Exercicios resolvidos analise de algoritmos via GPT

05_exercicios_02_analise_de_algoritmos_teoria_dos_grafos.pdf

Implemente uma classe Java chamada `Grafo` que realiza uma **busca em profundidade (DFS)** a partir do vértice 0. O grafo é **não direcionado** e deve ser representado por **lista de adjacências** usando um vetor de `LinkedList<Integer>`. A entrada será fornecida **via linha de comando (args)**, da seguinte forma:

- O primeiro valor representa o número de vértices do grafo.
- Os valores seguintes devem ser interpretados como pares de vértices conectados por uma aresta.

Exemplo de execução:

```
//java Grafo 6 0 1 0 2 1 3 2 4 4 5
```

Explicação da entrada:

- `6` → número de vértices (vértices de 0 a 5)
- `0 1, 0 2, 1 3, 2 4, 4 5` → arestas entre os vértices

O programa deve realizar a DFS a partir do vértice 0 e imprimir a ordem dos vértices visitados.
Saída esperada:

```
DFS a partir do vértice 0:
0 1 3 2 4 5
```

Dica: utilize um vetor `boolean[] visitado` para controlar quais vértices já foram visitados durante a execução do algoritmo.

```java
import java.util.*;

public class Grafo {

  private LinkedList<Integer>[] adjacencias; // Lista de adjacência
  private boolean[] visitado;               // Vetor de controle de visita

  // Construtor que recebe o número de vértices
  public Grafo(int numVertices) {
    // Inicializa a estrutura do grafo
    adjacencias = new LinkedList[numVertices];
    for (int i = 0; i < numVertices; i++) {
      adjacencias[i] = new LinkedList<>();
    }
    visitado = new boolean[numVertices]; // Todos começam como não visitados
  }

  // Adiciona uma aresta não direcionada entre dois vértices
  public void adicionarAresta(int u, int v) {
    adjacencias[u].add(v); // u → v
    adjacencias[v].add(u); // v → u (porque o grafo é não direcionado)
  }
```

```java
  // Função que inicia a DFS a partir do vértice 0
  public void dfs(int origem) {
    System.out.println("DFS a partir do vértice " + origem + ":");
    dfsRecursivo(origem); // Chamada inicial
    System.out.println(); // Pula linha no final da saída
  }

  // Função recursiva de DFS
  private void dfsRecursivo(int v) {
    visitado[v] = true;          // Marca vértice como visitado
    System.out.print(v);         // Imprime o vértice visitado
    for (int vizinho : adjacencias[v]) {
      if (!visitado[vizinho]) {
        dfsRecursivo(vizinho);   // Chamada recursiva para os vizinhos não visitados
      }
    }
  }

  // Função principal que processa a entrada da linha de comando
  public static void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Uso: java Grafo <numVertices> <arestas...>");
      return;
    }

    int numVertices = Integer.parseInt(args[0]);
    Grafo grafo = new Grafo(numVertices);

    // Adiciona arestas: cada par de valores representa uma aresta
    for (int i = 1; i < args.length - 1; i += 2) {
      int u = Integer.parseInt(args[i]);
      int v = Integer.parseInt(args[i + 1]);
      grafo.adicionarAresta(u, v);
    }

    // Inicia DFS a partir do vértice 0
    grafo.dfs(0);
  }
}
```

Implemente uma classe Java chamada `Grafo` que realiza uma **busca em largura (BFS)** para encontrar a **distância mínima** (menor número de arestas) entre dois vértices. A entrada será fornecida **via linha de comando (args)**, da seguinte forma:

- O primeiro valor representa o número de vértices do grafo.
- Os valores seguintes devem ser interpretados como pares de vértices conectados por uma aresta.
- Os dois últimos valores indicam o vértice de origem e o vértice de destino.

Exemplo de execução:

```
//java Grafo 6 0 1 0 3 1 5 2 5 3 4 4 5 0 5
```

Explicação da entrada:

- `6` → número de vértices (de 0 a 5)
- `0 1, 0 3, 1 5, 2 5, 3 4, 4 5` → arestas
- `0 5` → origem = 0, destino = 5

Nesse grafo, existem dois caminhos possíveis de 0 até 5:

- `0 → 1 → 5` (2 arestas)
- `0 → 3 → 4 → 5` (3 arestas)

O algoritmo deve encontrar o caminho mais curto (com menos arestas).

