INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Acre

VINÍCIUS BARROS DE MELO

GIOVANNA SOUZA CORREIA

GRAFOS

Teoria e Prática

Rio Branco - AC

2025

VINÍCIUS BARROS DE MELO

GIOVANNA SOUZA CORREIA

GRAFOS

Teoria e Prática

Trabalho de grafos, apresentado ao curso de Estrutura de Dados, como requisito para obtenção de nota de atividade.

Orientador: Prof. Breno

Rio Branco - AC

2025

Resumo

A **teoria dos grafos** é um ramo da matemática discreta que estuda as estruturas chamadas **grafos**. Um grafo é composto por **vértices** (também chamados de nós) e **arestas** (ou arcos), que conectam esses vértices. Essa teoria tem aplicações em diversas áreas, como ciência da computação, redes de comunicação, biologia, redes sociais, transportes, e muito mais. Abaixo, detalho alguns dos conceitos e tipos de grafos, além de como a teoria é aplicada em várias situações.

Definição de Grafo:

- **Vértices (ou Nós)**: São os elementos do grafo, representando objetos ou entidades.
- **Arestas (ou Arcos)**: São as conexões entre os vértices, representando as relações ou interações entre os objetos.

Classificação de Grafos:

1. **Grafo Simples**:
   - Não permite **laços** (arestas que conectam um vértice a si mesmo).
   - Não permite **arestas múltiplas** entre dois vértices.
   - Pode ser **direcionado** ou **não direcionado**.
2. **Grafo Direcionado (ou Dígrafo)**:
   - As arestas têm direção, ou seja, uma aresta conecta um vértice de origem a um vértice de destino, indicado por uma seta.
   - Exemplo: redes de tráfego de dados, onde o fluxo de dados vai de um ponto a outro.
3. **Grafo Ponderado**:
   - As arestas têm um peso ou custo associado, representando, por exemplo, distâncias, custos de transporte ou tempo de espera.
   - Exemplo: um grafo ponderado pode representar uma rede de estradas com distâncias como pesos nas arestas.
4. **Grafo Não Ponderado**:
   - As arestas não têm peso associado, ou seja, todos os caminhos são considerados iguais em termos de custo ou distância.
5. **Grafo Conexo**:
   - Um grafo é **conexo** se existe um caminho entre quaisquer dois vértices, ou seja, é possível ir de um vértice a qualquer outro vértice, independentemente da sequência de arestas.
6. **Árvore**:
   - Um **grafo acíclico** e **conexo**. Não há ciclos e todos os vértices estão interconectados.
   - Toda árvore tem V−1V - 1V−1 arestas, onde VVV é o número de vértices. As árvores são fundamentais para representar hierarquias, como em estruturas de diretórios e árvores genealógicas.
7. **Grafo Bipartido**:

    - Um grafo é bipartido se os seus vértices podem ser divididos em dois conjuntos disjuntos, de forma que cada aresta conecta um vértice de um conjunto a um vértice do outro conjunto. Não há arestas entre vértices do mesmo conjunto.
    - Exemplo: um grafo bipartido pode representar um problema de **emparelhamento** entre trabalhadores e tarefas.

8. **Grafo Completo**:
    - Em um grafo completo, cada par de vértices é conectado por uma aresta. Em um grafo não direcionado, um grafo completo com nnn vértices tem n(n−1)2\frac{n(n-1)}{2}2n(n−1) arestas.

Operações em Grafos:

- **Grau de um Vértice**: O grau de um vértice é o número de arestas que incidem sobre ele. Em um grafo direcionado, o grau de um vértice pode ser dividido em **grau de entrada** (número de arestas que chegam ao vértice) e **grau de saída** (número de arestas que saem do vértice).
- **Caminho**: Um caminho em um grafo é uma sequência de vértices onde cada par de vértices consecutivos é conectado por uma aresta. Se não há repetição de vértices ou arestas, é chamado de **caminho simples**.
- **Ciclo**: Um ciclo é um caminho que começa e termina no mesmo vértice e que não passa duas vezes pela mesma aresta ou vértice, exceto no início e no fim.
- **Componente Conexo**: Em grafos não direcionados, um componente conexo é um subgrafo em que qualquer par de vértices está interligado por um caminho, e não existe nenhum vértice fora do subgrafo que tenha um caminho para algum vértice dentro.

