

Teoria de Grafos e Implementação de Algoritmos Fundamentais

29 de outubro de 2025

Enzo Oliveira D'Onofrio	- RA: 23.01561-6
Leonardo Souza Olivieri	- RA: 23.01512-8
Arthur Gama Ruiz	- RA: 23.01445-8
João Vitor Morimoto Sesma	- RA: 23.01516-0
Pedro Wilian Palumbo Bevilacqua	- RA: 23.01307-9
Felipe Fazio da Costa	- RA: 23.00055-4

Resumo

Este artigo científico explora a importância da **Teoria dos Grafos** na Ciência da Computação, detalhando conceitos fundamentais e a implementação de algoritmos cruciais. Abordamos a **planaridade de grafos** e apresentamos as implementações em pseudocódigo dos principais algoritmos de busca e otimização: **Busca em Profundidade (DFS)**, **Algoritmo de Prim**, **Algoritmo de Kruskal** e **Algoritmo de Dijkstra**. O objetivo é fornecer uma visão concisa e técnica sobre como essas estruturas e métodos são aplicados na resolução de problemas computacionais complexos, incluindo exemplos práticos e análise de complexidade.

Palavras-chave: Teoria dos Grafos; Algoritmos; Busca em Profundidade; Algoritmo de Prim; Algoritmo de Kruskal; Algoritmo de Dijkstra; Planaridade; Complexidade.

1 Introdução

A Teoria dos Grafos é um ramo da matemática discreta que estuda as relações entre objetos. Em Ciência da Computação, os grafos são modelos essenciais para representar redes, fluxos, dependências e conexões, sendo a base para a resolução de inúmeros problemas, desde a otimização de rotas até a análise de redes sociais. A eficiência na manipulação dessas estruturas depende diretamente da escolha e implementação correta de algoritmos. Este trabalho se propõe a revisar a planaridade de grafos e detalhar a implementação de cinco algoritmos fundamentais: Busca em Profundidade (DFS), Prim, Kruskal e Dijkstra.

2 Desenvolvimento do Tema

2.1 Planaridade de Grafos

Um **grafo planar** é um grafo que pode ser desenhado em um plano de forma que nenhuma de suas arestas se cruze, exceto em seus vértices. A planaridade é uma propriedade importante em diversas áreas, como no projeto de circuitos integrados, onde a minimização de cruzamentos é crucial para evitar curtos-circuitos e simplificar a fabricação.

2.1.1 Critério de Planaridade

O **Teorema de Kuratowski** fornece uma caracterização fundamental para a planaridade: um grafo é planar se, e somente se, ele não contém um subgrafo que seja uma subdivisão de K_5 (grafo completo com 5 vértices) ou $K_{3,3}$ (grafo bipartido completo com 3 vértices em cada partição) [1].

Outra propriedade importante é a **Fórmula de Euler** para grafos planares conexos, que relaciona o número de vértices (v), arestas (e) e faces (f):

$$v - e + f = 2$$

Essa fórmula é frequentemente utilizada para provar que um grafo não é planar, pois se a relação não for satisfeita, o grafo não pode ser planar.

2.1.2 Aplicação Prática

A aplicação mais notável da planaridade está no **design de placas de circuito impresso (PCBs)**. Ao modelar as conexões de um circuito como um grafo, garantir a planaridade (ou minimizar os cruzamentos) é essencial para criar uma placa de camada única, reduzindo custos e complexidade de fabricação.

2.2 Implementação de Busca em Profundidade (DFS)

A **Busca em Profundidade (Depth-First Search - DFS)** é um algoritmo para percorrer ou buscar em uma estrutura de dados de árvore ou grafo. O algoritmo explora o mais longe possível ao longo de cada ramo antes de retroceder (*backtracking*). É tipicamente implementado de forma recursiva ou utilizando uma pilha (stack).

