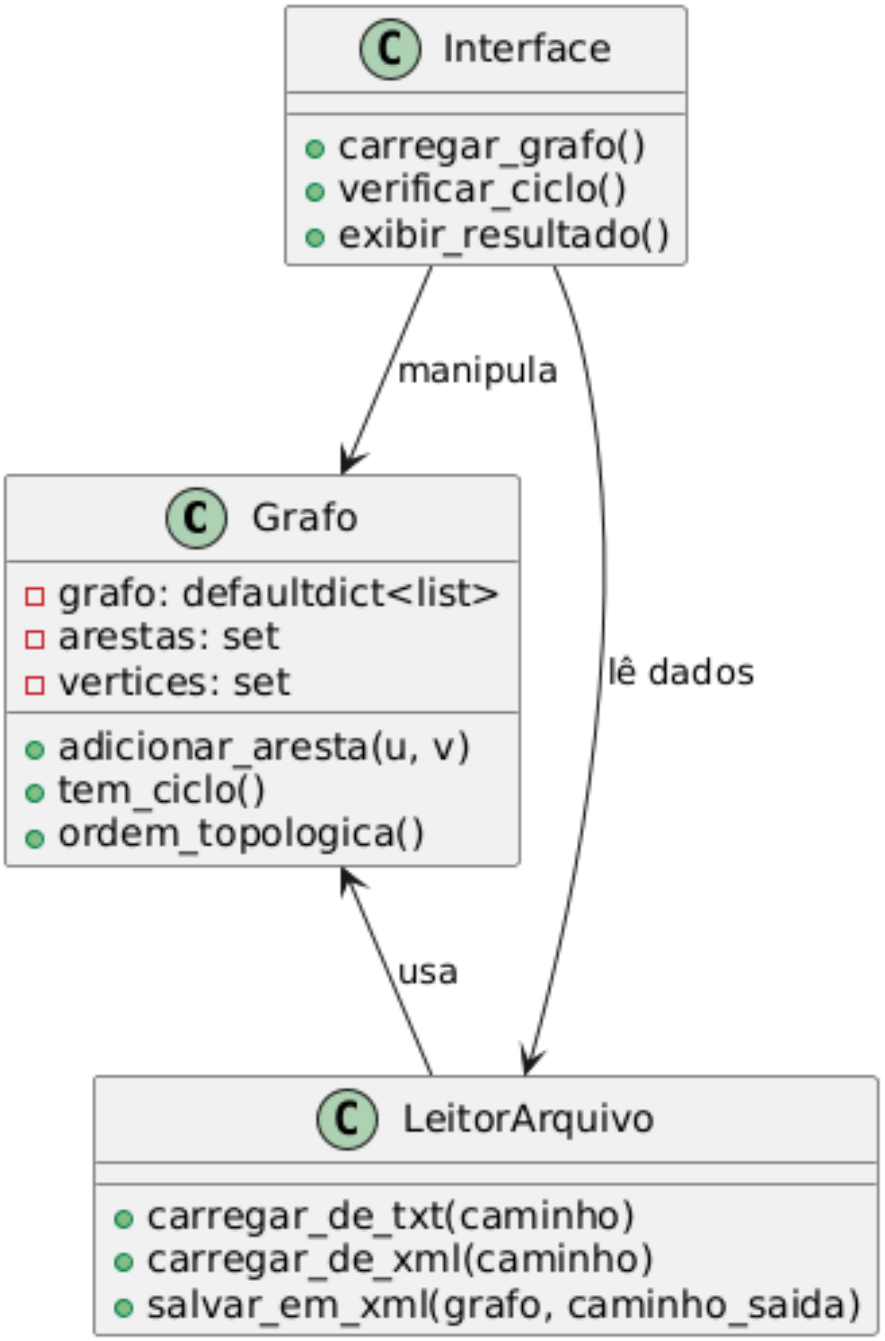

Figure 2. Diagrama de classe do sistema

O diagrama de classes ilustra a estrutura principal da aplicação, composta por três classes: Grafo, LeitorArquivo e Interface. A classe Grafo é responsável por representar a estrutura de dependências e aplicar os algoritmos de análise. A classe LeitorArquivo cuida da leitura e escrita de arquivos nos formatos suportados, como .txt e .xml. Já a classe Interface representa o ponto de interação com o usuário, coordenando a importação dos dados, verificação de ciclos e apresentação dos resultados.

