AEDSIII - Caminhos Críticos

Caio D.Alves[1,2]

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
CEP 35931-008 – João Monlevade – MG – Brasil

[2]Departamento de Computação e Sistemas (DECSI)
Universidade Federal de Ouro Preto (UFOP) – João Monlevade, MG – Brasil

{caio}@caio.damasceno@aluno.ufop.edu.br

Resumo. *Este trabalho prático tem como objetivo aplicar algoritmos de caminho crítico na grade curricular de um curso de graduação. O método permite identificar a sequência mais crítica de disciplinas e suas dependências.*
O programa desenvolvido lê dados de arquivos .csv, ajusta algoritmos de caminho mínimo para resolver o problema do caminho máximo e apresenta ao usuário o caminho crítico e o tempo mínimo de conclusão da graduação.

1. Introdução

O presente trabalho tem como objetivo desenvolver um algoritmo para encontrar o caminho crítico em cursos, aplicando conceitos de grafos. O caminho crítico é a sequência de etapas que, caso haja um atraso em qualquer uma delas, compromete o prazo mínimo de conclusão da graduação. Através de técnicas de grafos e algoritmos de caminho máximo, podemos automatizar a identificação do caminho crítico, otimizando a gestão do curso.

Neste trabalho pratico, o algoritmo utilizado para resolver o problema é uma variação do algoritmo de Bellman-Ford, modificado para encontrar o caminho mais longo em um grafo, uma vez que o caminho crítico é, na verdade, o caminho de maior duração entre o início e o fim do curso.

2. Repositório do Código

O código implementado pode ser encontrado no seguinte repositório GitHub (O codigo esta comentado linha a linha): `https://github.com/CaioDamascenoAlves/AEDSIII/tree/main/TRABALHO%20PRATICO%2002`

3. Implementação

A implementação do programa foi feita em Python, utilizando a biblioteca `csv` para leitura de arquivos e `defaultdict` da coleção `collections` para criar a estrutura de dados do grafo. A seguir, detalhamos as principais estruturas e funções implementadas.

4. Estruturas de Dados

O grafo, que representa as disciplinas do curso, é implementado como um dicionário, onde as chaves correspondem aos códigos das disciplinas e os valores são listas das disciplinas dependentes. Essa estrutura permite uma representação eficiente das dependências entre as disciplinas, facilitando a navegação e a manipulação dos dados.

Adicionalmente, um dicionário separado é utilizado para armazenar informações detalhadas sobre cada disciplina. Esse dicionário inclui dados como nome, duração e período, o que possibilita um gerenciamento mais organizado e claro das disciplinas ao longo do processo.

4.1. Exemplo da Estrutura do Grafo

A estrutura do grafo pode ser exemplificada da seguinte forma:

```
{
    'D1': ['D2', 'D3'],
    'D2': ['D4'],
    'D3': ['D4']
}
```

Neste exemplo, a disciplina `D1` possui dependências em relação às disciplinas `D2` e `D3`, ambas por sua vez, dependem da disciplina `D4`. Essa representação ilustra claramente a relação de precedência entre as disciplinas, essencial para a análise do caminho crítico.

4.2. Estruturas Utilizadas

Para garantir a eficiência na análise das disciplinas, duas estruturas de dados principais foram utilizadas:

- **Grafo:** Representado por um dicionário, onde cada chave é uma disciplina e os valores são listas que representam as disciplinas que dependem da disciplina correspondente. Essa configuração permite a construção de um grafo direcionado que reflete as relações de precedência.
- **Informações das Disciplinas:** Cada disciplina é armazenada em um dicionário separado, onde são incluídas as seguintes informações:
  - **Nome:** O nome descritivo da disciplina.
  - **Duração:** O tempo necessário para completar a disciplina.
  - **Período:** O período em que a disciplina deve ser realizada.

5. Funções Principais

5.1. ler_csv()

Esta função lê o arquivo CSV fornecido pelo usuário, carrega as tarefas e suas dependências no grafo, além de registrar suas durações. O arquivo CSV deve seguir o seguinte formato:

```
Código,Nome,Período,Duração,Dependências
T1,Lógica de Programação,1,5,
T2,Engenharia de Software,2,3,T1
T3,Sistemas Web e Mobile,3,2,T1
```

5.2. bellman_ford_caminho_mais_longo()

Esta função é uma adaptação do algoritmo de Bellman-Ford para encontrar o caminho mais longo em vez do caminho mais curto. Para isso, invertemos a lógica de relaxamento das arestas. Caso algum ciclo positivo seja detectado, o programa lança um erro. Conforme descrito por SESvTutorial [SESvTutorial 2024], o Bellman-Ford tradicionalmente resolve o problema de caminho mais curto. No entanto, aqui a mudança crucial foi invertemos a lógica de relaxamento das arestas:

Relaxation longest path:

```
if (d[u] + c(u,v) > d[v]) then d[v] = d[u] + c(u,v)
```

5.3. obter_caminho_critico()

Após a execução do algoritmo de Bellman-Ford, esta função percorre o grafo de trás para frente, a partir do nó de fim, e reconstrói o caminho crítico.

