Relatório Detalhado sobre Teoria dos Grafos

Este documento apresenta uma visão abrangente sobre a teoria dos grafos, cobrindo sua origem histórica, os problemas clássicos que ela modela, algoritmos fundamentais para sua análise, suas vastas aplicações no mundo real e exemplos de implementação em TypeScript.

Origem e História da Teoria dos Grafos

A teoria dos grafos, um ramo fundamental da matemática discreta e da ciência da computação, possui uma origem fascinante e relativamente recente na história da matemática, datando do século XVIII. Sua gênese está intrinsecamente ligada a um problema prático e popular que intrigava os habitantes da cidade de Königsberg, na Prússia (atual Kaliningrado, Rússia).

A cidade de Königsberg era cortada pelo Rio Prególia, que formava duas grandes ilhas no seu curso. Para conectar as diferentes partes da cidade (as duas margens e as duas ilhas), existiam, na época, sete pontes. Um passatempo popular entre os moradores era tentar encontrar um percurso que atravessasse cada uma das sete pontes exatamente uma vez, sem repetir nenhuma. Apesar das muitas tentativas, ninguém conseguia realizar tal feito, e a questão se tornou uma espécie de lenda urbana.

O problema chegou ao conhecimento do célebre matemático suíço Leonhard Euler. Em 1736, Euler publicou um artigo intitulado "Solutio problematis ad geometriam situs pertinentis" (Solução de um problema relativo à geometria de posição), no qual não apenas demonstrou rigorosamente a impossibilidade de realizar o passeio desejado, mas também estabeleceu as bases para um novo campo da matemática: a teoria dos grafos. Este trabalho é amplamente considerado o marco inicial desta disciplina [O1, O3].

Para analisar o problema, Euler adotou uma abordagem inovadora. Ele abstraiu a complexidade geográfica da cidade, representando as quatro regiões de terra (as duas margens e as duas ilhas) como pontos (vértices) e as sete pontes como linhas (arestas) conectando esses pontos. Essa representação esquemática, possivelmente o primeiro grafo formalmente estudado na história [O3], permitiu a Euler analisar a estrutura essencial do problema.

Euler percebeu que a possibilidade de percorrer um caminho passando por cada aresta (ponte) exatamente uma vez dependia do número de arestas que incidiam em cada vértice (região de terra). Ele definiu o "grau" de um vértice como o número de arestas conectadas a ele. Em sua análise, Euler concluiu que para um percurso que atravessa cada aresta exatamente uma vez ser possível (hoje conhecido como caminho euleriano ou trilha euleriana), o grafo correspondente poderia ter no máximo dois vértices de grau ímpar. Isso ocorre porque, ao entrar e sair de um vértice intermediário do percurso, utiliza-se um par de arestas. Apenas os vértices de início e fim do percurso poderiam ter um número ímpar de arestas incidentes. Se todos os vértices tivessem grau par, o percurso poderia começar e terminar no mesmo vértice (um ciclo euleriano) [O1, O3].

No caso específico das pontes de Königsberg, Euler verificou que todas as quatro regiões de terra (vértices) possuíam um número ímpar de pontes (arestas) conectadas a elas. Como havia mais de dois vértices com grau ímpar, Euler provou matematicamente que era impossível realizar o passeio desejado, confirmando a intuição popular [O1, O3].

A solução de Euler para o problema das pontes de Königsberg foi revolucionária não apenas por resolver um enigma local, mas principalmente por introduzir conceitos abstratos como vértices, arestas e graus, e por focar nas propriedades relacionais e estruturais do problema, independentemente da sua geometria exata. Isso marcou o nascimento da teoria dos grafos e da topologia.

Desde então, a teoria dos grafos evoluiu enormemente, encontrando vastas aplicações em diversas áreas do conhecimento e da tecnologia, como ciência da computação (redes de computadores, algoritmos, estruturas de dados), engenharia (circuitos elétricos, logística de transporte), ciências sociais (redes sociais), biologia (redes de interação de proteínas), química (modelagem molecular) e muitas outras [O1]. O estudo de grafos permite modelar e resolver problemas complexos envolvendo relações e conexões entre entidades.

