Otimização de Rotas de Entrega com Teoria dos Grafos

Nota ao leitor

Este material foi escrito em linguagem acessível, pensado para alunos de graduação que estão começando a estudar grafos e algoritmos de rotas. As citações fornecem base teórica e caminhos para estudo, mas não implicam domínio completo do tema por parte do autor do trabalho; elas servem para orientar e permitir verificação independente do conteúdo.

Ao ler, avance passo a passo: entenda primeiro os conceitos e exemplos básicos (condicionais, laços, listas, mapas e matrizes) e, em seguida, veja como eles se combinam para formar representações de grafos e o algoritmo de Dijkstra. Se algo parecer denso, retorne às seções anteriores e aos exemplos do anexo — a compreensão virá pela prática incremental.

Índice

1) Conceitos básicos: o que é um grafo

Na teoria dos grafos, um grafo é uma estrutura composta por vértices (também chamados de nós) e arestas (ligações entre os vértices). Vértices representam entidades e arestas representam relações ou conexões entre essas entidades. Arestas podem ser direcionadas (com sentido) ou não direcionadas, e podem ter peso, que é um valor associado à aresta, como distância, tempo, custo, consumo de combustível, entre outros. Um grafo ponderado é aquele em que as arestas possuem pesos; um grafo simples não tem laços (aresta que liga o vértice a si mesmo) nem múltiplas arestas entre o mesmo par de vértices.

2) Como representar uma cidade como grafo

Uma cidade pode ser modelada como um grafo ponderado e geralmente direcionado: cada interseção (cruzamento) vira um vértice e cada segmento de rua vira uma aresta. O peso da aresta pode ser a distância do trecho, o tempo médio de percurso (considerando limites de velocidade e semáforos), ou um custo composto que combine múltiplos fatores (tempo, pedágios, restrições de altura/peso, etc.). Ruas de mão dupla são representadas por duas arestas, uma em cada sentido; ruas de mão única são representadas por uma única aresta no sentido permitido. Garagens, centros de

distribuição e pontos de entrega também podem ser vértices especiais com atributos como janelas de horário e prioridades.

3) Encontrar a rota mais curta: algoritmo de Dijkstra (visão prática)

Para determinar a menor distância ou o menor tempo entre dois pontos, o algoritmo de Dijkstra é um dos mais usados em grafos com pesos não negativos.

- Ideia central: partindo do vértice de origem, o algoritmo mantém uma estimativa de menor custo até cada vértice e, repetidamente, “fixa” o vértice ainda não resolvido com menor custo atual, relaxando as arestas que dele saem.
- Entradas: grafo ponderado (pesos ≥ 0), vértice de origem e, opcionalmente, um destino.
- Saídas: menor custo da origem a todos os vértices (ou até o destino) e o caminho correspondente (por meio dos predecessores).

Passos resumidos: 1. Inicialize a distância da origem como 0 e de todos os demais vértices como infinito; predecessores nulos. 2. Use uma fila de prioridade (min-heap) para sempre extrair o vértice com menor distância estimada. 3. Para cada aresta que sai do vértice extraído, tente melhorar (relaxar) a distância do vizinho: se `dist[u] + peso(u,v) < dist[v]`, atualize `dist[v]` e registre `u` como predecessor de `v`. 4. Repita até resolver o destino ou esvaziar a fila. 5. Reconstrua o caminho seguindo os predecessores, do destino até a origem.

Observações práticas: - O “melhor” peso nem sempre é a menor distância; muitas vezes o objetivo real é o menor tempo ou o menor custo operacional. Ajuste o peso para refletir a métrica desejada. - Para múltiplas entregas e múltiplos veículos, o problema evolui para variantes do Problema de Roteamento de Veículos (VRP), que exigem técnicas adicionais além de um simples caminho mínimo.