Saída esperada:

```
Distância mínima de 0 até 5 é 2
```

```java
import java.util.*;

public class Grafo {

  private LinkedList<Integer>[] adjacencias; // Lista de adjacência

  // Construtor que recebe o número de vértices
  public Grafo(int numVertices) {
    adjacencias = new LinkedList[numVertices];
    for (int i = 0; i < numVertices; i++) {
      adjacencias[i] = new LinkedList<>();
    }
  }

  // Adiciona uma aresta não direcionada entre u e v
  public void adicionarAresta(int u, int v) {
    adjacencias[u].add(v);
    adjacencias[v].add(u);
  }

  // Realiza busca em largura (BFS) e retorna a menor distância de origem até destino
  public int bfs(int origem, int destino) {
    boolean[] visitado = new boolean[adjacencias.length]; // Marca os vértices visitados
    int[] distancia = new int[adjacencias.length];        // Guarda a distância da origem
```

```
    Queue<Integer> fila = new LinkedList<>();
    fila.offer(origem);
    visitado[origem] = true;
    distancia[origem] = 0;

    while (!fila.isEmpty()) {
      int atual = fila.poll();

      // Se chegou ao destino, pode encerrar
      if (atual == destino) {
        return distancia[atual];
      }

      // Visita os vizinhos não visitados
      for (int vizinho : adjacencias[atual]) {
        if (!visitado[vizinho]) {
          visitado[vizinho] = true;
          distancia[vizinho] = distancia[atual] + 1;
          fila.offer(vizinho);
        }
      }
    }

    // Se não encontrou caminho até o destino
    return -1;
  }

  // Função principal: processa os argumentos da linha de comando
  public static void main(String[] args) {
    if (args.length < 3) {
      System.out.println("Uso: java Grafo <numVertices> <arestas...> <origem> <destino>");
      return;
    }

    int numVertices = Integer.parseInt(args[0]);
    Grafo grafo = new Grafo(numVertices);

    // Arestas: pares de vértices até os dois últimos argumentos
    for (int i = 1; i < args.length - 2; i += 2) {
      int u = Integer.parseInt(args[i]);
      int v = Integer.parseInt(args[i + 1]);
      grafo.adicionarAresta(u, v);
    }

    // Últimos dois argumentos: origem e destino
    int origem = Integer.parseInt(args[args.length - 2]);
    int destino = Integer.parseInt(args[args.length - 1]);

    int distancia = grafo.bfs(origem, destino);

    if (distancia >= 0) {
      System.out.printf("Distância mínima de %d até %d é %d\n", origem, destino, distancia);
```

```
    } else {
      System.out.printf("Não existe caminho de %d até %d.\n", origem, destino);
    }
  }
}
```

06_exercicios_analise_de_algoritmos_teoria_dos_grafos_dijkstra.pdf

Visite o repositório a seguir para obter uma cópia da solução em que você vai basear as soluções dos exercícios propostos.
https://github.com/professorbossini/20251_maua_cic401

1 Adapte o programa fazendo com que cada caminho mínimo seja exibido no seguinte Formato:

a ->(p1) b ->(p2) -> c

Neste exemplo, a, b e c são vértices. p1 e p2 são os pesos das arestas que ligam o vértice a ao vértice b e o vértice b ao vértice c, respectivamente.

2 O programa mostra os menores caminhos a partir de um único vértice escolhido como origem. Adapte-o para que ele calcule e mostre os menores caminhos a partir de cada vértice do grafo

```
package menores_caminhos_de_origem_unica;
import java.util.*;

class Grafo {

  private final Vertice[] vertices;
  private List<List<Aresta>> adjacencias;

  Grafo(String[] nomesVertices) {
    this.adjacencias = new ArrayList<>();
    this.vertices = new Vertice[nomesVertices.length];
    for (int i = 0; i < nomesVertices.length; i++) {
      this.adjacencias.add(new ArrayList<>());
      this.vertices[i] = new Vertice(nomesVertices[i], i);
    }
  }

  public void adicionarAresta(int origem, int destino, int peso) {
    adjacencias.get(origem).add(new Aresta(destino, peso));
  }
```