Problemas Clássicos Modelados e Resolvidos com Grafos

A abstração poderosa fornecida pela teoria dos grafos permite modelar uma vasta gama de problemas do mundo real, transformando cenários complexos em estruturas de vértices e arestas que podem ser analisadas matematicamente. A capacidade de representar relações e conexões torna os grafos ferramentas indispensáveis em diversas disciplinas, desde a otimização de rotas logísticas até a análise de redes sociais e

biológicas. Diversos problemas clássicos encontraram soluções eficientes através da aplicação de algoritmos de grafos [P1, P2].

Um dos problemas mais fundamentais é o **problema do caminho mínimo**. Dada uma rede (representada como um grafo, possivelmente ponderado), o objetivo é encontrar o caminho mais curto (em termos de distância, custo, tempo, etc.) entre dois vértices específicos. Aplicações são ubíquas: sistemas de navegação GPS calculam a rota mais rápida entre dois pontos, roteadores de internet determinam o caminho mais eficiente para pacotes de dados, e planejadores logísticos otimizam rotas de entrega [P1]. A rede de relacionamentos mencionada anteriormente, onde se busca o menor número de conexões para apresentar duas pessoas, é um exemplo claro deste tipo de problema [P1].

Outra classe importante de problemas envolve encontrar a **árvore geradora mínima (MST - Minimum Spanning Tree)**. Dado um grafo conexo e ponderado, busca-se um subgrafo que seja uma árvore (sem ciclos), conecte todos os vértices originais e tenha a menor soma possível dos pesos das arestas. Este problema é crucial no design de redes eficientes e de baixo custo, como a construção de redes de energia elétrica, oleodutos, redes de telecomunicações ou sistemas de irrigação, onde o objetivo é conectar todos os pontos minimizando o custo total de instalação (cabos, tubulações, etc.) [P2].

Problemas de **conectividade** também são centrais na teoria dos grafos. Questões como verificar se um grafo é conexo (se existe um caminho entre qualquer par de vértices), encontrar componentes conexas (subgrafos máximos conexos), ou identificar vértices ou arestas cuja remoção desconectaria o grafo (pontos de articulação e pontes) são vitais para analisar a robustez e a vulnerabilidade de redes, sejam elas de comunicação, transporte ou infraestrutura [P2].

O estudo de **ciclos e caminhos especiais** em grafos, como os caminhos e ciclos eulerianos (que visitam cada aresta exatamente uma vez, remetendo ao problema original das pontes de Königsberg) e os caminhos e ciclos hamiltonianos (que visitam cada vértice exatamente uma vez), tem aplicações em áreas como planejamento de rotas (por exemplo, para inspeção de ruas ou coleta de lixo), otimização de processos industriais e até mesmo em bioinformática, na montagem de sequências de DNA [P2].

A **coloração de grafos** é outro problema clássico com aplicações práticas significativas. O problema consiste em atribuir cores aos vértices de um grafo de forma que vértices adjacentes tenham cores diferentes, utilizando o menor número possível de cores (o número cromático do grafo). Aplicações incluem a alocação de frequências para estações de rádio ou TV para evitar interferências, o agendamento de tarefas ou exames onde recursos compartilhados (salas, professores) não podem ser usados simultaneamente por tarefas conflitantes, e a coloração de mapas geográficos [P2, P3].

Finalmente, os problemas de **fluxo em redes** modelam situações onde alguma substância ou recurso (água, tráfego, dados, mercadorias) flui através de uma rede (grafo direcionado com capacidades nas arestas). O objetivo é frequentemente maximizar o fluxo total de uma fonte para um sumidouro, respeitando as capacidades das arestas. Isso tem aplicações diretas em logística, otimização de redes de transporte, planejamento de produção e análise de capacidade de redes de comunicação [P2]. Problemas de **emparelhamento (matching)**, que buscam encontrar conjuntos de arestas sem vértices em comum (por exemplo, para designar trabalhadores a tarefas ou formar pares), também são frequentemente estudados no contexto de grafos e fluxos [P2].

A riqueza e a variedade desses problemas demonstram a versatilidade da teoria dos grafos como uma ferramenta de modelagem e solução para desafios complexos em inúmeros domínios.

Algoritmos Fundamentais em Grafos

A resolução dos diversos problemas modelados pela teoria dos grafos depende de algoritmos eficientes que exploram a estrutura de conexões entre vértices e arestas. Três dos algoritmos mais fundamentais e amplamente utilizados são o Algoritmo de Dijkstra, a Busca em Largura (BFS - Breadth-First Search) e a Busca em Profundidade (DFS - Depth-First Search). Cada um possui características e aplicações distintas.