2.4. Diagrama de Sequencias Uml

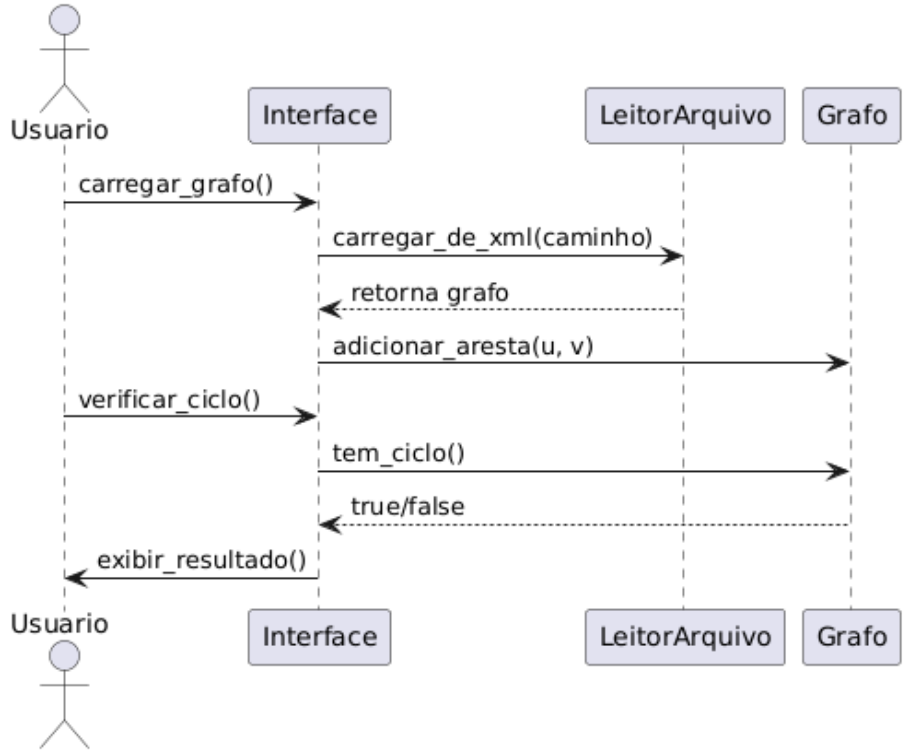

Figure 3. Diagrama de sequencia do sistema

Este diagrama representa o fluxo de execução principal do sistema: o usuário interage com a Interface, que coordena a leitura dos dados com a classe LeitorArquivo, repassa as informações para a construção do grafo via a classe Grafo, e por fim executa a verificação de ciclos. A resposta final é exibida ao usuário.

3. Tecnologias

A ferramenta foi inicialmente concebida para ser desenvolvida na linguagem Python, escolhida por sua versatilidade, legibilidade e ampla gama de bibliotecas voltadas ao processamento de dados e visualização de grafos. Dentre as bibliotecas utilizadas, destacam-se o `pandas`, responsável pelo tratamento e manipulação de dados estruturados; o textttmatplotlib.pyplot, empregado na geração de gráficos visuais para exibição da estrutura do grafo e seus elementos.

As fontes de dados utilizadas no projeto foram arquivos no formato `.xml`, provenientes de conjuntos de dados reais voltados à composição de serviços Web. Esses arquivos contêm descrições formais de tarefas e suas respectivas dependências, que foram convertidas em grafos durante o processo de ingestão de dados pela aplicação.

Além do código, o projeto foi documentado e representado visualmente por meio de diagramas UML. Foram elaborados três tipos principais de diagramas: o Diagrama de Casos de Uso, que representa as interações do usuário com o sistema; o Diagrama de Classes, que descreve a estrutura e relacionamento entre as entidades da aplicação; e o Diagrama de Sequência, que ilustra a dinâmica da execução dos processos e a comunicação entre os componentes durante a operação da ferramenta.