Conceitos Avançados:

1. **Índices de Centralidade**:
    - **Centralidade de Grau**: Mede o número de conexões de um vértice.
    - **Centralidade de Closeness**: Mede o quão próximo um vértice está de todos os outros.
    - **Centralidade de Betweenness**: Mede quantas vezes um vértice está entre outros vértices em um caminho mais curto.
2. **Planaridade**:
    - Um grafo é **planar** se pode ser desenhado no plano sem que as arestas se cruzem.
    - O **Teorema de Kuratowski** estabelece que um grafo é planar se e somente se não contém um subgrafo homeomorfo a K5K_5K5 (o grafo completo de 5

vértices) ou K3,3K_{3,3}K3,3 (o grafo bipartido completo com 3 vértices em cada conjunto).

3. **Algoritmos de Caminho Mínimo**:
   - O **Algoritmo de Dijkstra** é utilizado para encontrar o caminho mais curto em um grafo ponderado com arestas de peso não negativo.
   - O **Algoritmo de Bellman-Ford** pode lidar com grafos com arestas de peso negativo.
   - O **Algoritmo de Floyd-Warshall** encontra os caminhos mais curtos entre todos os pares de vértices.
4. **Árvore Geradora Mínima (MST)**:
   - Uma árvore geradora mínima de um grafo ponderado é uma árvore que conecta todos os vértices com o menor custo total possível.
   - Os algoritmos **Kruskal** e **Prim** são usados para encontrar uma árvore geradora mínima.

Aplicações dos Grafos:

1. **Redes de Computadores**: Os grafos são usados para modelar redes de computadores, onde os vértices representam computadores ou dispositivos, e as arestas representam conexões entre eles.
2. **Redes Sociais**: As redes sociais podem ser representadas como grafos, onde os vértices são usuários e as arestas representam interações (como amizades, seguidores, etc.).
3. **Roteamento e Navegação**: O uso de grafos para encontrar as rotas mais curtas em sistemas de transporte, como rodovias, sistemas de metrô e redes de comunicação.
4. **Fluxo de Rede**: Problemas de otimização de fluxo em redes de transporte, distribuição de recursos e comunicação de dados podem ser resolvidos usando algoritmos de fluxo máximo em grafos.
5. **Biologia e Genética**: Os grafos são usados para modelar redes de interações entre proteínas, vias metabólicas, ou para representar relacionamentos evolutivos entre espécies.

1. Algoritmos e Códigos em Grafos:

O algoritmo de Busca em Largura (BFS) explora os vértices de um grafo começando de um vértice inicial e visitando todos os vizinhos, um por um.

Código em C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_VERTICES 5

typedef struct Queue {
    int items[MAX_VERTICES];
    int front, rear;
} Queue;

void initQueue(Queue* q) {
    q->front = 0;
    q->rear = -1;
}

bool isQueueEmpty(Queue* q) {
    return q->rear == -1;
}

void enqueue(Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1)
        return;
    q->items[++(q->rear)] = value;
}

int dequeue(Queue* q) {
    if (isQueueEmpty(q))
        return -1;
    int value = q->items[q->front++];
    if (q->front > q->rear)
        q->front = q->rear = -1;
    return value;
}

void bfs(int graph[MAX_VERTICES][MAX_VERTICES], int start) {
    bool visited[MAX_VERTICES] = {false};
    Queue q;
    initQueue(&q);

    visited[start] = true;
    enqueue(&q, start);

    while (!isQueueEmpty(&q)) {
        int current = dequeue(&q);
        printf("%d ", current);

