Centro Paula Souza

Fatec Jahu

GRAFOS PONDERADOS

ESTRUTURA DE DADOS

CST em Desenvolvimento de Software Multiplataforma

PROF. Me. TIAGO A. SILVA

LIVRO DE REFERÊNCIA DA DISCIPLINA

- **BIBLIOGRAFIA BÁSICA:**
  - GRONER, Loiane. **Estrutura de dados e algoritmos com JavaScript**: escreva um código JavaScript complexo e eficaz usando a mais recente ECMASript. **São Paulo: Novatec Editora**, 2019.
- **NESTA AULA:**
  - **Capítulo 12 – Grafos**
    - Matriz de Adjacência
    - Busca em Largura (BFS)
    - Busca em Profundidade (DFS)
    - Algoritmo de Dijkstra

PARA SOBREVIVER AO JAVASCRIPT

Non-zero value

0

null

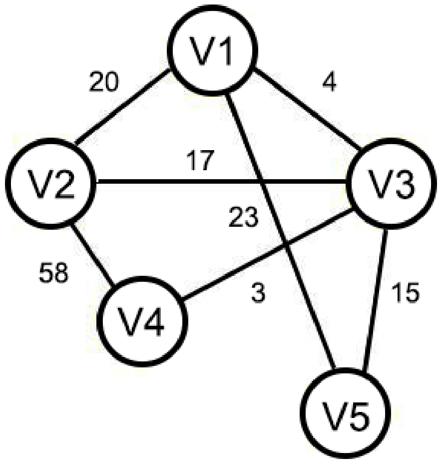
undefined

O QUE SÃO GRAFOS PONDERADOS?

- Um grafo ponderado (ou grafo com pesos) é um tipo de grafo em que cada aresta possui um valor associado, chamado de peso ou custo.
- Esse peso pode representar:
  - Distância entre cidades
  - Tempo de viagem
  - Custo de envio
  - Largura de banda
  - Qualquer outra medida associada à conexão entre dois nós
- Em um grafo ponderado, cada aresta (ligação entre dois vértices) tem uma informação extra: **o peso**.
- Se o grafo for direcionado, o peso só vale na direção da aresta.

O QUE SÃO GRAFOS PONDERADOS?

Matriz de Adjacência
grafo ponderado



	V1	V2	V3	V4	V5
V1	0*	20	4	0*	23
V2	20	0*	17	58	0*
V3	4	17	0*	3	15
V4	0*	58	3	0*	0*
V5	23	0*	15	0*	0*

*zero é um valor escolhido em código para considerar não ter nenhuma ligação entre os dois grafos, porém se seu grafos tiver zero como um valor valido deve se escolher outro valor.

Imagem: Paulo Martins

MÉTODO CONSTRUTOR

```
class GrafoPonderado {

    constructor() {
        // Conjunto de vértices únicos
        this.vertices = new Set();

        // Mapa onde cada vértice aponta para uma lista de objetos: { vertice, peso }
        this.adjacencia = new Map();
    }
```

ADICIONAR VÉRTICES

```
    // Adiciona um novo vértice ao grafo. Se já existir, nada é feito.
    //  Também inicializa sua lista de adjacência.
    adicionarVertice(v) {

        // Garante que o vértice está no conjunto
        this.vertices.add(v);
        if (!this.adjacencia.has(v)) {

            // Inicializa lista de vizinhos se ainda não existir
            this.adjacencia.set(v, []);
        }
    }
```

ADICIONAR ARESTAS

```
    // Adiciona uma aresta ponderada entre dois vértices.
    // Cria os vértices caso ainda não existam. Por padrão,
    // é um grafo direcionado.
    adicionarAresta(origem, destino, peso) {
        if (!this.adjacencia.has(origem)) this.adicionarVertice(origem);
        if (!this.adjacencia.has(destino)) this.adicionarVertice(destino);

        this.adjacencia.get(origem).push({ vertice: destino, peso });

        // Se o grafo for não-direcionado, descomente a linha abaixo:
        // this.adjacencia.get(destino).push({ vertice: origem, peso });
    }
```

IMPRIMIR O GRAFO - VISUALIZAÇÃO

```
    // Mostra a representação do grafo como lista de adjacência,
    // com os pesos visíveis.
    imprimirGrafo() {
        for (const [v, vizinhos] of this.adjacencia.entries()) {
            const lista = vizinhos.map(obj => `${obj.vertice}(${obj.peso})`).join(', ');
            console.log(`${v} -> ${lista}`);
        }
    }
```