4. Divisão de Responsabilidades

Etapa	Tarefa e Responsáveis
1	Modelagem e definição de entrada/saída — [Victor Lucas/Davi Augusto]
2	Algoritmos de ciclo e ordenação — [Pedro Moraes/Davi Augusto]
3	Interface gráfica e visualização — [Luisa/Luiz Filipe]
4	Integração, testes e relatório final — [DaviAugusto/Luiz Filipe/Victor Lucas]

5. Referencial Teórico

Um trabalho [Haeupler et al. 2012] apresenta algoritmos otimizados para detecção de ciclos e ordenação topológica em grafos dirigidos de forma incremental, ou seja, enquanto novos nós e arestas são adicionados dinamicamente. Isso é extremamente relevante para o seu projeto, pois em sistemas com tarefas interdependentes (como pipelines de CI/CD ou instalação de pacotes), o grafo evolui continuamente e precisa ser verificado em tempo real. A aplicação desses algoritmos garante que o sistema mantenha a consistência e evite loops que comprometam a execução.

Um artigo recente [Chen et al. 2024] propõe uma abordagem quase linear para ordenação topológica em grafos com atualizações frequentes, com foco em manter a performance mesmo em grandes sistemas de dependência. Para o seu projeto, essa abordagem é útil especialmente se o sistema modelado (ex: um gerenciador de pacotes) sofrer atualizações constantes e precisar reorganizar a execução de forma eficiente, sem reprocessar o grafo completo.

O estudo de [Sigursson 2016] compara empiricamente diferentes algoritmos para ordenação topológica incremental. Sua aplicação no trabalho está na escolha e justificativa da técnica mais eficiente para detectar conflitos de dependência à medida que o sistema cresce. O artigo ajuda a embasar decisões práticas sobre qual algoritmo utilizar para balancear desempenho e precisão na resolução de conflitos.

O princípio das dependências acíclicas (ADP), descrito por Robert C. Martin [Martin 2003], estabelece que os módulos de um sistema de software devem ser organizados de forma a evitar ciclos nas suas dependências. Esse princípio fundamenta diretamente o objetivo do presente trabalho, que busca modelar sistemas interdependentes como grafos dirigidos e identificar automaticamente ciclos que poderiam violar esse princípio. A ferramenta proposta atua como um verificador automático do ADP, promovendo uma arquitetura de software mais modular, escalável e sustentável.

O livro de Cormen e colaboradores [Cormen et al. 2009] é uma das principais referências acadêmicas em algoritmos, e oferece uma abordagem formal e aprofundada sobre grafos dirigidos, ordenação topológica e detecção de ciclos. Esses conceitos formam a base teórica da implementação deste projeto, onde a ordenação topológica garante uma execução segura de tarefas interdependentes e a detecção de ciclos atua como mecanismo de prevenção de conflitos. O algoritmo de Kahn e a busca em profundidade (DFS) são discutidos como técnicas essenciais para esses propósitos.

6. Desenvolvimento

O desenvolvimento da ferramenta foi realizado utilizando a linguagem de programação Python, escolhida por sua clareza sintática, ampla base de bibliotecas e suporte consolidado ao processamento de grafos e visualização de dados.

O sistema foi projetado para receber como entrada arquivos estruturados contendo descrições de tarefas e suas dependências. Esses arquivos, no formato .xml, são processados e convertidos em um grafo dirigido, em que cada nó representa uma tarefa e cada aresta indica uma dependência direta entre duas tarefas. A seguir, o grafo é submetido à análise por meio de algoritmos de detecção de ciclos e ordenação topológica. Os resultados são apresentados ao usuário, tanto em formato textual quanto visual, com destaque para os ciclos detectados e a ordem segura de execução proposta.

7. Algoritmos

A base teórica do sistema se apoia em algoritmos clássicos da teoria dos grafos, amplamente documentados na literatura. Para a detecção de ciclos, o sistema implementa duas abordagens complementares:

A Busca em Profundidade que permite identificar ciclos à medida que vértices já visitados voltam a ser encontrados na pilha de execução.

O Algoritmo de Tarjan, mais sofisticado, que detecta componentes fortemente conectados (SCCs) e é eficiente para encontrar ciclos complexos em grandes grafos.

Para a ordenamento topológico, a ferramenta utiliza:

O Algoritmo de Kahn, baseado na remoção iterativa de nós com grau de entrada zero, garantindo uma ordenação linear válida sempre que o grafo for acíclico.

Uma versão baseada em DFS inverso (pós-ordem), que visita recursivamente os nós e empilha as tarefas conforme conclui a visita, resultando em uma ordenação segura.

Caso o grafo contenha ciclos, a ordenação topológica não é executada, e o sistema retorna ao usuário um alerta indicando os conflitos.