        for (int i = 0; i < MAX_VERTICES; i++) {
            if (graph[current][i] == 1 && !visited[i]) {
                visited[i] = true;
                enqueue(&q, i);
            }
        }
    }
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 1, 1, 0, 0},
        {1, 0, 1, 1, 0},
        {1, 1, 0, 1, 1},
        {0, 1, 1, 0, 1},
        {0, 0, 1, 1, 0}
    };

    printf("BFS starting from vertex 0:\n");
    bfs(graph, 0);
    return 0;
}
```

2. Busca em Profundidade (DFS)

A Busca em Profundidade (DFS) explora os vértices do grafo recursivamente, indo até o fundo de cada ramo antes de retroceder.

Código em C:

```c
#include <stdio.h>
#include <stdbool.h>

#define MAX_VERTICES 5

void dfs(int graph[MAX_VERTICES][MAX_VERTICES], int vertex, bool visited[MAX_VERTICES]) {
    visited[vertex] = true;
    printf("%d ", vertex);

    for (int i = 0; i < MAX_VERTICES; i++) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            dfs(graph, i, visited);
        }
    }
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 1, 1, 0, 0},
        {1, 0, 1, 1, 0},
        {1, 1, 0, 1, 1},
        {0, 1, 1, 0, 1},
        {0, 0, 1, 1, 0}
    };

    bool visited[MAX_VERTICES] = {false};

    printf("DFS starting from vertex 0:\n");
    dfs(graph, 0, visited);
    return 0;
}
```

3. Algoritmo de Dijkstra (Caminho mais curto)

O algoritmo de Dijkstra é usado para encontrar o caminho mais curto entre dois vértices em um grafo ponderado.

Código em C:

```c
#include <stdio.h>
#include <limits.h>

#define V 4 // Número de vértices

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printf("Distâncias a partir do vértice %d:\n", src);
    for (int i = 0; i < V; i++) {
        printf("Vértice %d: %d\n", i, dist[i]);
    }
}

int main() {
    int graph[V][V] = {
        {0, 1, 4, 0},
        {1, 0, 4, 2},
        {4, 4, 0, 3},
        {0, 2, 3, 0}
    };

    dijkstra(graph, 0);
    return 0;
}
```

4. Algoritmo de Kruskal (Árvore Geradora Mínima)

O algoritmo de Kruskal é utilizado para encontrar uma árvore geradora mínima de um grafo ponderado.

Código em C:

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 4
#define MAX_ARESTAS 6

typedef struct Edge {
    int src, dest, weight;
} Edge;

int find(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return find(parent, parent[i]);
}

void unionSets(int parent[], int rank[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);

    if (rank[xroot] < rank[yroot])
        parent[xroot] = yroot;
    else if (rank[xroot] > rank[yroot])
        parent[yroot] = xroot;
    else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}

int compare(const void *a, const void *b) {
    return ((Edge *)a)->weight - ((Edge *)b)->weight;
}

void kruskal(Edge edges[], int n) {
    int parent[MAX_VERTICES];
    int rank[MAX_VERTICES] = {0};
    for (int i = 0; i < MAX_VERTICES; i++)
        parent[i] = i;

    qsort(edges, n, sizeof(Edge), compare);

    printf("Arestas da árvore geradora mínima:\n");
    for (int i = 0; i < n; i++) {
        int x = find(parent, edges[i].src);
        int y = find(parent, edges[i].dest);

        if (x != y) {
            printf("%d - %d: %d\n", edges[i].src, edges[i].dest, edges[i].weight);
            unionSets(parent, rank, x, y);
        }
    }
}

int main() {
    Edge edges[MAX_ARESTAS] = {
        {0, 1, 10}, {0, 2, 6}, {0, 3, 5},
        {1, 3, 15}, {2, 3, 4}
    };

    kruskal(edges, MAX_ARESTAS);
    return 0;
}
```