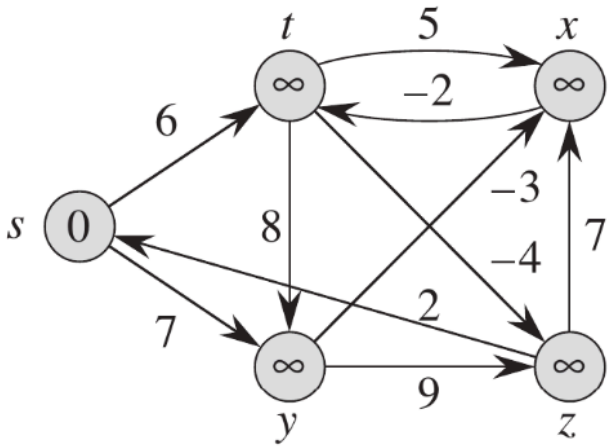
```java
  public List<Aresta> vizinhos(int u) {
    return adjacencias.get(u);
  }

  public Vertice[] getVertices() {
    return vertices;
  }

  public int quantidadeVertices() {
    return vertices.length;
  }

  // Método auxiliar para obter o peso da aresta entre dois vértices
  public int pesoAresta(int u, int v) {
    for (Aresta a : adjacencias.get(u)) {
      if (a.destino == v) return a.peso;
    }
    return -1; // Aresta inexistente
  }

  static class Vertice {
    String nome;
    int indice;
    int distancia;
    Vertice predecessor;

    Vertice(String nome, int indice) {
      this.nome = nome;
      this.indice = indice;
      this.distancia = Integer.MAX_VALUE;
      this.predecessor = null;
    }
  }

  static class Aresta {
    int destino;
    int peso;

    Aresta(int destino, int peso) {
      this.destino = destino;
      this.peso = peso;
    }
  }
}

public class Dijkstra {

  public void executar(Grafo g, int s) {
    inicializarFonteUnica(g, s);
    List<Grafo.Vertice> q = new ArrayList<>(List.of(g.getVertices()));
    while (!q.isEmpty()) {
      var u = extrairMinimo(q);
```

```
    for (Grafo.Aresta aresta : g.vizinhos(u.indice)) {
      relaxar(u, g.getVertices()[aresta.destino], aresta.peso);
    }
  }
}

private void relaxar(Grafo.Vertice u, Grafo.Vertice v, int w) {
  if (u.distancia + w < v.distancia) {
    v.distancia = u.distancia + w;
    v.predecessor = u;
  }
}

private void inicializarFonteUnica(Grafo g, int s) {
  for (Grafo.Vertice v : g.getVertices()) {
    v.distancia = Integer.MAX_VALUE;
    v.predecessor = null;
  }
  g.getVertices()[s].distancia = 0;
}

private Grafo.Vertice extrairMinimo(List<Grafo.Vertice> q) {
  int indiceMin = 0;
  for (int i = 1; i < q.size(); i++) {
    if (q.get(i).distancia < q.get(indiceMin).distancia) {
      indiceMin = i;
    }
  }
  return q.remove(indiceMin);
}

// Reconstrói e imprime o caminho no formato: a ->(p1) b ->(p2) c
private void imprimirCaminho(Grafo g, Grafo.Vertice destino) {
  if (destino.predecessor == null) {
    System.out.print(destino.nome);
  } else {
    imprimirCaminho(g, destino.predecessor); // Caminho recursivo
    int peso = g.pesoAresta(destino.predecessor.indice, destino.indice);
    System.out.print(" ->(" + peso + ") " + destino.nome);
  }
}

public static void main(String[] args) {
  String[] nomes = {"s", "t", "x", "y", "z"};
  Grafo g = new Grafo(nomes);

  // Define o grafo com arestas direcionadas e pesos
  g.adicionarAresta(0, 1, 10);
  g.adicionarAresta(0, 3, 5);
  g.adicionarAresta(1, 2, 1);
  g.adicionarAresta(1, 3, 2);
  g.adicionarAresta(2, 4, 4);
```

```
    g.adicionarAresta(3, 1, 3);
    g.adicionarAresta(3, 2, 9);
    g.adicionarAresta(3, 4, 2);
    g.adicionarAresta(4, 2, 6);

    // Calcula e mostra os caminhos a partir de CADA vértice como origem
    var dijkstra = new Dijkstra();

    for (int i = 0; i < g.quantidadeVertices(); i++) {
      System.out.println("\n==============================");
      System.out.println("Origem: " + g.getVertices()[i].nome);
      dijkstra.executar(g, i); // Executa Dijkstra com vértice i como origem

      for (Grafo.Vertice v : g.getVertices()) {
        System.out.print("Caminho até " + v.nome + ": ");
        dijkstra.imprimirCaminho(g, v);
        System.out.println(" | Distância: " + v.distancia);
      }
    }
  }
}
```

