Análise de Complexidade: USEI11 - Análise de Conetividade e Subgrafos

1. Introdução Teórica

O objetivo da **USEI11** é analisar a conetividade da rede ferroviária, permitindo identificar se todos os pontos são alcançáveis e determinar os subgrafos (componentes ligadas) da rede. Para tal, foram implementados dois algoritmos fundamentais:

1. **Breadth-First Search (BFS):** Para determinar todas as estações alcançáveis a partir de um ponto de origem.
2. **Algoritmo de Dijkstra:** Para calcular o caminho mais rápido entre dois pontos, considerando as restrições de velocidade.

2. Análise de Complexidade Temporal

2.1 Conetividade (Método findAllReachableFacilities)

O método utiliza uma pesquisa em largura (BFS) para explorar a rede.

- **Construção da Lista de Adjacência:** O código percorre todos os segmentos (E) uma vez para criar o mapa bidirecional.
  - Complexidade: **O(E)**
- **Execução da BFS:** Cada estação (V) é inserida na fila uma única vez e cada ligação (E) é verificada no máximo duas vezes (uma para cada extremidade).
  - Complexidade: **O(V + E)**

Complexidade Temporal Total (Conetividade): O(V + E)

Código de Suporte (BFS):

```
public List<Integer> findAllReachableFacilities(int startFacilityId) {
Set<Integer> reachable = new HashSet<>();
Queue<Integer> queue = new LinkedList<>();
// 1. Constrói a Lista de Adjacência bidirecional
```

```
Map<Integer, List<LineSegment>> adj = new HashMap<>();
for (LineSegment seg : segmentoRepo.findAll()) {
   adj.computeIfAbsent(seg.getIdEstacaoInicio(), k -> new ArrayList<>()).ad
d(seg);
   adj.computeIfAbsent(seg.getIdEstacaoFim(), k -> new ArrayList<>()).add
(seg);
}

// 2. Inicializa a busca em largura (BFS)
queue.add(startFacilityId);
reachable.add(startFacilityId);

while (!queue.isEmpty()) {
   int current = queue.poll();
   for (LineSegment seg : adj.getOrDefault(current, Collections.emptyList
())) {
      int neighbor = (seg.getIdEstacaoInicio() == current) ? seg.getIdEstaca
oFim() : seg.getIdEstacaoInicio();
      if (!reachable.contains(neighbor)) {
         reachable.add(neighbor);
         queue.add(neighbor);
      }
   }
}
reachable.remove(startFacilityId);
return new ArrayList<>(reachable);
}
```

2.2 Caminho Mais Rápido (Método findFastestPath)

Utiliza o algoritmo de Dijkstra com uma fila de prioridade (Min-Heap).

- **Inicialização:** Atribuição de tempo infinito a todas as estações (V).
  - Complexidade: **O(V)**
- **Ciclo Principal:**
  - Extração do nó com menor tempo: **O(log V)**, executado no máximo V vezes.

- Relaxamento de arestas: Para cada ligação (E), verifica-se o tempo e atualiza-se a PriorityQueue (**O(log V)**).
- Complexidade: **O((V + E) * log V)**

Complexidade Temporal Total (Caminho): O((V + E) * log V)

Código de Suporte (Dijkstra):

```
// Relaxar SOMENTE as arestas (segmentos) que saem de 'u'
for (LineSegment seg : adj.getOrDefault(u, Collections.emptyList())) {
int v = seg.getIdEstacaoFim();
// Calcula o tempo de viagem: Tempo = Comprimento / min(VelocidadeSeg
mento, VelocidadeMaxima)
double effectiveSpeed = Math.min(seg.getVelocidadeMaxima(), maxSpeed
Limit);
double lengthKm = seg.getComprimento();
double travelTime = lengthKm / effectiveSpeed;

if (timeTo.get(u) + travelTime < timeTo.get(v)) {
    double newTime = timeTo.get(u) + travelTime;
    timeTo.put(v, newTime);
    edgeTo.put(v, seg);
    predecessorNode.put(v, u);

    // Atualização eficiente na PriorityQueue
    Map.Entry<Integer, Double> oldEntry = new AbstractMap.SimpleEntry<>
(v, timeTo.get(v));
    pq.remove(oldEntry);
    pq.add(new AbstractMap.SimpleEntry<>(v, newTime));
}
```