4) Desafios práticos em entregas reais

- Dinâmica do tráfego: o peso das arestas (tempo) muda ao longo do dia por congestionamentos, incidentes e obras. É útil atualizar pesos com dados em tempo real.
- Restrições e regras de trânsito: ruas com sentido único, horários de restrição para caminhões, ruas fechadas temporariamente e zonas de pedestre exigem que o grafo seja fiel às regras locais.
- Janelas de tempo e prioridades: clientes podem exigir entregas em intervalos específicos; incorporar penalidades por atraso muda a função de custo.
- Múltiplos veículos e capacidade: quando há frota e limites de carga/volume, o problema vira VRP; heurísticas e metaheurísticas

(Savings, Clarke-Wright, 2-opt/3-opt, Tabu Search, Simulated Annealing) são comuns.
- Dados imperfeitos: mapas desatualizados, endereços imprecisos e geocodificação ruidosa demandam validação e correção de dados.

5) Benefícios para empresas de logística

- Redução de custos: menor consumo de combustível, menos horas de motorista e desgaste de frota.
- Aumento de pontualidade: roteiros mais curtos e aderentes a janelas de entrega elevam o nível de serviço.
- Escalabilidade operacional: automatizar a geração de rotas permite atender mais pedidos sem aumentar proporcionalmente a equipe.
- Simulação e planejamento: é possível testar cenários (novos hubs, mudanças de horários, expansão de área) antes de executá-los.
- Tomada de decisão baseada em dados: métricas claras (tempo médio por entrega, custo por rota, ocupação de veículos) viabilizam melhoria contínua.

Referências e fontes de pesquisa

- Cormen, Leiserson, Rivest, Stein. “Introduction to Algorithms (CLRS)”. MIT Press.
- Sedgewick, Wayne. “Algorithms” (4th Ed.). Addison-Wesley.
- Dijkstra, E. W. (1959). “A note on two problems in connexion with graphs”. Numerische Mathematik.
- Documentação do NetworkX: `https://networkx.org/documentation/stable/`
- OR-Tools (Google) para rotas e VRP: `https://developers.google.com/optimization`
- Bastos, R. et al. Materiais de Grafos e Otimização em cursos de Ciência da Computação (apostilas e slides universitários).

Anexo: Exemplos básicos em Java (do básico até grafos)

Os exemplos abaixo mostram, de forma incremental e didática, estruturas fundamentais da linguagem Java (condicionais, laços, listas, matrizes, mapas) até a construção de um grafo simples e um grafo ponderado, incluindo uma implementação direta do algoritmo de Dijkstra. A ideia é: dominando cada peça isoladamente, você entende naturalmente Dijkstra, pois ele é “só” uma composição de decisões (ifs), repetições controladas (laços), estruturas de dados adequadas (listas, mapas, filas/heap) e uma boa representação do problema (grafo como lista de adjacência ou matriz de custos).

Como cada tópico se conecta ao Dijkstra: - If/else: decidir quando atualizar uma distância (relaxamento) é uma verificação condicional. - Laços: o algoritmo percorre vértices e arestas várias vezes de forma sistemática. -

