UNIVERSIDA FEDERAL DOS VALES JEQUITINHONHA E MUCURI – UFVJM

SISTEMAS DE INFORMAÇÃO

LUANN MOREIRA FERNANDES DE OLIVEIRA

TRABALHO PRATICO:

Análise de Dependências com Grafos

DIAMANTINA

2025

1 INTRODUÇÃO

Sistemas computacionais frequentemente são compostos por múltiplos pacotes de software que possuem dependências entre si. A correta análise dessas dependências é essencial para garantir a integridade da instalação e funcionamento do sistema. Este trabalho propõe a modelagem dessas relações por meio de grafos direcionados, utilizando algoritmos clássicos como ordenação topológica, detecção de ciclos e busca de componentes fortemente conectados.

2 DESENVOLVIMENTO

2.1 Estrutura Geral do Código

O código foi desenvolvido em Python, utilizando a biblioteca `networkx` para manipulação de grafos e `matplotlib` para visualização opcional. A estrutura do programa é modular e baseada em funções específicas para cada tarefa exigida pelo enunciado. Para ser possível a utilização no Google Colab é necessário fazer upload do arquivo com entrada.txt na plataforma

```
# IMPORTAÇÕES PARA MANIPULÇÃO DE GRAFOS
import networkx as nx
import matplotlib.pyplot as plt

# CARREGAR ARQUIVO E CONSTRUIR O GRAFO
def carregar_dependencias(arquivo):
    grafo = nx.DiGraph()
    with open(arquivo, 'r') as f:
        for linha in f:
            partes = linha.strip().split()
            if len(partes) == 1:
                grafo.add_node(partes[0])
            elif len(partes) == 2:
                grafo.add_edge(partes[0], partes[1])
    return grafo

# VERIFICAR SE É POSSIVEL INSTALAR OS PACOTES (SE O GRAFO É ACICLICO)
def eh_instalavel(grafo):
    return nx.is_directed_acyclic_graph(grafo)

# LISTAR ORDEM DE ORDENAÇÃO (ORDENAÇÃO TOPOLOGICA)
```

```
def ordem_instalacao(grafo):
    if eh_instalavel(grafo):
        return list(nx.topological_sort(grafo))
    else:
        return None

# ENCONTRAR CICLOS (COMPONENTE CONECTADOS A MAIS DE UM PACOTE)
def encontrar_ciclos(grafo):
    scc = list(nx.strongly_connected_components(grafo))
    ciclos = [c for c in scc if len(c) > 1]
    return ciclos

# DEPENDENCIAS DIRETAS E INDIRETAS DE UM PACOTE
def dependencias_de(pacote, grafo):
    if pacote not in grafo:
        return []
    return list(nx.descendants(grafo, pacote))

# SIMULAR REMOÇÃO DE UM PACOTE E VER PACOTES AFETADOS POR ESSA REMOÇÃO
def pacotes_afetados(pacote, grafo):
    afetados = []
    for no in grafo.nodes:
        if pacote in dependencias_de(no, grafo):
            afetados.append(no)
    return afetados

# INDENTIFICAR PACOTES CRITICOS
def pacotes_criticos(grafo, top_n=5):
    dependencias = {no: len(list(nx.ancestors(grafo, no))) for no in
grafo.nodes}
    return sorted(dependencias.items(), key=lambda x: x[1],
reverse=True)[:top_n]
# INTERFACE DE MENU E EXECUÇÃO DAS OPÇÕES
def menu():
    grafo = carregar_dependencias('entrada.txt')
    while True:
        print("\n--- Menu ---")
        print("1. Verificar se é possível instalar todos os pacotes")
        print("2. Mostrar ordem de instalação (ordem topológica)")
        print("3. Identificar ciclos de dependência")
        print("4. Consultar dependências de um pacote")
        print("5. Simular remoção de um pacote")
        print("6. Mostrar pacotes críticos")
        print("7. Sair")
        opcao = input("Escolha uma opção: ")
```

```
        if opcao == '1':
            if eh_instalavel(grafo):
                print("É possível instalar todos os pacotes (grafo acíclico).")
            else:
                print("Não é possível instalar todos os pacotes (ciclos
encontrados).")
        elif opcao == '2':
            ordem = ordem_instalacao(grafo)
            if ordem:
                print("Ordem de instalação:", ordem)
            else:
                print("Não é possível ordenar. Existem ciclos.")
        elif opcao == '3':
            ciclos = encontrar_ciclos(grafo)
            if ciclos:
                print("Ciclos encontrados:")
                for c in ciclos:
                    print(" - ", list(c))
            else:
                print("Nenhum ciclo encontrado.")
        elif opcao == '4':
            pacote = input("Digite o nome do pacote: ")
            deps = dependencias_de(pacote, grafo)
            print(f"Dependências de {pacote}:", deps)
        elif opcao == '5':
            pacote = input("Digite o pacote a remover: ")
            afetados = pacotes_afetados(pacote, grafo)
            print(f"Pacotes afetados pela remoção de {pacote}:", afetados)
        elif opcao == '6':
            criticos = pacotes_criticos(grafo)
            print("Pacotes mais críticos (com mais dependentes):")
            for pacote, qtd in criticos:
                print(f"{pacote}: {qtd} dependentes")
        elif opcao == '7':
            break
        else:
            print("Opção inválida.")
# EXECUTAR O MENU
menu()
```