Algoritmo de Dijkstra

Desenvolvido por Edsger W. Dijkstra em 1956 e publicado em 1959, o Algoritmo de Dijkstra é um dos algoritmos mais célebres para resolver o problema do caminho mínimo de fonte única em grafos ponderados [A1, A2]. Dado um vértice de origem, o algoritmo encontra o caminho de menor custo (ou menor distância, menor tempo, etc.) deste vértice para todos os outros vértices alcançáveis no grafo. Uma restrição crucial para o funcionamento correto do algoritmo original é que todos os pesos (custos) das arestas devem ser não negativos [A2].

O algoritmo opera de forma iterativa, mantendo um conjunto de vértices já visitados (para os quais o caminho mínimo a partir da origem já foi determinado) e um conjunto de vértices não visitados. Inicialmente, apenas o vértice de origem está no conjunto de visitados, e sua distância para si mesmo é definida como 0, enquanto a distância para todos os outros vértices é considerada infinita [A2].

A cada passo, o algoritmo seleciona o vértice não visitado que possui a menor distância atualmente conhecida a partir da origem. Este vértice é então marcado como visitado e adicionado à árvore de caminhos mínimos que está sendo construída. Em seguida, o algoritmo atualiza as distâncias dos vizinhos não visitados do vértice recém-visitado. Se o caminho até um vizinho passando pelo vértice atual for mais curto do que a distância previamente registrada para esse vizinho, a distância é atualizada [A1, A2]. Esse processo de selecionar o vértice não visitado mais próximo e relaxar as arestas de seus vizinhos continua até que todos os vértices alcançáveis tenham sido visitados, ou até que o vértice de destino (se houver um específico) seja alcançado.

Para implementar eficientemente a seleção do vértice não visitado com menor distância, geralmente se utiliza uma estrutura de dados como uma fila de prioridade (min-heap). Isso permite encontrar o vértice mínimo e atualizar as distâncias de forma mais rápida do que uma busca linear [A1].

A principal aplicação do Algoritmo de Dijkstra é, naturalmente, o cálculo de rotas em sistemas de navegação (GPS), onde os vértices representam interseções ou locais e as arestas representam ruas ou estradas com pesos correspondentes à distância ou ao tempo de percurso [A2]. Ele também é fundamental em roteamento de redes de computadores (como no protocolo OSPF - Open Shortest Path First), análise de redes sociais para encontrar graus de separação, e em diversas outras áreas onde a otimização de caminhos é necessária.

Sua limitação mais significativa é a incapacidade de lidar corretamente com arestas de peso negativo. Se um grafo contiver ciclos de peso negativo, o conceito de caminho mínimo pode nem mesmo ser bem definido. Para grafos com arestas de peso negativo, mas sem ciclos negativos, outros algoritmos como o de Bellman-Ford devem ser utilizados.

Busca em Largura (BFS - Breadth-First Search)

A Busca em Largura (BFS) é outro algoritmo fundamental para percorrer ou buscar em grafos. Diferente do Dijkstra, que prioriza o custo acumulado, a BFS explora o grafo camada por camada a partir de um vértice de origem especificado [A3, A4]. Ela visita primeiro o vértice de origem, depois todos os seus vizinhos diretos (a uma distância de 1 aresta), depois todos os vizinhos destes que ainda não foram visitados (a uma distância de 2 arestas), e assim sucessivamente [A3].

Para gerenciar a ordem de visitação, a BFS utiliza uma estrutura de dados do tipo fila (FIFO - First-In, First-Out). O algoritmo começa colocando o vértice de origem na fila. Em cada passo, ele remove um vértice da frente da fila, visita-o (marca como visitado) e adiciona todos os seus vizinhos não visitados ao final da fila [A3, A4]. O processo

continua até que a fila esteja vazia, o que significa que todos os vértices alcançáveis a partir da origem foram visitados.

Uma das propriedades mais importantes da BFS é que, em grafos não ponderados (ou com pesos uniformes), ela encontra o caminho mais curto em termos de número de arestas entre o vértice de origem e todos os outros vértices alcançáveis [A4, A5]. Isso ocorre porque a exploração em camadas garante que os vértices mais próximos (em número de arestas) sejam descobertos primeiro. Ao encontrar um vértice, a BFS estabelece o caminho com o menor número de arestas até ele a partir da origem.