IMPLEMENTAÇÃO DA MATRIZ DE ADJACÊNCIA

```javascript
    // Gera e imprime a matriz de adjacência do grafo.
    // Usa Infinity para representar ausência de aresta.
    imprimirMatrizAdjacencia() {
        const vertices = Array.from(this.vertices);
        const n = vertices.length;
        const matriz = Array.from({ length: n }, () => Array(n).fill(Infinity));

        vertices.forEach((v, i) => {
            matriz[i][i] = 0; // distância para si mesmo = 0
            for (const vizinho of this.adjacencia.get(v)) {
                const j = vertices.indexOf(vizinho.vertice);
                matriz[i][j] = vizinho.peso;
            }
        });

        console.log('Matriz de Adjacência (valores ∞ representam ausência de aresta):');
        console.log('   ', vertices.join('  '));
        matriz.forEach((linha, i) => {
            console.log(vertices[i], linha.map(x => x === Infinity ? '∞' : x).join('  '));
        });
    }
```

O QUE É MATRIZ DE ADJACÊNCIA?

- A matriz de adjacência é uma matriz (tabela) bidimensional usada para representar as conexões entre os vértices (nós) de um grafo.

- Se houver uma aresta entre o vértice i e o vértice j, então a posição matriz[i][j] recebe o valor 1 (ou o peso, se for um grafo ponderado).

- Se não houver aresta, o valor é 0 (ou Infinity ou null, dependendo do caso).

BUSCA EM PROFUNDIDADE

```javascript
    // Busca em Profundidade (Depth-First Search)
    // Realiza uma busca em profundidade, visitando
    // o vértice inicial e seus vizinhos recursivamente
    // até esgotar os caminhos.
    dfs(inicio) {
        const visitados = new Set();
        const resultado = [];

        const visitar = (v) => {
            visitados.add(v);
            resultado.push(v);

            for (const vizinho of this.adjacencia.get(v)) {
                if (!visitados.has(vizinho.vertice)) {
                    visitar(vizinho.vertice);
                }
            }
        };
```

O QUE É BUSCA EM PROFUNDIDADE?

- A Busca em Profundidade percorre um grafo indo o mais fundo possível em cada caminho antes de voltar e explorar outras possibilidades. Ela é feita de forma recursiva ou com o uso de uma pilha.
- É útil para:
  - Explorar todos os vértices alcançáveis de um ponto.
  - Detectar ciclos.
  - Determinar componentes conexos.
  - Resolver labirintos.
  - Construir árvores de espalhamento.

BUSCA EM LARGURA

```
// Busca em Largura (Breadth-First Search)
// Realiza uma busca em largura, explorando
// primeiro os vizinhos mais próximos, usando uma fila.
bfs(inicio) {
    const visitados = new Set();
    const fila = [inicio];
    const resultado = [];

    visitados.add(inicio);

    while (fila.length > 0) {
        const atual = fila.shift();
        resultado.push(atual);

        for (const vizinho of this.adjacencia.get(atual))
            if (!visitados.has(vizinho.vertice)) {
                visitados.add(vizinho.vertice);
                fila.push(vizinho.vertice);
            }
        }
    }

    console.log('BFS:', resultado.join(' -> '));
}
```

O QUE É BUSCA EM LARGURA?

- A Busca em Largura é um algoritmo de exploração de grafos que visita os vértices em camadas, ou seja, primeiro visita todos os vizinhos do vértice inicial, depois os vizinhos desses vizinhos, e assim por diante.
- Ela é ideal para:
  - Descobrir o menor número de passos até um nó (em grafos não ponderados).
  - Explorar todos os nós acessíveis a partir de um ponto.
  - Verificar conectividade.
- Conceitos fundamentais
  - Fila (queue): usada para armazenar os próximos vértices a serem visitados, na ordem em que foram descobertos.
  - Visitado: estrutura (normalmente um objeto ou vetor booleano) usada para evitar visitar o mesmo vértice duas vezes.

COMPARAÇÃO DFS E BFS

	BFS (BUSCA EM LARGURA)	DFS (BUSCA EM PROFUNDIDADE)
Estrutura usada:	Fila	Pilha (ou recursão)
Visita:	Camada por camada	Vai até o fundo e volta
Útil para:	Menor caminho (sem peso)	Detectar ciclos, labirintos
Ordem de visita:	Mais ampla	Mais profunda

IMPLEMENTAÇÃO DO ALGORITMO DE DIJKSTRA

```javascript
// Algoritmo de Dijkstra para encontrar o caminho mais curto
// Calcula as menores distâncias entre o vértice inicial e
// todos os demais, com base nos pesos das arestas.
// Usa a abordagem clássica de Dijkstra.
dijkstra(inicio) {
    const distancias = {};
    const anteriores = {};
    const naoVisitados = new Set(this.vertices);

    for (const v of this.vertices) {
        distancias[v] = Infinity;
        anteriores[v] = null;
    }
    distancias[inicio] = 0;

    while (naoVisitados.size > 0) {
        // Encontra o vértice não visitado com a menor distância conhecida
        const atual = [...naoVisitados].reduce((a, b) =>
            distancias[a] < distancias[b] ? a : b
        );
        naoVisitados.delete(atual);

        // Atualiza distâncias para vizinhos
        for (const vizinho of this.adjacencia.get(atual)) {
            const alt = distancias[atual] + vizinho.peso;
            if (alt < distancias[vizinho.vertice]) {
                distancias[vizinho.vertice] = alt;
                anteriores[vizinho.vertice] = atual;
            }
        }
    }
```