7.1. Interface

or meio de uma interface gráfica simples, o usuário pode carregar arquivos de entrada nos formatos .xml, .json ou .csv, executar os algoritmos de análise e visualizar os resultados de forma clara.

A interface é composta pelas seguintes principais funcionalidades: “Detectar Ciclos”, “Executar Ordenação” e “Exibir Grafo”. Ao carregar um arquivo válido, a estrutura de dependências é exibida graficamente, com destaque em vermelho para os ciclos identificados, quando existentes. Caso o grafo seja acíclico, o sistema apresenta a sequência segura de execução das tarefas. Além da visualização gráfica, as informações também são exibidas em texto, oferecendo ao usuário uma visão detalhada dos resultados.

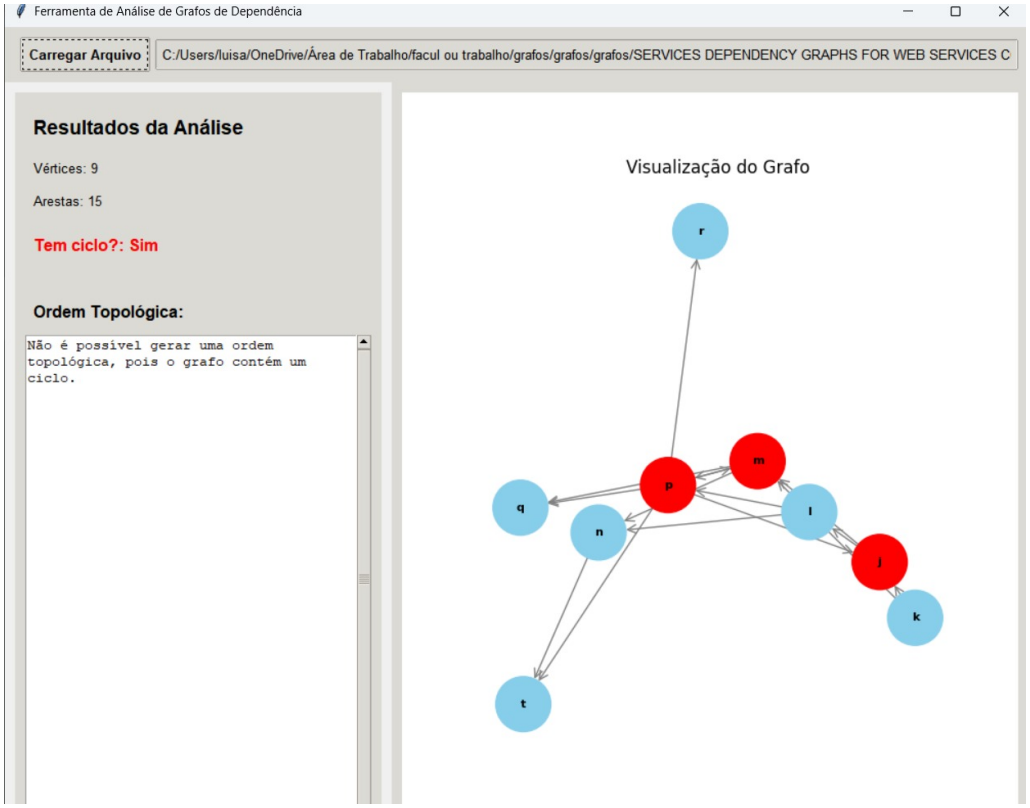

Figure 4. Exemplo de interface

8. Análise dos Resultados Obtidos

Durante os testes realizados com a ferramenta de análise de grafos de dependência, foram utilizados três arquivos de entrada com características bastante distintas, tanto em quantidade de vértices quanto em complexidade estrutural. O primeiro caso envolveu um grafo com mais de 5.900 vértices e 8.791 arestas;