Devido a essa propriedade, a BFS é amplamente utilizada para encontrar o caminho mais curto em grafos não ponderados, como em redes sociais para determinar o menor número de conexões entre duas pessoas (graus de separação), em jogos para encontrar o menor número de movimentos, ou em redes de computadores para encontrar caminhos com o menor número de saltos (hops) [A4]. Outras aplicações incluem a descoberta de componentes conexas em um grafo, a implementação de web crawlers que exploram páginas web nível por nível, e algoritmos de broadcast em redes.

A BFS, assim como o Dijkstra, constrói implicitamente uma árvore de busca (árvore BFS) que contém os caminhos mais curtos (em número de arestas) da origem para todos os outros nós alcançáveis [A3].

Busca em Profundidade (DFS - Depth-First Search)

A Busca em Profundidade (DFS) oferece uma estratégia de exploração de grafos contrastante com a BFS. Em vez de explorar camada por camada, a DFS avança o máximo possível ao longo de um caminho antes de retroceder (backtracking) para explorar outros caminhos [A6, A7]. Iniciando em um vértice de origem, a DFS explora uma aresta que sai dele, chegando a um novo vértice. A partir deste novo vértice, ela continua explorando uma aresta de saída para um vértice ainda não visitado, aprofundando-se no grafo. Esse processo continua até que se atinja um vértice sem vizinhos não visitados ou um vértice cujos vizinhos já foram todos explorados a partir dele. Nesse ponto, o algoritmo retrocede para o vértice anterior e tenta explorar outro caminho não visitado [A6].

A implementação clássica da DFS utiliza recursão, que gerencia implicitamente a ordem de visitação e o backtracking através da pilha de chamadas de função. Alternativamente, pode-se usar uma pilha explícita (LIFO - Last-In, First-Out) para controlar os vértices a serem visitados [A6]. A DFS marca os vértices como visitados para evitar ciclos infinitos e reprocessamento.

Durante a execução, a DFS pode registrar informações valiosas, como os tempos de descoberta e finalização de cada vértice. O tempo de descoberta é quando um vértice é visitado pela primeira vez, e o tempo de finalização é quando a exploração a partir daquele vértice (incluindo todos os seus descendentes na árvore DFS) está completa [A1]. Essas informações permitem classificar as arestas do grafo em arestas de árvore (pertencentes à floresta DFS gerada), arestas de retorno (conectando um vértice a um ancestral na árvore DFS, indicando um ciclo), arestas de avanço (conectando um vértice a um descendente não-filho na árvore DFS) e arestas de cruzamento (conectando vértices que não têm relação ancestral/descendente direta) [A1].

A DFS é particularmente útil para várias tarefas em grafos:

- **Detecção de Ciclos:** A presença de uma aresta de retorno durante a DFS indica a existência de um ciclo no grafo [A1].
- **Ordenação Topológica:** Em Grafos Acíclicos Direcionados (DAGs), a DFS pode ser usada para produzir uma ordenação linear dos vértices tal que, para toda aresta direcionada de `u` para `v`, `u` vem antes de `v` na ordenação. Isso é crucial em planejamento de tarefas com dependências [A1].
- **Componentes Fortemente Conexos:** Em grafos direcionados, algoritmos baseados em DFS (como o de Tarjan ou Kosaraju) podem encontrar conjuntos de vértices onde qualquer vértice pode alcançar qualquer outro vértice dentro do mesmo conjunto.
- **Componentes Conexas:** Em grafos não direcionados, a DFS pode ser usada para encontrar todos os vértices alcançáveis a partir de um ponto de partida, identificando assim as componentes conexas do grafo [A6].
- **Resolução de Labirintos e Puzzles:** A natureza exploratória da DFS é adequada para encontrar caminhos em estruturas semelhantes a labirintos [A6].

A DFS gera uma floresta de busca em profundidade (uma coleção de árvores DFS), pois pode ser necessário reiniciar a busca a partir de um vértice não visitado se o grafo não for conexo ou se a busca inicial não alcançar todos os vértices [A6].