3. Análise de Complexidade Espacial

A memória necessária para suportar estas operações foca-se no armazenamento do grafo e estruturas auxiliares:

- **Estrutura do Grafo (Mapas de Adjacência):** Armazena todas as estações e as suas ligações.
  - Complexidade: **O(V + E)**

- **Estruturas Auxiliares (Set de visitados, PriorityQueue, Mapas de Predecessores):** No pior caso, armazenam informação sobre todos os nós da rede.
    - Complexidade: **O(V)**

Complexidade Espacial Total: O(V + E)

4. Fundamentação com o Código

A implementação no ficheiro **RailwayNetworkService.java** comprova estas métricas através de:

1. Uso de **HashMap** para garantir acesso em tempo constante **O(1)** aos nós durante a exploração.
2. Utilização de uma **PriorityQueue** para otimizar a seleção do próximo nó no algoritmo de Dijkstra, evitando a pesquisa linear **O(V)** que tornaria o algoritmo **$O(V^2)$**.
3. Implementação de uma **Queue** (LinkedList) e um **Set** (HashSet) no método `findAllReachableFacilities` para garantir que nenhum nó é processado redundantemente na análise de conetividade.

Análise de Complexidade: USEI12 - Minimal Backbone Network

1. Introdução Teórica

O objetivo desta User Story é identificar a rede mínima de transporte (Backbone) que ligue todas as estações alcançáveis com o menor comprimento total de via.

Em teoria de grafos, este problema é resolvido através de uma **Árvore Geradora Mínima (MST - Minimum Spanning Tree)**. O algoritmo escolhido para a implementação foi o **Algoritmo de Kruskal**, devido à sua eficiência em grafos esparsos (como são as redes ferroviárias) e facilidade de lidar com componentes ligadas.

2. Análise de Complexidade Temporal

A complexidade temporal do algoritmo de Kruskal é definida como:
O(E log E) ou **O(E log V)**

Onde:

- **V (Vertices):** Número total de estações.
- **E (Edges):** Número total de troços ferroviários.

Justificação detalhada:

1. **Leitura dos Dados:** O método percorre a lista de todos os segmentos para inicializar a estrutura.
    - Complexidade: **O(E)**
2. **Ordenação das Arestas:** As arestas (segmentos) devem ser ordenadas por peso (comprimento) de forma crescente. Esta é a etapa dominante.
    - Complexidade: **O(E log E)**
3. **Union-Find (Deteção de Ciclos):** Para cada aresta, verificamos se os nós pertencem ao mesmo conjunto para evitar ciclos. Usando *Path Compression* e *Union by Rank*:

- Complexidade: **O(E · α(V))**, onde **α** é a função inversa de Ackermann (praticamente constante).

Complexidade Total: O(E log E)

3. Análise de Complexidade Espacial

A complexidade espacial refere-se à memória necessária para armazenar o grafo e as estruturas auxiliares (DSU).

Complexidade: O(V + E)

Justificação:

- **Lista de Adjacência/Arestas:** Armazenamos os **E** segmentos originais.
- **Estrutura Disjoint Set (DSU):** Necessita de arrays para "pais" e "rank" de tamanho **V**.
- **Grafo Resultante:** A árvore geradora final contém **V - 1** arestas.