07_exercicios_analise_de_algoritmos_teoria_dos_grafos_bellmanford.pdf

1 Execute o algoritmo de Bellman Ford para o seguinte grafo.



2 Adapte o algoritmo para que, caso um ciclo de peso negativo a partir da fonte seja encontrado, ele seja exibido.

3 Adapte o algoritmo para que ele exiba os menores caminhos tendo como fonte cada um dos vértices do grafo.

```java
package menores_caminhos_de_origem_unica;
import java.util.*;

class Grafo {
  private final Vertice[] vertices;
  private List<List<Aresta>> adjacencias;

  Grafo(String[] nomesVertices) {
    this.adjacencias = new ArrayList<>();
    this.vertices = new Vertice[nomesVertices.length];
    for (int i = 0; i < nomesVertices.length; i++) {
      this.adjacencias.add(new ArrayList<>());
      this.vertices[i] = new Vertice(nomesVertices[i], i);
    }
  }

  public void adicionarAresta(int origem, int destino, int peso) {
    adjacencias.get(origem).add(new Aresta(destino, peso));
  }

  public List<Aresta> vizinhos(int u) {
    return adjacencias.get(u);
  }

  public int pesoAresta(int u, int v) {
    for (Aresta a : adjacencias.get(u)) {
      if (a.destino == v) return a.peso;
    }
    return Integer.MAX_VALUE;
  }

  public Vertice[] getVertices() {
    return vertices;
  }

  public int quantidadeVertices() {
    return vertices.length;
  }

  static class Vertice {
    String nome;
    int indice;
    int distancia;
    Vertice predecessor;

    Vertice(String nome, int indice) {
      this.nome = nome;
      this.indice = indice;
      this.distancia = Integer.MAX_VALUE;
      this.predecessor = null;
    }
  }
```

```
  static class Aresta {
    int destino;
    int peso;

    Aresta(int destino, int peso) {
      this.destino = destino;
      this.peso = peso;
    }
  }
}

public class BellmanFord {

  public boolean executar(Grafo g, int origem) {
    inicializarFonteUnica(g, origem);

    // Relaxa todas as arestas (V - 1) vezes
    for (int i = 0; i < g.quantidadeVertices() - 1; i++) {
      for (var u : g.getVertices()) {
        for (var a : g.vizinhos(u.indice)) {
          var v = g.getVertices()[a.destino];
          relaxar(u, v, a.peso);
        }
      }
    }

    // Verifica ciclos negativos
    for (var u : g.getVertices()) {
      for (var a : g.vizinhos(u.indice)) {
        var v = g.getVertices()[a.destino];
        if (v.distancia > u.distancia + a.peso) {
          System.out.println("⚠️ Ciclo de peso negativo detectado:");
          exibirCicloNegativo(v);
          return false;
        }
      }
    }

    return true;
  }

  private void relaxar(Grafo.Vertice u, Grafo.Vertice v, int w) {
    if (u.distancia != Integer.MAX_VALUE && u.distancia + w < v.distancia) {
      v.distancia = u.distancia + w;
      v.predecessor = u;
    }
  }

  private void inicializarFonteUnica(Grafo g, int s) {
    for (Grafo.Vertice v : g.getVertices()) {
      v.distancia = Integer.MAX_VALUE;
```

```
      v.predecessor = null;
    }
    g.getVertices()[s].distancia = 0;
  }

  private void exibirCaminho(Grafo g, Grafo.Vertice v) {
    if (v.predecessor == null) {
      System.out.print(v.nome);
    } else {
      exibirCaminho(g, v.predecessor);
      int peso = g.pesoAresta(v.predecessor.indice, v.indice);
      System.out.print(" ->(" + peso + ") " + v.nome);
    }
  }

  private void exibirCicloNegativo(Grafo.Vertice v) {
    Set<Grafo.Vertice> visitados = new HashSet<>();
    while (v != null && !visitados.contains(v)) {
      visitados.add(v);
      v = v.predecessor;
    }

    // Mostra o ciclo
    if (v != null) {
      Grafo.Vertice cicloInicio = v;
      List<String> ciclo = new ArrayList<>();
      do {
        ciclo.add(v.nome);
        v = v.predecessor;
      } while (v != null && v != cicloInicio);

      if (v == cicloInicio) {
        ciclo.add(cicloInicio.nome); // fecha o ciclo
        Collections.reverse(ciclo);
        System.out.println("Ciclo: " + String.join(" -> ", ciclo));
      }
    }
  }

  public static void main(String[] args) {
    String[] nomes = {"s", "t", "x", "y", "z"};
    Grafo g = new Grafo(nomes);

    // Arestas conforme o grafo da imagem
    g.adicionarAresta(0, 1, 6);   // s -> t
    g.adicionarAresta(0, 3, 7);   // s -> y
    g.adicionarAresta(1, 2, 5);   // t -> x
    g.adicionarAresta(1, 3, 8);   // t -> y
    g.adicionarAresta(1, 4, -4);  // t -> z
    g.adicionarAresta(2, 1, -2);  // x -> t
    g.adicionarAresta(3, 2, -3);  // y -> x
    g.adicionarAresta(3, 4, 9);   // y -> z
```

```
    g.adicionarAresta(4, 0, 2);   // z -> s
    g.adicionarAresta(4, 2, 7);   // z -> x

    BellmanFord bf = new BellmanFord();

    // Executa Bellman-Ford a partir de cada vértice
    for (int i = 0; i < g.quantidadeVertices(); i++) {
      System.out.println("\n======================================");
      System.out.println("Origem: " + g.getVertices()[i].nome);
      boolean ok = bf.executar(g, i);

      if (ok) {
        for (Grafo.Vertice v : g.getVertices()) {
          System.out.print("Caminho até " + v.nome + ": ");
          bf.exibirCaminho(g, v);
          System.out.println(" | Distância: " + (v.distancia == Integer.MAX_VALUE ? "∞" :
v.distancia));
        }
      }
    }
  }
}
```

08_exercicios_analise_de_algoritmos_teoria_dos_grafos_kruskal_prim.pdf

Exercícios

Visite o repositório a seguir para obter uma cópia da solução em que você vai basear as soluções dos exercícios propostos.

https://github.com/professorbossini/20251_maua_cic401

1 Aprimore o algoritmo de Kruskal visto em aula, aplicando as técnicas Path Compression e Union By Rank do capítulo 21 do livro do Cormen.