Aplicações Reais da Teoria dos Grafos

A capacidade da teoria dos grafos de modelar relações e estruturas complexas confere-lhe uma aplicabilidade extraordinariamente vasta, permeando inúmeras áreas da ciência, tecnologia e do cotidiano. Desde a otimização de rotas de transporte até a análise de interações moleculares, os grafos fornecem uma linguagem comum e ferramentas poderosas para resolver problemas práticos [AP1, AP2].

Redes de Computadores e a Internet: Talvez uma das aplicações mais evidentes seja na modelagem da internet e de redes de computadores. A própria World Wide Web pode ser vista como um imenso grafo direcionado, onde as páginas web são os vértices e os hiperlinks são as arestas direcionadas [AP1]. Algoritmos de busca como o PageRank do Google utilizam a estrutura deste grafo para determinar a relevância das páginas. Em redes locais e na internet, roteadores (vértices) e as conexões entre eles (arestas) formam grafos. Algoritmos como Dijkstra e Bellman-Ford são usados para encontrar os caminhos mais eficientes (menor latência, maior largura de banda) para o tráfego de dados (protocolos de roteamento como OSPF e BGP) [AP3]. A análise de conectividade e a identificação de gargalos também são cruciais para garantir a robustez da rede.

Logística e Transportes: A otimização de rotas é um problema clássico resolvido com grafos. Empresas de logística utilizam grafos para modelar redes de distribuição, onde armazéns e clientes são vértices e as rotas de transporte são arestas ponderadas por distância, tempo ou custo. Algoritmos de caminho mínimo (Dijkstra) ajudam a planejar as rotas de entrega mais eficientes [AP1, AP4]. Redes de transporte público (metrô, ônibus) e malhas aéreas também são modeladas como grafos para planejamento de horários, cálculo de tarifas e análise de fluxo de passageiros [AP1, AP4]. O Problema do Caixeiro Viajante (TSP), embora computacionalmente difícil, busca a rota mais curta que visita um conjunto de cidades (vértices) exatamente uma vez, sendo um exemplo extremo de otimização logística baseada em grafos.

Redes Sociais: Plataformas como Facebook, LinkedIn e Twitter podem ser representadas como grafos, onde usuários são vértices e conexões (amizade, seguir, etc.) são arestas (direcionadas ou não). A análise desses grafos permite identificar comunidades de usuários com interesses comuns, encontrar influenciadores (vértices com alto grau ou centralidade), recomendar conexões e analisar a disseminação de informações ou tendências [AP1]. Algoritmos de busca como BFS são usados para calcular os "graus de separação" entre usuários.

Biologia e Bioinformática: Grafos são ferramentas essenciais na biologia moderna. Redes de interação proteína-proteína (PPI) modelam como as proteínas interagem dentro de uma célula, ajudando a entender funções celulares e doenças. Redes metabólicas representam as reações químicas que ocorrem em um organismo. Em genômica, grafos de De Bruijn são usados na montagem de sequências de DNA a partir de fragmentos menores. A análise de redes neurais no cérebro também se beneficia da teoria dos grafos.

Engenharia e Infraestrutura: O design e análise de circuitos elétricos utilizam grafos para representar componentes e conexões. A análise da robustez de redes de energia elétrica, identificando pontos críticos cuja falha poderia causar apagões, é feita usando conceitos de conectividade de grafos [AP1, AP4]. O planejamento de redes de

saneamento, telecomunicações e outras infraestruturas também se baseia em encontrar estruturas eficientes, como árvores geradoras mínimas, para conectar pontos com custo mínimo.

Química e Ciência dos Materiais: Moléculas podem ser representadas como grafos, onde átomos são vértices e ligações químicas são arestas. A teoria dos grafos ajuda a analisar estruturas moleculares, prever propriedades e identificar isômeros (moléculas com a mesma fórmula química, mas estruturas diferentes).

Ciência da Computação: Além das redes, grafos são fundamentais em muitas outras áreas da computação. Compiladores usam grafos de fluxo de controle e grafos de dependência de dados para otimizar código. A alocação de registradores pode ser modelada como um problema de coloração de grafos [AP2]. Estruturas de dados como árvores (um tipo especial de grafo) são onipresentes. Bancos de dados orientados a grafos estão se tornando populares para gerenciar dados altamente conectados.

Outras Aplicações: A teoria dos grafos também encontra aplicações em pesquisa operacional (problemas de agendamento, alocação de recursos), teoria dos jogos, epidemiologia (modelagem da disseminação de doenças), ecologia (redes alimentares), linguística (análise de relações semânticas) e até mesmo em humanidades (análise de redes de personagens em literatura).