4. Fundamentação com o Código

Abaixo está o exemplo da lógica implementada no `RailwayNetworkService.java` que comprova a análise:

```
public List<LineSegment> computeMinimumBackbone() {
// 1. Obter todos os troços - O(E)
List<LineSegment> allEdges = segmentoRepo.findAll();
// 2. Ordenar por comprimento - O(E log E)
allEdges.sort(Comparator.comparingDouble(LineSegment::getCompriment
o));

// 3. Inicializar Union-Find - O(V)
DisjointSet dsu = new DisjointSet(facilityRepo.count());
List<LineSegment> mst = new ArrayList<>();

// 4. Processar arestas - O(E * alpha(V))
for (LineSegment edge : allEdges) {
    if (dsu.find(edge.getV1()) != dsu.find(edge.getV2())) {
        dsu.union(edge.getV1(), edge.getV2());
```

```
        mst.add(edge);
    }
}
return mst;
}
```

5. Conclusão para o Dataset

Considerando os dados da rede ferroviária (ex: Bélgica):

- **V ≈ 559**
- **E ≈ 691**

O algoritmo de Kruskal é ideal, pois o número de arestas (**E**) é muito próximo do número de vértices (**V**). A ordenação de apenas 691 elementos é processada em poucos milissegundos, garantindo que a aplicação responda rapidamente mesmo com o crescimento da rede Europeia.

Análise de Complexidade: USEI13 - Network Metrics

1. Introdução Teórica

O objetivo da USEI13 é calcular métricas de centralidade (como *Betweenness* ou *Closeness*) para identificar as estações mais influentes na rede ferroviária. Estas métricas permitem determinar quais as estações que funcionam como "pontes" ou "hubs" logísticos.

A base algorítmica para estas métricas assenta na execução repetida do **Algoritmo de Dijkstra** para encontrar os caminhos mais curtos entre todos os pares de nós da rede (*All-Pairs Shortest Paths*).

2. Análise de Complexidade Temporal

A complexidade temporal para o cálculo de métricas globais é:
O(V * (E log V))

Onde:

- **V (Vertices):** Número total de estações (~559 no dataset da Bélgica).
- **E (Edges):** Número total de troços ferroviários (~691).

Justificação Detalhada:

1. **Iteração Total:** O algoritmo tem de calcular o caminho mais curto a partir de cada uma das **V** estações.
2. **Custo do Dijkstra:** Para cada iteração, o Dijkstra (usando uma *Priority Queue*) tem um custo de **O(E log V)**.
3. **Agregação de Resultados:** No final de cada Dijkstra, percorrem-se os resultados para somar as distâncias ou contar passagens em nós, o que acrescenta um custo de **O(V)** por iteração.

Complexidade Total: O(V * E log V)

3. Análise de Complexidade Espacial

A complexidade espacial foca-se no armazenamento da estrutura do grafo e dos resultados das métricas por nó.

Complexidade: O(V + E)

Justificação:

- **Estrutura do Grafo:** A lista de adjacência ocupa **O(V + E)**.
- **Resultados das Métricas:** É necessário um mapa ou array para guardar o valor final da métrica para cada uma das **V** estações, resultando em **O(V)**.
- **Memória Auxiliar:** Durante a execução, as estruturas de "distâncias" e "predecessores" ocupam **O(V)**.

4. Fundamentação com o Código

O trecho de código abaixo (baseado no `RailwayNetworkService.java`) exemplifica a execução iterativa que justifica a complexidade:

```
public Map<Integer, Double> calculateClosenessCentrality() {
    Map<Integer, Double> closenessMap = new HashMap<>();
    List<Integer> allStations = repository.findAllIds();

    for (Integer startNode : allStations) { // Executa V vezes
        // Dijkstra para o nó atual - O(E log V)
        Map<Integer, Double> distances = runDijkstra(startNode);

        double sumDistances = 0;
        for (Double d : distances.values()) {
            sumDistances += d;
        }

        // Closeness = (n-1) / soma das distâncias
        closenessMap.put(startNode, (allStations.size() - 1) / sumDistances);
    }
    return closenessMap;
}
```

5. Conclusão para o Dataset

No contexto da rede belga:

- **V = 559**, **E = 691**.
- O cálculo de caminhos para todos os pares envolve aproximadamente 559 execuções do Dijkstra.
- Devido à rede ser esparsa (E pequeno), o tempo de processamento é reduzido (escassos segundos), cumprindo os requisitos de eficiência do sistema mesmo para uma análise estatística pesada.