2. Implemente o algoritmo de Prim.

1)

```
import java.util.*;

class Aresta implements Comparable<Aresta> {
  int u, v, peso;
```

```java
  Aresta(int u, int v, int peso) {
    this.u = u;
    this.v = v;
    this.peso = peso;
  }

  @Override
  public int compareTo(Aresta o) {
    return Integer.compare(this.peso, o.peso); // ordena por peso crescente
  }
}

// Estrutura Union-Find aprimorada com path compression e union by rank
class UnionFind {
  int[] pai;     // Representa o pai de cada nó
  int[] rank;    // Altura estimada da árvore

  UnionFind(int n) {
    pai = new int[n];
    rank = new int[n];
    for (int i = 0; i < n; i++) {
      pai[i] = i;    // Cada elemento começa como seu próprio pai
      rank[i] = 0;   // Todos começam com rank 0
    }
  }

  // Find com compressão de caminho
  public int find(int x) {
    if (pai[x] != x) {
      pai[x] = find(pai[x]); // Aponta diretamente para o representante raiz
    }
    return pai[x];
  }

  // Union por rank
  public void union(int x, int y) {
    int raizX = find(x);
    int raizY = find(y);
    if (raizX == raizY) return; // Já estão no mesmo conjunto

    // Une a árvore menor na maior
    if (rank[raizX] < rank[raizY]) {
      pai[raizX] = raizY;
    } else if (rank[raizX] > rank[raizY]) {
      pai[raizY] = raizX;
    } else {
      pai[raizY] = raizX;
      rank[raizX]++; // Aumenta a altura
    }
  }
}
```

```java
public class KruskalAprimorado {

  public List<Aresta> kruskal(int n, List<Aresta> arestas) {
    Collections.sort(arestas); // Ordena arestas por peso crescente
    UnionFind uf = new UnionFind(n); // Inicializa estrutura de conjuntos disjuntos
    List<Aresta> agm = new ArrayList<>(); // Lista da árvore geradora mínima

    for (Aresta a : arestas) {
      if (uf.find(a.u) != uf.find(a.v)) {
        uf.union(a.u, a.v);  // Une os conjuntos
        agm.add(a);          // Adiciona aresta à AGM
      }
    }

    return agm;
  }

  public static void main(String[] args) {
    int n = 5;
    List<Aresta> arestas = Arrays.asList(
      new Aresta(0, 1, 10),
      new Aresta(0, 2, 6),
      new Aresta(0, 3, 5),
      new Aresta(1, 3, 15),
      new Aresta(2, 3, 4)
    );

    var agm = new KruskalAprimorado().kruskal(n, arestas);
    int total = 0;
    for (Aresta a : agm) {
      System.out.printf("(%d, %d, %d) ", a.u, a.v, a.peso);
      total += a.peso;
    }
    System.out.println("\nPeso total da AGM: " + total);
  }
}
```

2)

```java
import java.util.*;

class ArestaPrim {
  int destino, peso;

  ArestaPrim(int destino, int peso) {
    this.destino = destino;
    this.peso = peso;
  }
}

class Par implements Comparable<Par> {
```

```
  int vertice, peso;

  Par(int vertice, int peso) {
    this.vertice = vertice;
    this.peso = peso;
  }

  @Override
  public int compareTo(Par outro) {
    return Integer.compare(this.peso, outro.peso); // menor peso primeiro
  }
}

public class Prim {

  public int prim(int n, List<List<ArestaPrim>> grafo, int origem) {
    boolean[] visitado = new boolean[n]; // Marca os vértices já incluídos na AGM
    PriorityQueue<Par> fila = new PriorityQueue<>(); // Fila de prioridade para escolher
menor peso
    fila.offer(new Par(origem, 0)); // Começa com o vértice de origem
    int custoTotal = 0;

    while (!fila.isEmpty()) {
      Par atual = fila.poll();

      if (visitado[atual.vertice]) continue; // Ignora se já foi incluído

      visitado[atual.vertice] = true; // Marca como visitado
      custoTotal += atual.peso;       // Soma peso da aresta na AGM

      // Para cada vizinho do vértice atual
      for (ArestaPrim vizinho : grafo.get(atual.vertice)) {
        if (!visitado[vizinho.destino]) {
          fila.offer(new Par(vizinho.destino, vizinho.peso)); // Candidata aresta para a AGM
        }
      }
    }

    return custoTotal;
  }

  public static void main(String[] args) {
    int n = 5;
    List<List<ArestaPrim>> grafo = new ArrayList<>();
    for (int i = 0; i < n; i++) grafo.add(new ArrayList<>());

    // Grafo não direcionado
    grafo.get(0).add(new ArestaPrim(1, 10));
    grafo.get(1).add(new ArestaPrim(0, 10));

    grafo.get(0).add(new ArestaPrim(2, 6));
    grafo.get(2).add(new ArestaPrim(0, 6));
```

```
    grafo.get(0).add(new ArestaPrim(3, 5));
    grafo.get(3).add(new ArestaPrim(0, 5));

    grafo.get(1).add(new ArestaPrim(3, 15));
    grafo.get(3).add(new ArestaPrim(1, 15));

    grafo.get(2).add(new ArestaPrim(3, 4));
    grafo.get(3).add(new ArestaPrim(2, 4));

    var prim = new Prim();
    int custo = prim.prim(n, grafo, 0);
    System.out.println("Custo total da AGM (Prim): " + custo);
  }
}
```

09_exercicios_analise_de_algoritmos_heapsort.pdf

1 Os algoritmos a seguir envolvem o uso de uma estrutura de dados Heap para fazer a implementação de uma fila de prioridades, tal como descrito no livro do Cormen. Faça a sua implementação em Java.