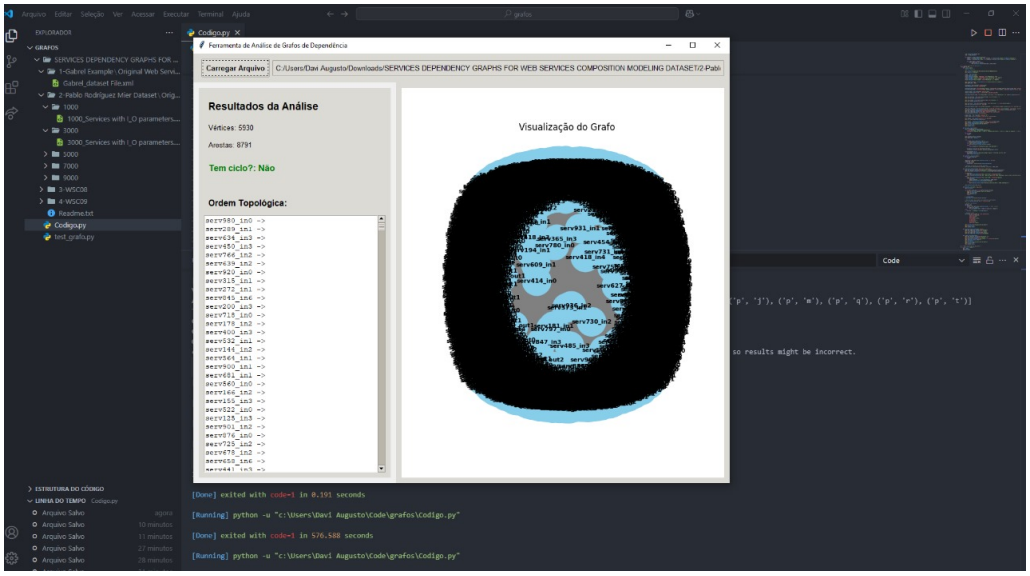

Figure 5. Resultado da analise sem ciclo grande

o segundo, um grafo significativamente menor, com apenas 22 vértices e 32 arestas.

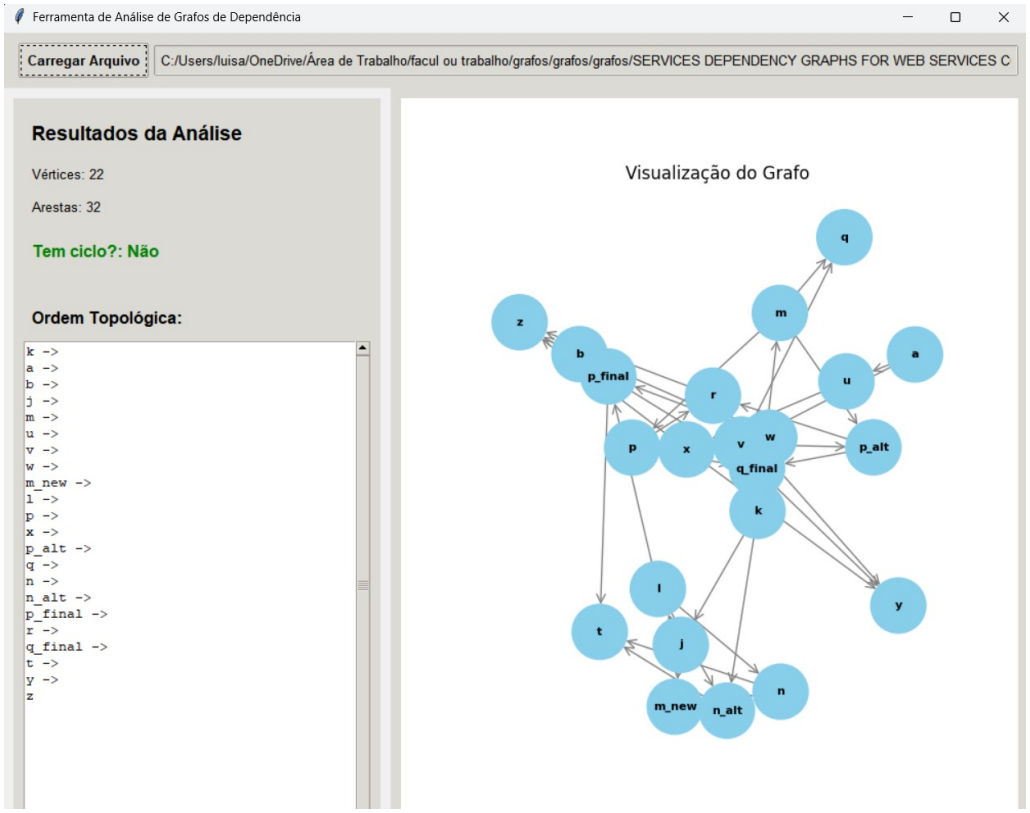

Figure 6. Resultado da analise sem ciclo menor

Em todos os cenários testados, a aplicação foi capaz de identificar corretamente a presença (ou ausência) de ciclos e produzir uma ordenação topológica válida, quando possível, mas mesmo quando este não é o caso a duração do algoritmo não sofre muita alternancia.