IMPLEMENTAÇÃO DO ALGORITMO DE DIJKSTRA - FINALIZAÇÃO

```
        
        // Exibe o resultado
        console.log(`Menores distâncias a partir de ${inicio}:`);
        for (const v of this.vertices) {
            console.log(`${v}: ${distancias[v]}`);
        }
    } // Fecha dijkstra
} // Fecha GrafoPonderado

// Exporta a classe para uso em outros módulos
module.exports = GrafoPonderado;
```

EXEMPLO DE USO DA CLASSE

Importando GrafoPonderado e usando os métodos

EXEMPLO DE USO DA CLASSE

```javascript
const GrafoPonderado = require('./GrafoPonderado.js');

// Exemplo de uso:
const grafo = new GrafoPonderado();
grafo.adicionarAresta('A', 'B', 2);
grafo.adicionarAresta('A', 'C', 5);
grafo.adicionarAresta('B', 'C', 1);
grafo.adicionarAresta('B', 'D', 4);
grafo.adicionarAresta('C', 'D', 2);

grafo.imprimirGrafo();
grafo.imprimirMatrizAdjacencia();
grafo.dfs('A');
grafo.bfs('A');
grafo.dijkstra('A');
```

EXERCÍCIOS

Use os métodos da classe GrafoPonderado

EXERCÍCIO 1

- Na Cidade dos Gnomos, as ruas conectam casas mágicas com diferentes distâncias encantadas (pesos).
  - Crie o grafo a seguir e:
  - Imprima a lista de adjacência.
  - Imprima a matriz de adjacência.
  - Use DFS e BFS a partir da Casa A.
  - Use Dijkstra a partir da Casa A para saber o caminho mais rápido até a Casa E.
- Ruas mágicas (arestas):
  - A → B (3)
  - A → C (2)
  - B → D (4)
  - C → D (1)
  - D → E (5)

EXERCÍCIO 2

- Um trem precisa cruzar uma rede ferroviária entre cidades com diferentes tempos de viagem. Construa o grafo a seguir e responda:
  - Qual a menor distância de São Paulo até Porto Alegre?
  - Mostre os percursos em DFS e BFS a partir de São Paulo.

- Conexões ferroviárias:
  - São Paulo → Campinas (1)
  - Campinas → Curitiba (4)
  - São Paulo → Curitiba (2)
  - Curitiba → Florianópolis (3)
  - Florianópolis → Porto Alegre (2)

EXERCÍCIO 3

- Um entregador de sucos precisa traçar a melhor rota entre lojas para entregar os pedidos rapidamente. Use Dijkstra para descobrir o menor tempo de entrega de sucos de Loja A até Loja F.
- Rotas:
  - A → B (1)
  - A → C (4)
  - B → D (2)
  - C → D (1)
  - D → E (3)
  - E → F (2)
- Desafios:
  - Liste os caminhos visitados em DFS e BFS a partir de A.
  - Qual a menor distância de A até F?

EXERCÍCIO 4

- Um mago precisa viajar por reinos para encontrar o pergaminho sagrado no Reino Z. O grafo representa portais mágicos com o custo de energia (peso) para usá-los.
- Portais mágicos:
  - X → Y (6)
  - X → W (2)
  - W → Y (2)
  - Y → Z (3)
  - W → Z (7)
- Objetivos:
  - Mostre a matriz de adjacência.
  - Calcule o caminho com menor custo de energia de X até Z.
  - Compare os caminhos encontrados em DFS e BFS a partir de X.

EXERCÍCIO 5

- Um personagem está em um labirinto com túneis de diferentes dificuldades (pesos). Ele precisa encontrar o caminho mais fácil até a saída.
- Túneis do labirinto:
  - Entrada → A (2)
  - A → B (2)
  - B → Saída (1)
  - Entrada → C (5)
  - C → Saída (1)
- Tarefas:
  - Modele esse labirinto como um grafo.
  - Use dijkstra('Entrada') para descobrir a melhor rota até 'Saída'.
  - Compare com os caminhos encontrados por DFS e BFS.

OBRIGADO!

- Encontre este **material on-line** em:
  - Slides: Plataforma Microsoft Teams

- Em caso de **dúvidas**, entre em contato:
  - **Prof. Tiago**: tiago.silva238@fatec.sp.gov.br