HEAP-MAXIMUM(A)
1 **return** $A[1]$

HEAP-EXTRACT-MAX(A)
1 **if** $A.heap\text{-}size < 1$
2 **error** "heap underflow"
3 $max = A[1]$
4 $A[1] = A[A.heap\text{-}size]$
5 $A.heap\text{-}size = A.heap\text{-}size - 1$
6 MAX-HEAPIFY(A, 1)
7 **return** max

HEAP-INCREASE-KEY(A, i, key)
1 **if** $key < A[i]$
2 **error** "new key is smaller than current key"
3 $A[i] = key$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5 exchange $A[i]$ with $A[\text{PARENT}(i)]$
6 $i = \text{PARENT}(i)$

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

```java
import java.util.*;

public class MaxHeapPriorityQueue {

  private List<Integer> heap;

  // Construtor: inicializa o heap com índice 1 como no Cormen
  public MaxHeapPriorityQueue() {
    heap = new ArrayList<>();
    heap.add(null); // posição 0 não é usada (estilo 1-based index)
  }

  // Retorna o índice do pai
  private int parent(int i) {
    return i / 2;
  }

  // Retorna o índice do filho esquerdo
  private int left(int i) {
    return 2 * i;
  }

  // Retorna o índice do filho direito
  private int right(int i) {
    return 2 * i + 1;
  }

  // Troca dois elementos no heap
  private void swap(int i, int j) {
    int temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }

  // Retorna o maior elemento (A[1])
  public int heapMaximum() {
    if (heap.size() <= 1)
      throw new NoSuchElementException("Heap is empty");
    return heap.get(1);
  }

  // Remove e retorna o maior elemento do heap
  public int heapExtractMax() {
    if (heap.size() <= 1)
```

```
    throw new NoSuchElementException("Heap underflow");

  int max = heap.get(1); // maior valor
  int last = heap.remove(heap.size() - 1); // remove o último elemento

  if (heap.size() == 1) return max; // só tinha um elemento

  heap.set(1, last); // move o último para a raiz
  maxHeapify(1);     // restaura a propriedade de max-heap
  return max;
}

// Corrige a subárvore enraizada em i para manter a propriedade de max-heap
private void maxHeapify(int i) {
  int l = left(i);
  int r = right(i);
  int largest = i;

  if (l < heap.size() && heap.get(l) > heap.get(largest))
    largest = l;
  if (r < heap.size() && heap.get(r) > heap.get(largest))
    largest = r;

  if (largest != i) {
    swap(i, largest);
    maxHeapify(largest);
  }
}

// Aumenta a chave no índice i para um novo valor
public void heapIncreaseKey(int i, int key) {
  if (i <= 0 || i >= heap.size())
    throw new IllegalArgumentException("Índice inválido");
  if (key < heap.get(i))
    throw new IllegalArgumentException("Nova chave é menor que a atual");

  heap.set(i, key); // atualiza a chave
  while (i > 1 && heap.get(parent(i)) < heap.get(i)) {
    swap(i, parent(i)); // sobe se for maior que o pai
    i = parent(i);
  }
}

// Insere uma nova chave no heap
public void maxHeapInsert(int key) {
  heap.add(Integer.MIN_VALUE); // adiciona um valor mínimo temporário
  heapIncreaseKey(heap.size() - 1, key); // aumenta até a posição correta
}

// Exibe o heap para debug
public void imprimirHeap() {
  for (int i = 1; i < heap.size(); i++) {
```

```
      System.out.print(heap.get(i) + " ");
    }
    System.out.println();
  }

  // Testa a implementação
  public static void main(String[] args) {
    MaxHeapPriorityQueue fila = new MaxHeapPriorityQueue();

    fila.maxHeapInsert(15);
    fila.maxHeapInsert(10);
    fila.maxHeapInsert(30);
    fila.maxHeapInsert(40);
    fila.maxHeapInsert(50);
    fila.imprimirHeap(); // deve mostrar uma heap válida

    System.out.println("Máximo: " + fila.heapMaximum());
    System.out.println("Extraído: " + fila.heapExtractMax());
    fila.imprimirHeap();

    fila.heapIncreaseKey(2, 60);
    fila.imprimirHeap();
  }
}
```

10_exercicios_analise_de_algoritmos_heap_miqueue_dijkstra.pdf

Uma empresa de logística quer transportar um item entre duas regiões da cidade utilizando rotas previamente mapeadas. A cidade é representada por um grafo direcionado, em que as regiões são os vértices e as vias entre elas são arestas com pesos positivos, representando o custo de deslocamento (tempo, distância, ou outro critério).
Para encontrar a melhor rota, a empresa precisa de um programa que calcule o caminho de menor custo entre uma região de origem e uma região de destino.
O que o programa deve fazer:

- Ler os dados do grafo: cada linha informa uma ligação entre duas regiões e o custo dessa ligação.
- Construir um grafo direcionado.
- Solicitar a região de origem e a região de destino.
- Calcular o caminho de menor custo entre origem e destino utilizando o algoritmo de Dijkstra.
    - Implementar a fila de prioridade com base em heap manualmente (sem usar bibliotecas prontas como PriorityQueue). Use o que fizemos em aula.
    - Exibir:

O caminho encontrado (na ordem correta, da origem ao destino).
O custo total do caminho.
Entrada esperada:

Um número inteiro representando o número de vias (arestas).
Em seguida, para cada via: uma linha com três valores: regiao_origem regiao_destino custo
Depois, duas linhas com os nomes da região de origem e da região de destino.
Professor Rodrigo Bossini <professorbossini@gmail.com>
Produção: 2025
2
Saída esperada:
O caminho de menor custo entre origem e destino.
O custo total do caminho.