Os resultados foram apresentados ao usuário tanto em forma textual (ordem topológica e estatísticas do grafo) quanto visual, com a geração gráfica do grafo em tempo real. A ferramenta também destacou corretamente a quantidade de vértices e arestas e, no caso de grafos acíclicos, sinalizou que a execução era segura.

Visualmente, observou-se que em grafos pequenos (como o de 22 vértices) a visualização do grafo foi clara e informativa, permitindo identificar rapidamente os fluxos de dependência entre os nós. Já nos casos maiores, como o grafo com quase 9.000 arestas, a visualização manteve a estrutura global do grafo, mas tornou-se densa e de difícil leitura, evidenciando a necessidade de estratégias de agrupamento, zoom ou filtragem para contextos mais complexos.

9. Avaliação do Desempenho

Os testes também revelaram diferenças marcantes entre o desempenho do backend (processamento computacional) e o frontend (interface gráfica) da ferramenta. O tempo de execução da análise de grafos no backend foi altamente eficiente, mesmo para grandes volumes de dados. Para o maior grafo testado (com 8.791 arestas), o tempo de análise computacional foi de apenas 0.8068 segundos. Já para um grafo com 2.000 arestas, o tempo registrado foi ainda menor: 0.0077 segundos. Para o grafo simples com 22 vértices, o tempo de execução foi praticamente instantâneo: 0.00001 segundos.

Por outro lado, a renderização gráfica no frontend mostrou-se um gargalo de desempenho. O mesmo grafo de 8.791 arestas que foi processado em menos de 1 segundo pelo backend, levou aproximadamente 576 segundos (cerca de 10 minutos) para ser exibido na interface gráfica. Mesmo o grafo pequeno, de apenas 22 vértices, exigiu cerca de 8 segundos para renderizar, o que é considerável frente ao tempo de execução da análise.

Esses resultados indicam que a geração da visualização do grafo é o ponto mais custoso do sistema, especialmente em casos grandes. Isso pode ser explicado pela densidade das conexões, pelo uso de algoritmos de layout mais pesados e pela limitação da renderização dinâmica em interfaces gráficas baseadas em tkinter ou bibliotecas similares.

10. Conclusões

A ferramenta desenvolvida atingiu com sucesso os objetivos propostos: modelar sistemas de dependência como grafos dirigidos, detectar automaticamente a presença de ciclos e, quando possível, realizar a ordenação topológica para garantir uma execução segura. O backend demonstrou ser altamente eficiente, conseguindo analisar grafos com milhares de arestas em menos de um segundo.

Entretanto, a avaliação de desempenho revelou que a interface gráfica representa um gargalo significativo, especialmente na renderização de grafos grandes. A diferença entre o tempo de processamento e o tempo de exibição gráfica é expressiva e merece atenção para futuras melhorias.

Entre as possíveis soluções para esse problema estão: a substituição da biblioteca gráfica atual por soluções mais performáticas (como PyQtGraph ou Plotly), a implementação de filtros de visualização por grupos de nós, e a utilização de layouts hierárquicos ou simplificados para evitar sobreposição excessiva de elementos.

Em síntese, a ferramenta provou ser funcional e eficaz para análise de dependências, apresentando grande potencial de aplicação tanto em contextos educacionais quanto em sistemas de software reais. Com ajustes na camada visual, ela poderá se tornar ainda mais escalável e responsiva.

References

Chen, L., Diks, K., Łacki, J., Sankowski, P., and Wulff-Nilsen, C. (2024). Incremental topological sort and cycle detection in expected total time. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition. Seções sobre grafos dirigidos, ordenação topológica e detecção de ciclos.

Haeupler, B., Sen, S., and Tarjan, R. E. (2012). Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)*, 8(1):1–33.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*, chapter The Acyclic Dependencies Principle, pages 183–192. Prentice Hall.

Sigursson, R. L. (2016). Practical performance of incremental topological sorting and cycle detection algorithms.