5.4. analisar()

Esta é a função principal, que chama as funções anteriores para ler o arquivo CSV, processar o grafo e, por fim, exibir o caminho crítico e o tempo mínimo de conclusão.

5.5. Listagem do Código

Aqui está a listagem do código utilizado na implementação:

```
# Importa  o das bibliotecas necess rias
import csv
from collections import defaultdict

class AnalisadorCaminhoCritico:
    def __init__(self, nome_arquivo):
        self.nome_arquivo = nome_arquivo
        self.grafo = defaultdict(list)
        self.tarefas = {}

    def ler_csv(self):
        with open(self.nome_arquivo, 'r', encoding='utf-8') as arquivo:
            leitor_csv = csv.DictReader(arquivo)
            for linha in leitor_csv:
                codigo = linha['C digo']
                self.tarefas[codigo] = {
                    'nome': linha['Nome'],
                    'periodo': int(linha['Per odo']),
                    'duracao': int(linha['Dura  o'])
                }
                dependencias = linha['Depend ncias'].split(';') if linha['Depend ncias
                    '] else []
                for dep in dependencias:
                    if dep:
                        self.grafo[dep].append(codigo)

    def bellman_ford_caminho_mais_longo(self):
        distancia = {tarefa: float('-inf') for tarefa in self.tarefas}
        distancia['INICIO'] = 0
        distancia['FIM'] = float('-inf')
        predecessor = {tarefa: None for tarefa in self.tarefas}
        predecessor['INICIO'] = None
        predecessor['FIM'] = None

        for tarefa in self.tarefas:
            if not any(tarefa in deps for deps in self.grafo.values()):
                self.grafo['INICIO'].append(tarefa)
```

```
            if tarefa not in self.grafo:
                self.grafo[tarefa].append('FIM')

        for _ in range(len(self.tarefas) + 2):
            for u in self.grafo:
                for v in self.grafo[u]:
                    peso = self.tarefas[v]['duracao'] if v != 'FIM' else 0
                    if distancia[u] + peso > distancia[v]:
                        distancia[v] = distancia[u] + peso
                        predecessor[v] = u

        for u in self.grafo:
            for v in self.grafo[u]:
                peso = self.tarefas[v]['duracao'] if v != 'FIM' else 0
                if distancia[u] + peso > distancia[v]:
                    raise ValueError("O grafo cont m um ciclo de peso positivo")

        return distancia, predecessor

    def obter_caminho_critico(self, predecessor, fim):
        caminho = []
        atual = fim
        while atual != 'INICIO':
            caminho.append(atual)
            atual = predecessor[atual]
        caminho.reverse()
        return caminho[:-1]

    def analisar(self):
        self.ler_csv()
        try:
            distancia, predecessor = self.bellman_ford_caminho_mais_longo()
            caminho_critico = self.obter_caminho_critico(predecessor, 'FIM')
            print("Caminho cr tico:")
            for tarefa in caminho_critico:
                if tarefa in self.tarefas:
                    print(f"{tarefa}: {self.tarefas[tarefa]['nome']}")
            print(f"\nTempo m nimo de conclus o: {distancia['FIM']} per odos")
        except ValueError as e:
            print(f"Erro: {e}")
```

Listing 1. Implementação do Algoritmo Bellman-Ford CPM

6. Testes Executados

Foram realizados testes utilizando diferentes arquivos CSV que representam disciplinas de cursos e suas dependências. Os arquivos fornecidos pelo professor, como o TOY, foram utilizados como base, e arquivos adicionais foram criados representando cursos reais da Ufop.

6.1. Exemplo de Teste

Para rodar o projeto, basta executar o arquivo `main.py` usando o comando:

```
python main.py
```

O programa solicitará ao usuário o caminho do arquivo CSV contendo as tarefas e dependências. Como exemplo, o arquivo `SJM.csv` foi processado da seguinte maneira:

Ao rodar o programa e fornecer o arquivo `SJM.csv`, a saída foi:

```
Caminho crítico:
CSI101: PROGRAMAÇÃO DE COMPUTADORES I
CSI103: ALGORITMOS E ESTRUTURA DE DADOS I
```

```
CSI105: ALGORITMOS E ESTRUTURA DE DADOS III
CSI106: FUNDAMENTOS TEÓRICOS DA COMPUTAÇÃO
CSI107: LINGUAGENS DE PROGRAMAÇÃO

Tempo mínimo de conclusão: 5 períodos
```

O programa então permite que o usuário insira novos arquivos CSV ou digite '0' para sair. Ao digitar '0', o programa é encerrado com a mensagem:

```
Encerrando o programa.
```

Esse exemplo mostra como o algoritmo foi capaz de identificar o caminho crítico de dependências no curso e calcular o tempo mínimo necessário para a conclusão das disciplinas.

7. Conclusão

Este trabalho permitiu aplicar conhecimentos de grafos para resolver um problema real de gestão de disciplinas em um curso de graduação. A principal dificuldade foi ajustar o algoritmo de Bellman-Ford para o problema do caminho máximo. O programa desenvolvido cumpre os requisitos propostos e pode ser facilmente adaptado para diferentes contextos de planejamento de projetos.

Referências

[SESvTutorial 2024] SESvTutorial (2024). Bellman-ford algorithm tutorial. Accessed: 2024-09-24.