1)

```
import java.util.*;

public class DijkstraHeapManual {

    // Classe que representa uma aresta com destino e peso
    static class Aresta {
        int destino;
        int peso;

        Aresta(int destino, int peso) {
            this.destino = destino;
            this.peso = peso;
        }
    }

    // Classe que representa um vértice com nome, índice, distância e predecessor
    static class Vertice {
        String nome;
        int indice;
        int distancia = Integer.MAX_VALUE;
        Vertice predecessor = null;

        Vertice(String nome, int indice) {
            this.nome = nome;
            this.indice = indice;
        }
    }

    // Classe que representa a heap mínima (baseada na distância do vértice)
    static class MinHeap {
        Vertice[] heap;
        int tamanho;

        MinHeap(int capacidade) {
            heap = new Vertice[capacidade];
            tamanho = 0;
        }

        // Insere um vértice na heap e ajusta
```

```
        void inserir(Vertice v) {
            heap[tamanho] = v;
            subir(tamanho);
            tamanho++;
        }

        // Extrai o vértice com menor distância
        Vertice extrairMin() {
            Vertice min = heap[0];
            heap[0] = heap[tamanho - 1];
            tamanho--;
            descer(0);
            return min;
        }

        boolean estaVazio() {
            return tamanho == 0;
        }

        // Corrige a heap de baixo para cima
        void subir(int i) {
            while (i > 0 && heap[i].distancia < heap[(i - 1) / 2].distancia) {
                trocar(i, (i - 1) / 2);
                i = (i - 1) / 2;
            }
        }

        // Corrige a heap de cima para baixo
        void descer(int i) {
            int menor = i;
            int esq = 2 * i + 1;
            int dir = 2 * i + 2;

            if (esq < tamanho && heap[esq].distancia < heap[menor].distancia)
                menor = esq;
            if (dir < tamanho && heap[dir].distancia < heap[menor].distancia)
                menor = dir;

            if (menor != i) {
                trocar(i, menor);
                descer(menor);
            }
        }

        void trocar(int i, int j) {
            Vertice temp = heap[i];
            heap[i] = heap[j];
            heap[j] = temp;
        }
    }

    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);

        // Lê o número de vias (arestas)
        int m = Integer.parseInt(sc.nextLine());

        // Map para associar nomes de regiões aos índices
        Map<String, Integer> mapa = new HashMap<>();
        List<String> nomes = new ArrayList<>();
        List<List<Aresta>> grafo = new ArrayList<>();

        // Lê as arestas e constrói o grafo
        for (int i = 0; i < m; i++) {
            String[] linha = sc.nextLine().split(" ");
            String origem = linha[0];
            String destino = linha[1];
            int peso = Integer.parseInt(linha[2]);

            // Se a região ainda não foi registrada, adiciona
            if (!mapa.containsKey(origem)) {
                mapa.put(origem, nomes.size());
                nomes.add(origem);
                grafo.add(new ArrayList<>());
            }
            if (!mapa.containsKey(destino)) {
                mapa.put(destino, nomes.size());
                nomes.add(destino);
                grafo.add(new ArrayList<>());
            }

            int u = mapa.get(origem);
            int v = mapa.get(destino);

            // Adiciona aresta direcionada u → v
            grafo.get(u).add(new Aresta(v, peso));
        }

        // Lê as regiões de origem e destino
        String origemNome = sc.nextLine();
        String destinoNome = sc.nextLine();

        int origemIndice = mapa.get(origemNome);
        int destinoIndice = mapa.get(destinoNome);

        // Inicializa os vértices
        Vertice[] vertices = new Vertice[nomes.size()];
        for (int i = 0; i < nomes.size(); i++) {
            vertices[i] = new Vertice(nomes.get(i), i);
        }

        // Dijkstra com heap manual
        vertices[origemIndice].distancia = 0;
        MinHeap heap = new MinHeap(nomes.size());
```

```
        for (Vertice v : vertices) {
            heap.inserir(v);
        }

        while (!heap.estaVazio()) {
            Vertice u = heap.extrairMin();

            // Para cada vizinho do vértice atual
            for (Aresta a : grafo.get(u.indice)) {
                Vertice v = vertices[a.destino];

                // Relaxamento: se encontrou caminho melhor, atualiza
                if (u.distancia + a.peso < v.distancia) {
                    v.distancia = u.distancia + a.peso;
                    v.predecessor = u;
                    // Após mudar distância, a heap pode ficar incorreta
                    // Para simplicidade, vamos reconstruí-la do zero
                    heap = new MinHeap(nomes.size());
                    for (Vertice x : vertices) {
                        if (x.distancia != Integer.MAX_VALUE) {
                            heap.inserir(x);
                        }
                    }
                }
            }
        }

        // Imprime o caminho de origem até destino
        List<String> caminho = new ArrayList<>();
        Vertice atual = vertices[destinoIndice];

        if (atual.distancia == Integer.MAX_VALUE) {
            System.out.println("Não há caminho entre " + origemNome + " e " + destinoNome +
".");
        } else {
            while (atual != null) {
                caminho.add(atual.nome);
                atual = atual.predecessor;
            }

            // Inverte o caminho (pois foi construído de trás para frente)
            Collections.reverse(caminho);
            System.out.println("Caminho de menor custo:");
            for (String nome : caminho) {
                System.out.print(nome + " ");
            }
            System.out.println("\nCusto total: " + vertices[destinoIndice].distancia);
        }
    }
}
```