List e Map: listas guardam vizinhos; mapas guardam distâncias, predecessores e estruturas de adjacência. - Matrizes: alternativa de representação; útil para grafos densos ou para visualizar custos. - Fila de prioridade (heap): seleciona sempre o próximo vértice com menor distância estimada (núcleo da eficiência do Dijkstra).

```java
// Exemplo 1: if/else e operador ternário
public class ExemploIfElse {
    public static void main(String[] args) {
        int idade = 20;

        // Condicional simples
        if (idade >= 18) {
            System.out.println("Maior de idade");
        } else {
            System.out.println("Menor de idade");
        }

        // Ternário para decisão curta
        String categoria = (idade >= 18) ? "Adulto" : "Menor";
        System.out.println("Categoria: " + categoria);
    }
}

// Exemplo 2: laços (for, while, for-each)
import java.util.Arrays;

public class ExemploLacos {
    public static void main(String[] args) {
        // for tradicional
        for (int i = 0; i < 3; i++) {
            System.out.println("i = " + i);
        }

        // while
        int j = 0;
        while (j < 3) {
            System.out.println("j = " + j);
            j++;
        }

        // for-each em array
        int[] numeros = {1, 2, 3};
        for (int n : numeros) {
            System.out.println("n = " + n);
        }

        // for-each em lista
        java.util.List<String> nomes = Arrays.asList("Ana", "Bruno",
"Carlos");
```

```java
        for (String nome : nomes) {
            System.out.println("Olá, " + nome);
        }
    }
}

// Exemplo 3: List (ArrayList)
import java.util.ArrayList;
import java.util.List;

public class ExemploLista {
    public static void main(String[] args) {
        List<String> frutas = new ArrayList<>();
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Laranja");

        System.out.println("Tamanho: " + frutas.size());
        System.out.println("Contém Banana? " +
frutas.contains("Banana"));

        // Removendo e iterando
        frutas.remove("Banana");
        for (String f : frutas) {
            System.out.println(f);
        }
    }
}

// Exemplo 4: Matrizes (arrays 2D)
public class ExemploMatriz {
    public static void main(String[] args) {
        // Matriz 3x3
        int[][] matriz = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Acesso por linha e coluna
        System.out.println("Elemento (1,2): " + matriz[1][2]); // 6

        // Percorrendo toda a matriz
        for (int linha = 0; linha < matriz.length; linha++) {
            for (int col = 0; col < matriz[linha].length; col++) {
                System.out.print(matriz[linha][col] + " ");
            }
            System.out.println();
        }
```

```
    }
}

// Exemplo 5: Map (HashMap)
import java.util.HashMap;
import java.util.Map;

public class ExemploMap {
    public static void main(String[] args) {
        Map<String, Integer> estoque = new HashMap<>();
        estoque.put("Caneta", 100);
        estoque.put("Caderno", 50);
        estoque.put("Borracha", 20);

        // Leitura
        System.out.println("Caderno em estoque: " +
estoque.get("Caderno"));

        // Atualização
        estoque.put("Caderno", estoque.get("Caderno") + 10);

        // Iteração sobre entradas
        for (Map.Entry<String, Integer> e : estoque.entrySet()) {
            System.out.println(e.getKey() + ": " + e.getValue());
        }
    }
}

// Exemplo 6: Grafo simples (lista de adjacência não ponderada)
// Representa um grafo direcionado onde cada vértice aponta para seus
vizinhos.
import java.util.*;

class GrafoSimples {
    private final Map<String, List<String>> adjacencia = new
HashMap<>();

    // Adiciona um vértice ao grafo
    public void adicionarVertice(String v) {
        adjacencia.putIfAbsent(v, new ArrayList<>());
    }

    // Adiciona uma aresta direcionada v -> w
    public void adicionarAresta(String v, String w) {
        adicionarVertice(v);
        adicionarVertice(w);
        adjacencia.get(v).add(w);
    }

    // Retorna vizinhos de v
```

```
    public List<String> vizinhos(String v) {
        return adjacencia.getOrDefault(v, Collections.emptyList());
    }

    // Busca em largura (BFS) para encontrar o menor número de arestas
de origem até destino
    public List<String> bfsCaminho(String origem, String destino) {
        Map<String, String> predecessor = new HashMap<>();