2.2.1 Pseudocódigo e Complexidade

O pseudocódigo abaixo ilustra a abordagem recursiva:

```
1 DFS(G, v):
2     Marcar v como visitado
3     Para cada vizinho w de v:
4         Se w não foi visitado:
5             DFS(G, w)
```

Listing 1: Pseudocódigo para Busca em Profundidade (DFS)

A complexidade de tempo do DFS é de $O(V + E)$, onde V é o número de vértices e E é o número de arestas, pois cada vértice e cada aresta são examinados no máximo uma vez.

2.2.2 Aplicações Práticas

O DFS é fundamental para:

- **Ordenação Topológica:** Determinar uma ordem linear de vértices em um grafo acíclico dirigido (DAG), essencial para agendamento de tarefas com dependências.
- **Identificação de Componentes Conexos:** Encontrar todos os vértices que podem ser alcançados a partir de um determinado vértice.
- **Detecção de Ciclos:** Identificar se um grafo contém ciclos.

2.3 Implementação do Algoritmo de Prim

O **Algoritmo de Prim** é um algoritmo guloso que encontra uma **Árvore Geradora Mínima (Minimum Spanning Tree - MST)** para um grafo conexo e ponderado. A MST é um subconjunto das arestas que conecta todos os vértices sem ciclos e com o menor peso total possível. Prim constrói a MST incrementalmente, adicionando a aresta de menor peso que conecta a árvore parcial a um vértice fora dela.

2.3.1 Pseudocódigo e Complexidade

A implementação eficiente do Algoritmo de Prim utiliza uma **fila de prioridade** para selecionar rapidamente o próximo vértice a ser adicionado.

```
1 Prim(G):
2     Inicializar MST = {}
3     Escolher um vértice inicial r
4     Para cada vértice v:
5         chave[v] = infinito // Menor peso de aresta conectando v à MST
6         pai[v] = NULO          // Vértice pai de v na MST
7         chave[r] = 0
8         Q = todos os vértices em G (fila de prioridade, usando chave[v] como prioridade)
9
10    Enquanto Q não está vazio:
11        u = extraír o vértice com menor chave de Q
```

```

12     Para cada vizinho v de u:
13         Se v está em Q e peso(u, v) < chave[v]:
14             pai[v] = u
15             chave[v] = peso(u, v)
16             Atualizar v em Q
17     Retornar MST

```

Listing 2: Pseudocódigo para o Algoritmo de Prim

Com o uso de um *heap* binário (fila de prioridade), a complexidade de tempo é $O(E \log V)$ ou $O(E + V \log V)$, dependendo da estrutura de dados exata utilizada.

2.3.2 Aplicações Práticas

O Algoritmo de Prim é ideal para problemas de otimização de infraestrutura, como:

- **Redes de Telecomunicações:** Projetar a rede de cabos mais barata para conectar um conjunto de cidades.
- **Redes de Água/Electricidade:** Planejar o layout de tubulações ou linhas de transmissão para minimizar o custo total de material.

2.4 Implementação do Algoritmo de Kruskal

O **Algoritmo de Kruskal** é, assim como Prim, um algoritmo guloso que encontra a MST. A diferença é que Kruskal opera diretamente nas arestas do grafo, examinando-as em ordem crescente de peso. Uma aresta é adicionada à MST se e somente se ela não formar um ciclo com as arestas já selecionadas.

2.4.1 Pseudocódigo e Complexidade

O Kruskal depende da estrutura de dados **Union-Find (Conjuntos Disjuntos)** para gerenciar os componentes conexos e detectar ciclos de forma eficiente.

```

1 Kruskal(G):
2     MST = {}
3     Para cada vértice v em G:
4         CriarConjunto(v) // Cada vértice é seu próprio conjunto
5         Ordenar todas as arestas de G por peso crescente
6
7     Para cada aresta (u, v) na ordem crescente:
8         Se EncontrarConjunto(u) != EncontrarConjunto(v): // Não
9             forma ciclo
10            Adicionar (u, v) à MST
11            UnirConjuntos(u, v) // Une os componentes de u e v
12     Retornar MST

```

Listing 3: Pseudocódigo para o Algoritmo de Kruskal

A complexidade de tempo é dominada pela ordenação das arestas, resultando em $O(E \log E)$. Como $\log E$ é, no pior caso, $O(\log V^2)$, a complexidade é frequentemente simplificada para $O(E \log V)$.