2) Refaça usando PriorityQueue da API do Java.

```
import java.util.*;

public class DijkstraComPriorityQueue {

    // Classe que representa uma aresta com destino e peso
    static class Aresta {
        int destino;
        int peso;

        Aresta(int destino, int peso) {
            this.destino = destino;
            this.peso = peso;
        }
    }

    // Classe que representa um vértice com nome, índice, distância e predecessor
    static class Vertice implements Comparable<Vertice> {
        String nome;
        int indice;
        int distancia = Integer.MAX_VALUE; // Inicia com "infinito"
        Vertice predecessor = null;

        Vertice(String nome, int indice) {
            this.nome = nome;
            this.indice = indice;
        }

        // Comparação por distância -- usada na PriorityQueue
        @Override
        public int compareTo(Vertice outro) {
            return Integer.compare(this.distancia, outro.distancia);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Lê o número de vias (arestas do grafo)
        int m = Integer.parseInt(sc.nextLine());

        // Estrutura para mapear nomes de regiões a índices
        Map<String, Integer> mapa = new HashMap<>();
        List<String> nomes = new ArrayList<>();
        List<List<Aresta>> grafo = new ArrayList<>();

        // Leitura das vias: origem, destino e custo
        for (int i = 0; i < m; i++) {
            String[] linha = sc.nextLine().split(" ");
            String origem = linha[0];
            String destino = linha[1];
```

```
            int peso = Integer.parseInt(linha[2]);

            // Adiciona os vértices ao mapa se ainda não existirem
            if (!mapa.containsKey(origem)) {
                mapa.put(origem, nomes.size());
                nomes.add(origem);
                grafo.add(new ArrayList<>());
            }
            if (!mapa.containsKey(destino)) {
                mapa.put(destino, nomes.size());
                nomes.add(destino);
                grafo.add(new ArrayList<>());
            }

            int u = mapa.get(origem);
            int v = mapa.get(destino);

            // Adiciona a aresta dirigida u → v com peso
            grafo.get(u).add(new Aresta(v, peso));
        }

        // Lê os nomes das regiões de origem e destino
        String origemNome = sc.nextLine();
        String destinoNome = sc.nextLine();

        int origemIndice = mapa.get(origemNome);
        int destinoIndice = mapa.get(destinoNome);

        // Cria os vértices do grafo
        Vertice[] vertices = new Vertice[nomes.size()];
        for (int i = 0; i < nomes.size(); i++) {
            vertices[i] = new Vertice(nomes.get(i), i);
        }

        // Inicializa a origem com distância 0
        vertices[origemIndice].distancia = 0;

        // Cria a fila de prioridade com base na distância mínima
        PriorityQueue<Vertice> fila = new PriorityQueue<>();
        fila.offer(vertices[origemIndice]);

        // Dijkstra com PriorityQueue
        while (!fila.isEmpty()) {
            Vertice u = fila.poll(); // extrai vértice com menor distância

            // Explora os vizinhos do vértice atual
            for (Aresta a : grafo.get(u.indice)) {
                Vertice v = vertices[a.destino];

                // Relaxamento: se encontrou caminho melhor
                if (u.distancia + a.peso < v.distancia) {
                    v.distancia = u.distancia + a.peso;
```

```
                    v.predecessor = u;

                    // Adiciona v novamente na fila (com nova distância)
                    // A PriorityQueue não atualiza automaticamente os elementos
                    // então reinserimos v
                    fila.offer(v);
                }
            }
        }

        // Reconstrói o caminho da origem até o destino
        List<String> caminho = new ArrayList<>();
        Vertice atual = vertices[destinoIndice];

        if (atual.distancia == Integer.MAX_VALUE) {
            System.out.println("Não há caminho entre " + origemNome + " e " + destinoNome +
".");
        } else {
            while (atual != null) {
                caminho.add(atual.nome);
                atual = atual.predecessor;
            }

            // Inverte o caminho (foi construído de trás para frente)
            Collections.reverse(caminho);
            System.out.println("Caminho de menor custo:");
            for (String nome : caminho) {
                System.out.print(nome + " ");
            }
            System.out.println("\nCusto total: " + vertices[destinoIndice].distancia);
        }
    }
}
```