        Queue<String> fila = new ArrayDeque<>();
        Set<String> visitado = new HashSet<>();

        fila.add(origem);
        visitado.add(origem);

        while (!fila.isEmpty()) {
            String atual = fila.remove();
            if (atual.equals(destino)) break;
            for (String viz : vizinhos(atual)) {
                if (!visitado.contains(viz)) {
                    visitado.add(viz);
                    predecessor.put(viz, atual);
                    fila.add(viz);
                }
            }
        }

        // Reconstrução do caminho
        if (!origem.equals(destino) && !
predecessor.containsKey(destino)) {
            return Collections.emptyList(); // sem caminho
        }
        List<String> caminho = new ArrayList<>();
        String passo = destino;
        while (passo != null) {
            caminho.add(passo);
            passo = predecessor.get(passo);
            if (passo != null && passo.equals(origem)) {
                caminho.add(passo);
                break;
            }
        }
        Collections.reverse(caminho);
        return caminho;
    }

    public static void main(String[] args) {
        GrafoSimples g = new GrafoSimples();
        g.adicionarAresta("A", "B");
        g.adicionarAresta("A", "C");
        g.adicionarAresta("B", "D");
```

```
        g.adicionarAresta("C", "D");

        System.out.println("Vizinhos de A: " + g.vizinhos("A"));
        System.out.println("Caminho A -> D (BFS): " +
g.bfsCaminho("A", "D"));
    }
}

// Exemplo 7: Grafo ponderado + Dijkstra (menor caminho por peso)
import java.util.*;

class ArestaPonderada {
    final String destino;
    final double peso;
    ArestaPonderada(String destino, double peso) {
        this.destino = destino;
        this.peso = peso;
    }
}

class GrafoPonderado {
    private final Map<String, List<ArestaPonderada>> adj = new
HashMap<>();

    public void adicionarVertice(String v) {
        adj.putIfAbsent(v, new ArrayList<>());
    }

    public void adicionarAresta(String v, String w, double peso) {
        adicionarVertice(v);
        adicionarVertice(w);
        adj.get(v).add(new ArestaPonderada(w, peso));
    }

    public Map<String, Double> dijkstra(String origem) {
        // distâncias iniciadas com infinito
        Map<String, Double> dist = new HashMap<>();
        Map<String, String> prev = new HashMap<>();
        for (String v : adj.keySet()) {
            dist.put(v, Double.POSITIVE_INFINITY);
        }
        dist.put(origem, 0.0);

        // fila de prioridade por menor distância
        PriorityQueue<String> pq = new
PriorityQueue<>(Comparator.comparingDouble(dist::get));
        pq.add(origem);

        while (!pq.isEmpty()) {
            String u = pq.poll();
```

```
            for (ArestaPonderada e : adj.getOrDefault(u,
Collections.emptyList())) {
                double alt = dist.get(u) + e.peso;
                if (alt < dist.getOrDefault(e.destino,
Double.POSITIVE_INFINITY)) {
                    dist.put(e.destino, alt);
                    prev.put(e.destino, u);
                    // Atualiza a prioridade: remove e re-adiciona
                    pq.remove(e.destino);
                    pq.add(e.destino);
                }
            }
        }
        return dist; // pode-se expor também 'prev' para reconstrução
de caminhos
    }

    public List<String> reconstruirCaminho(String origem, String
destino, Map<String, String> prev) {
        List<String> caminho = new ArrayList<>();
        if (!origem.equals(destino) && !prev.containsKey(destino))
return caminho;
        String v = destino;
        while (v != null) {
            caminho.add(v);
            v = prev.get(v);
            if (v != null && v.equals(origem)) {
                caminho.add(v);
                break;
            }
        }
        Collections.reverse(caminho);
        return caminho;
    }

    public static void main(String[] args) {
        GrafoPonderado g = new GrafoPonderado();
        // Grafo com pesos representando, por exemplo, tempo de
deslocamento
        g.adicionarAresta("A", "B", 4);
        g.adicionarAresta("A", "C", 2);
        g.adicionarAresta("C", "B", 1);
        g.adicionarAresta("B", "D", 5);
        g.adicionarAresta("C", "D", 8);

        Map<String, Double> dist = g.dijkstra("A");
        System.out.println("Distâncias a partir de A: " + dist);
    }
}
```