Análise de Complexidade: USEI14 - Maximum Throughput (Fluxo Máximo)

1. Introdução Teórica

O objetivo da USEI14 é calcular a capacidade máxima de transporte de carga entre duas estações da rede ferroviária. Para solucionar este problema de fluxo em grafos, foi implementado o **Algoritmo de Edmonds-Karp**.

Este algoritmo é uma variação do método de Ford-Fulkerson que utiliza uma pesquisa em largura (**BFS**) para encontrar caminhos de aumento (*augmenting paths*). A utilização da BFS garante que o caminho escolhido seja sempre o mais curto em número de arestas, conferindo ao algoritmo uma complexidade temporal polinomial independente dos valores das capacidades.

2. Análise de Complexidade Temporal

A complexidade temporal teórica do algoritmo de Edmonds-Karp é:
$O(V * E^2)$

Onde:

- **V (Vértices):** Representa o número total de estações.
- **E (Edges):** Representa o número total de troços ferroviários.

Justificação detalhada:

1. **Cálculo de Caminhos (BFS):** O algoritmo executa sucessivas pesquisas BFS no grafo residual. Cada execução da BFS percorre o grafo em **$O(V + E)$**. Em redes ferroviárias reais, o número de ligações é reduzido, simplificando para **$O(E)$**.
2. **Número de Iterações:** No pior caso, o número total de caminhos de aumento encontrados pelo Edmonds-Karp é **$O(V * E)$**.
3. **Complexidade Total:** Multiplicando o custo de cada pesquisa (**$O(E)$**) pelo número máximo de iterações (**$O(V * E)$**), obtemos o limite superior de **$O(V * E^2)$**.

3. Análise de Complexidade Espacial

A complexidade espacial foca-se na memória necessária para as estruturas de dados de suporte.

Complexidade Total: O(V + E)

Justificação:

- **Grafo Residual:** Armazenado através de um `Map<Integer, Map<Integer, Double>>` (Mapa de Adjacência). Esta estrutura é altamente eficiente para grafos esparsos, ocupando apenas o espaço necessário para os nós e as suas ligações reais, evitando o custo **$O(V^2)$** de uma matriz.
- **Estruturas Auxiliares:** O uso de um `parentMap` para reconstruir o caminho e um `Set` de visitados durante a BFS consome espaço proporcional ao número de estações, ou seja, **O(V)**.

4. Fundamentação com o Código

A lógica presente no `RailwayFlowService.java` reflete esta análise através do seguinte fluxo:

```
public double calculateMaxFlow(int source, int sink) {
    // 1. Construção do Grafo Residual - O(V + E)
    Map<Integer, Map<Integer, Double>> residualGraph = initializeResidualG
raph();
    double maxFlow = 0;

    // 2. Loop de procura de caminhos - O(V * E) iterações
    while (true) {
        Map<Integer, Integer> parentMap = new HashMap<>();

        // Pesquisa em Largura (BFS) para encontrar caminho de aumento - O
(E)
        if (!bfs(residualGraph, source, sink, parentMap)) {
            break; // Não há mais caminhos disponíveis
        }

        // 3. Identificação do gargalo (bottleneck) no caminho encontrado
        double pathFlow = Double.MAX_VALUE;
        int current = sink;
        while (current != source) {
```

```
            int prev = parentMap.get(current);
            pathFlow = Math.min(pathFlow, residualGraph.get(prev).get(curren
t));
            current = prev;
        }

        // 4. Atualização das capacidades residuais (Forward e Backward)
        current = sink;
        while (current != source) {
            int prev = parentMap.get(current);
            // Reduz capacidade na aresta direta
            residualGraph.get(prev).put(current, residualGraph.get(prev).get(cu
rrent) - pathFlow);
            // Aumenta capacidade na aresta residual (backward)
            residualGraph.get(current).put(prev, residualGraph.get(current).get
OrDefault(prev, 0.0) + pathFlow);
            current = prev;
        }
        maxFlow += pathFlow;
    }
    return maxFlow;
}
```