2.4.2 Exemplo de Uso

Enquanto Prim é mais eficiente em grafos densos, Kruskal se destaca em **grafos esparsos** (com poucas arestas). Uma aplicação prática é na **criação de clusters** em análise de dados, onde as arestas representam a distância entre pontos de dados, e a MST ajuda a identificar a estrutura subjacente com o mínimo de conexões.

2.5 Implementação do Algoritmo de Dijkstra

O **Algoritmo de Dijkstra** resolve o problema do **caminho mais curto de uma única origem** em um grafo com pesos de arestas **não negativos**. Ele calcula o caminho de custo mínimo de um vértice inicial (origem) para todos os outros vértices do grafo.

2.5.1 Pseudocódigo e Complexidade

O algoritmo mantém uma estimativa da distância mais curta da origem para cada vértice e aprimora essa estimativa (relaxamento) até que o caminho mais curto seja encontrado.

```
1 Dijkstra(G, s):
2     Para cada vértice v em G:
3         distancia[v] = infinito
4         anterior[v] = NULO
5     distancia[s] = 0
6     Q = todos os vértices em G (fila de prioridade, usando
        distancia[v] como prioridade)
7
8     Enquanto Q não está vazio:
9         u = extrair o vértice com menor distancia de Q
10        Para cada vizinho v de u:
11            alt = distancia[u] + peso(u, v)
12            Se alt < distancia[v]:
13                distancia[v] = alt
14                anterior[v] = u
15                Atualizar v em Q
16    Retornar distancia, anterior
```

Listing 4: Pseudocódigo para o Algoritmo de Dijkstra

Assim como Prim, a complexidade de Dijkstra com uma fila de prioridade é $O(E \log V)$ ou $O(E + V \log V)$, sendo altamente eficiente.

2.5.2 Exemplo de Aplicação

O Dijkstra é o coração de muitos sistemas de navegação e roteamento.

- **GPS e Mapas:** Encontrar o caminho mais rápido ou mais curto entre dois pontos em uma rede rodoviária (onde os pesos das arestas são tempo ou distância).
- **Roteamento de Redes:** Protocolos de roteamento de redes de computadores utilizam variações de Dijkstra para encontrar a rota de menor custo para pacotes de dados.

3 Conclusões

A Teoria dos Grafos, com seus conceitos de planaridade e seus algoritmos fundamentais (DFS, Prim, Kruskal e Dijkstra), permanece como um pilar da Ciência da Computação. A **Busca em Profundidade** é essencial para a exploração e análise estrutural de grafos, enquanto os algoritmos de **Prim** e **Kruskal** resolvem eficientemente o problema da **Árvore Geradora Mínima**, otimizando custos de conexão. Por fim, o algoritmo de **Dijkstra** é a base para a otimização de caminhos e roteamento. A compreensão de suas implementações, complexidade e aplicações práticas é crucial para a modelagem e solução de problemas computacionais complexos no mundo real.

4 Referências Bibliográficas

- [1] Teorema de Kuratowski. Wikipedia — Teorema de Kuratowski.
- [2] Teoria dos Grafos. Wikipedia — Teoria dos Grafos.
- [3] Grafo Planar. Wikipedia — Grafo Planar.
- [4] Busca em Profundidade. Wikipedia — Busca em Profundidade.
- [5] Algoritmo de Prim. Wikipedia — Algoritmo de Prim.
- [6] Algoritmo de Kruskal. Wikipedia — Algoritmo de Kruskal.
- [7] Algoritmo de Dijkstra. Wikipedia — Algoritmo de Dijkstra.