2.2 Carregamento das Dependências

A função `carregar_dependencias()` lê o arquivo `entrada.txt`, identificando os pacotes e suas dependências. Para cada linha com dois termos, é criada uma aresta direcionada no grafo,

representando a dependência de um pacote em relação ao outro. Linhas com apenas um pacote representam pacotes independentes, que são adicionados como nós isolados.

2.3 Verificação de Ciclos

A função `eh_instalavel()` verifica se o grafo é acíclico utilizando o método `is_directed_acyclic_graph()` da biblioteca `networkx`. A presença de ciclos impede a instalação correta dos pacotes, sendo, portanto, uma verificação fundamental.

2.4 Ordenação Topológica

A função `ordem_instalacao()` retorna uma lista com a ordem válida de instalação dos pacotes, desde que o grafo seja acíclico. Essa ordenação é obtida por meio do algoritmo de ordenação topológica (`topological_sort()`).

2.5 Detecção de Ciclos

Com a função `encontrar_ciclos()`, são identificadas as componentes fortemente conectadas do grafo com mais de um nó. Essas componentes representam ciclos de dependência, e sua detecção é essencial para correção da estrutura do sistema.

2.6 Consulta de Dependências

A função `dependencias_de()` permite consultar todas as dependências diretas e indiretas de um pacote específico. Essa análise é feita utilizando o método `descendants()` do `networkx`, que percorre todos os caminhos descendentes a partir do nó informado.

2.7 Simulação de Remoção de Pacotes

A função `pacotes_afetados()` identifica quais pacotes deixariam de funcionar caso um determinado pacote fosse removido. Para isso, ela verifica todos os pacotes que possuem dependência direta ou indireta do pacote alvo.

2.8 Identificação de Pacotes Críticos

A função `pacotes_criticos()` analisa quais pacotes são mais críticos, ou seja, aqueles com maior número de dependentes no sistema. A contagem é feita utilizando `ancestors()`, retornando os `n` pacotes mais influentes na estrutura do grafo.

2.9 Interface com o Usuário

O programa apresenta um menu interativo por terminal, com sete opções principais. Essa interface permite a execução das funcionalidades descritas de forma intuitiva, facilitando a análise de diferentes arquivos de entrada.

3 CONSIDERAÇÕES FINAIS

O projeto atendeu a todos os requisitos propostos, aplicando corretamente os conceitos de grafos estudados em sala de aula. A implementação é flexível, aceita diferentes formatos e tamanhos de entrada, e fornece uma análise completa sobre as relações de dependência entre pacotes.

Por meio das funções desenvolvidas, é possível detectar problemas, encontrar soluções viáveis de instalação, e identificar pacotes críticos ao funcionamento do sistema.