A diversidade dessas aplicações demonstra como a abstração matemática dos grafos fornece um framework poderoso e unificador para entender e resolver problemas complexos em quase todas as áreas do conhecimento humano.

Implementando Grafos em TypeScript

Após explorar a teoria, os problemas, algoritmos e aplicações dos grafos, vamos agora ver como podemos representar e manipular essas estruturas em código, utilizando a linguagem TypeScript. TypeScript, com seu sistema de tipos estáticos, oferece uma boa base para definir estruturas de dados claras e robustas como os grafos.

A escolha da representação do grafo é crucial e impacta a eficiência dos algoritmos. Duas representações comuns são a Matriz de Adjacências e a Lista de Adjacências. A Lista de Adjacências é frequentemente preferida para grafos esparsos (com relativamente poucas arestas), pois ocupa menos espaço e permite iterar sobre os vizinhos de um vértice de forma mais eficiente. Vamos adotar a Lista de Adjacências para nossos exemplos.

Representação com Lista de Adjacências

Nesta representação, mantemos um mapa (ou dicionário) onde as chaves são os identificadores dos vértices. O valor associado a cada chave é uma lista contendo os identificadores dos vértices adjacentes.

```
// Interface para representar um grafo não direcionado e não ponderado
interface IGraph<T> {
  addVertex(vertex: T): void;
  addEdge(vertex1: T, vertex2: T): void;
  getNeighbors(vertex: T): T[];
  getVertices(): T[];
  printGraph(): void;
}

// Implementação da classe Graph usando Lista de Adjacências
class Graph<T> implements IGraph<T> {
  private adjacencyList: Map<T, T[]> = new Map();

  /**
   * Adiciona um novo vértice ao grafo.
   * Se o vértice já existir, nenhuma ação é tomada.
   * @param vertex O vértice a ser adicionado.
   */
  addVertex(vertex: T): void {
    if (!this.adjacencyList.has(vertex)) {
      this.adjacencyList.set(vertex, []);
    }
  }

  /**
   * Adiciona uma aresta não direcionada entre dois vértices.
   * Se os vértices não existirem, eles são adicionados primeiro.
   * @param vertex1 O primeiro vértice.
   * @param vertex2 O segundo vértice.
   */
  addEdge(vertex1: T, vertex2: T): void {
    // Garante que ambos os vértices existam no grafo
    this.addVertex(vertex1);
    this.addVertex(vertex2);

    // Adiciona a aresta (não direcionada)
    // Evita adicionar arestas duplicadas se já existir
    if (!this.adjacencyList.get(vertex1)?.includes(vertex2)) {
        this.adjacencyList.get(vertex1)?.push(vertex2);
    }
    if (!this.adjacencyList.get(vertex2)?.includes(vertex1)) {
        this.adjacencyList.get(vertex2)?.push(vertex1);
```

```
    }
  }

  /**
   * Retorna a lista de vizinhos de um determinado vértice.
   * @param vertex O vértice cujos vizinhos são desejados.
   * @returns Um array com os vizinhos, ou um array vazio se o
vértice não existir.
   */
  getNeighbors(vertex: T): T[] {
    return this.adjacencyList.get(vertex) || [];
  }

  /**
   * Retorna uma lista com todos os vértices do grafo.
   * @returns Um array com todos os vértices.
   */
  getVertices(): T[] {
    return Array.from(this.adjacencyList.keys());
  }

  /**
   * Imprime a representação da lista de adjacências do grafo no
console.
   */
  printGraph(): void {
    console.log("Representação do Grafo (Lista de
Adjacências):");
    this.adjacencyList.forEach((neighbors, vertex) => {
      console.log(`${vertex} -> ${neighbors.join(', ')}`);
    });
  }

  // --- Adicionando o método BFS à classe Graph ---

  /**
   * Realiza uma Busca em Largura (BFS) a partir de um vértice
inicial.
   * Retorna um mapa com as distâncias (em número de arestas) do
vértice inicial
   * para todos os outros vértices alcançáveis e o caminho.
   * @param startVertex O vértice inicial da busca.
   * @returns Um objeto contendo 'distances' (distâncias) e
'previous' (mapa para reconstruir caminhos).
   */
  bfs(startVertex: T): { distances: Map<T, number>; previous:
Map<T, T | null> } {
    if (!this.adjacencyList.has(startVertex)) {
      throw new Error("Vértice inicial não encontrado no
grafo.");
    }
```

```
    const distances = new Map<T, number>();
    const previous = new Map<T, T | null>();
    const queue: T[] = [];
    const visited = new Set<T>();

    // Inicialização
    this.getVertices().forEach(vertex => {
      distances.set(vertex, Infinity);
      previous.set(vertex, null);
    });

    distances.set(startVertex, 0);
    queue.push(startVertex);
    visited.add(startVertex);

    while (queue.length > 0) {
      const currentVertex =
queue.shift()!; // Remove o primeiro elemento da fila

      this.getNeighbors(currentVertex).forEach(neighbor => {
        if (!visited.has(neighbor)) {
          visited.add(neighbor);
          distances.set(neighbor, distances.get(currentVertex)!
+ 1);
          previous.set(neighbor, currentVertex);
          queue.push(neighbor);
        }
      });
    }

    return { distances, previous };
  }

  /**
   * Reconstrói o caminho mais curto de startVertex até
endVertex usando o resultado da BFS.
   * @param previous O mapa 'previous' retornado pela BFS.
   * @param startVertex O vértice inicial.
   * @param endVertex O vértice final.
   * @returns Um array representando o caminho, ou null se não
houver caminho.
   */
  reconstructPath(previous: Map<T, T | null>, startVertex: T,
endVertex: T): T[] | null {
      const path: T[] = [];
      let current: T | null = endVertex;

      // Volta do final para o início usando o mapa 'previous'
      while (current !== null && current !== startVertex) {
          path.unshift(current);
          current = previous.get(current) ?? null;
      }
```

```
      // Se chegamos ao início, adiciona-o e retorna o caminho
      if (current === startVertex) {
          path.unshift(startVertex);
          return path;
      }

      // Se current se tornou null antes de encontrar
startVertex, não há caminho
      return null;
  }
}

// Exemplo de uso:
const myGraph = new Graph<string>();
myGraph.addVertex("A");
myGraph.addVertex("B");
myGraph.addVertex("C");
myGraph.addVertex("D");
myGraph.addVertex("E");

myGraph.addEdge("A", "B");
myGraph.addEdge("A", "C");
myGraph.addEdge("B", "D");
myGraph.addEdge("C", "E");
myGraph.addEdge("D", "E");
myGraph.addEdge("D", "A"); // Adicionando outra conexão

console.log("--- Exemplo Grafo Básico ---");
myGraph.printGraph();
console.log("Vizinhos de D:", myGraph.getNeighbors("D"));
console.log("Todos os vértices:", myGraph.getVertices());


// Exemplo de uso do BFS:
const myGraphBfs = new Graph<string>();
myGraphBfs.addVertex("A");
myGraphBfs.addVertex("B");
myGraphBfs.addVertex("C");
myGraphBfs.addVertex("D");
myGraphBfs.addVertex("E");
myGraphBfs.addVertex("F"); // Novo vértice

myGraphBfs.addEdge("A", "B");
myGraphBfs.addEdge("A", "C");
myGraphBfs.addEdge("B", "D");
myGraphBfs.addEdge("C", "E");
myGraphBfs.addEdge("D", "E");
myGraphBfs.addEdge("D", "F");
myGraphBfs.addEdge("E", "F");

console.log("\n--- Exemplo BFS ---");
```

```
myGraphBfs.printGraph();

const startNode = "A";
const endNode = "F";
const bfsResult = myGraphBfs.bfs(startNode);

console.log(`\nDistâncias a partir de ${startNode}:`);
bfsResult.distances.forEach((dist, vertex) => {
  console.log(`${vertex}: ${dist === Infinity ?
'Inalcançável' : dist}`);
});

console.log(`\nCaminho mais curto de ${startNode} para $
{endNode}:`);
const path = myGraphBfs.reconstructPath(bfsResult.previous,
startNode, endNode);

if (path) {
  console.log(path.join(' -> '));
} else {
  console.log(`Não há caminho de ${startNode} para $
{endNode}.`);
}
```

Este código define uma classe `Graph` genérica que utiliza um `Map` para armazenar a lista de adjacências. Ele permite adicionar vértices e arestas não direcionadas, obter vizinhos e listar todos os vértices. O exemplo também inclui a implementação e uso do algoritmo BFS dentro da classe.

Estes são exemplos básicos para ilustrar a implementação. Grafos ponderados exigiriam armazenar o peso junto com o vizinho na lista de adjacências, e algoritmos como Dijkstra precisariam de estruturas adicionais (como uma fila de prioridade) para funcionar eficientemente.

Referências Consolidadas

Origem e História (O):

- [O1] Matemateca IME-USP. (s.d.). AS PONTES DE KÖNIGSBERG. Recuperado de https://matemateca.ime.usp.br/acervo/pontes_konigsberg.html
- [O2] Wikipédia. (s.d.). Sete pontes de Königsberg. Recuperado de https://pt.wikipedia.org/wiki/Sete_pontes_de_K%C3%B6nigsberg
- [O3] Euler, L. (1736). Solutio problematis ad geometriam situs pertinentis. Commentarii academiae scientiarum Petropolitanae 8, pp. 128-140. (Referência citada em [O2])

Problemas Clássicos (P):

- [P1] Dias, L. A. (2019, 18 de novembro). Grafos, teoria e aplicações. Medium. Recuperado de https://medium.com/xp-inc/grafos-teoria-e-aplica%C3%A7%C3%B5es-2a87444df855
- [P2] Cáceres, E. N. (s.d.). Algoritmos Paralelos para Problemas em Grafos [Tese de Doutorado, Universidade Federal do Rio de Janeiro]. Recuperado de https://www.cos.ufrj.br/uploadfile/1339608520.pdf (Consultado em 26 de maio de 2025)
- [P3] Ibilce-Unesp. (s.d.). Decomposição - Teoria dos Grafos. Recuperado de https://www.ibilce.unesp.br/Home/Departamentos/MatematicaAplicada/docentes/socorro/decomposicoesarestas.pdf (Referência de resultado de busca)

Algoritmos Fundamentais (A):

- [A1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- [A2] Navone, E. C. (2022, 27 de março). Algoritmo de caminho de custo mínimo de Dijkstra - uma introdução detalhada e visual. freeCodeCamp. Recuperado de https://www.freecodecamp.org/portuguese/news/algoritmo-de-caminho-de-custo-minimo-de-dijkstra-uma-introducao-detalhada-e-visual/
- [A3] Paulo Feofiloff, IME-USP. (s.d.). Busca em largura (BFS) num grafo. Recuperado de https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html
- [A4] Khan Academy. (s.d.). Busca em largura e seus usos. Recuperado de https://pt.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/breadth-first-search-and-its-uses
- [A5] freeCodeCamp. (2024, 11 de abril). Usando a busca em largura para encontrar os caminhos mínimos. Recuperado de https://www.freecodecamp.org/portuguese/news/usando-a-busca-em-largura-para-encontrar-os-caminhos-minimos/
- [A6] Paulo Feofiloff, IME-USP. (s.d.). Busca em profundidade (DFS) num grafo. Recuperado de https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html
- [A7] Wikipédia. (s.d.). Busca em profundidade. Recuperado de https://pt.wikipedia.org/wiki/Busca_em_profundidade

Aplicações Reais (AP):

- [AP1] Dias, L. A. (2019, 18 de novembro). Grafos, teoria e aplicações. Medium. Recuperado de https://medium.com/xp-inc/grafos-teoria-e-aplica%C3%A7%C3%B5es-2a87444df855

- [AP2] Reddit User Discussion. (2021, 20 de fevereiro). Graph Theory Applications. r/math. Recuperado de https://www.reddit.com/r/math/comments/lob7nk/graph_theory_applications/
- [AP3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- [AP4] Maquengo, G. L. (s.d.). Teoria dos Grafos e Aplicações: Redes Eléctricas e de Transportes [Dissertação de Mestrado, Universidade de Évora]. Recuperado de https://dspace.uevora.pt/rdpc/bitstream/10174/25738/1/Mestrado-Matem%C3%A1tica_e_Aplica%C3%A7%C3%B5es-Gabriel_Lima_Maquengo-Teoria_dos_grafos_e_aplica%C3%A7%C3%B5es....pdf (Consultado em 26 de maio de 2025)

Implementação (I):

- [I1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press. (Referência geral para estruturas de dados e algoritmos de grafos)
- [I2] Documentação oficial do TypeScript: https://www.typescriptlang.org/docs/