5. Conclusão para o Dataset

No dataset da Bélgica (~559 estações e ~691 troços):

- Sendo a rede **esparsa** (onde o número de arestas **E** é próximo de **V**), a performance é excelente.
- O algoritmo processa o fluxo máximo em milissegundos, garantindo que o sistema suporte análises de carga em tempo real sem degradação de performance.

Análise de Complexidade: USEI15 - Optimal Upgrade Plan

1. Introdução Teórica

O objetivo da USEI15 é determinar o plano de upgrade mais económico para a rede ferroviária, garantindo que a rede se mantenha totalmente conectada (ou seja, que todas as estações originais continuem a poder comunicar entre si) após a substituição de troços antigos por novos troços de alta performance.

Este é um problema clássico de **Árvore Geradora Mínima (MST)**. A diferença para a USEI12 é que aqui o "peso" das arestas não é a distância geográfica, mas sim o **custo financeiro** de upgrade de cada segmento.

2. Análise de Complexidade Temporal

A complexidade temporal baseia-se no algoritmo de **Kruskal**, que é o mais eficiente para este tipo de rede (grafos esparsos):

Complexidade Total: O(E log E)

Justificação por Etapas:

1. **Filtragem e Preparação (O(E)):** O sistema identifica todos os segmentos candidatos a upgrade e os seus respetivos custos.
2. **Ordenação por Custo (O(E log E)):** Esta é a operação dominante. Ordenamos os segmentos do menor custo para o maior custo para garantir a "ganância" (greediness) do algoritmo.
3. **Processamento de Conetividade (O(E * α(V))):** Utilizamos a estrutura *Disjoint Set Union* (DSU) com *path compression*. O termo **α(V)** é a função inversa de Ackermann, que para valores reais de estações (V < 10^6) é considerada constante (~4).

3. Análise de Complexidade Espacial

A complexidade espacial foca-se na manutenção das estruturas de suporte ao grafo residual e à árvore final.

Complexidade Total: O(V + E)

Justificação:

- **V (Vértices):** Necessário para armazenar o array de "pais" (parent pointers) no DSU para as ~559 estações.
- **E (Arestas):** Necessário para carregar a lista de todos os segmentos ferroviários da base de dados antes da ordenação.

4. Fundamentação com o Código

A lógica implementada no `UpgradePlanService.java` reflete esta eficiência:

```
public UpgradePlan calculateOptimalPlan() {
    // 1. Obter todos os segmentos candidatos - O(E)
    List<SegmentUpgrade> candidates = repo.getUpgradeCandidates();

    // 2. Ordenar por custo ascendente - O(E log E)
    candidates.sort(Comparator.comparingDouble(SegmentUpgrade::getCo
st));

    // 3. Aplicar Kruskal para MST - O(E * alpha(V))
    DisjointSet dsu = new DisjointSet(totalStations);
    List<SegmentUpgrade> plan = new ArrayList<>();

    for (SegmentUpgrade s : candidates) {
        if (dsu.find(s.u) != dsu.find(s.v)) {
            dsu.union(s.u, s.v);
            plan.add(s);
        }
    }
    return new UpgradePlan(plan);
}
```

5. Conclusão para o Dataset

Para o dataset da Bélgica (~559 estações e ~691 troços):

- O número de arestas **E** é pequeno, pelo que a ordenação **O(E log E)** é executada instantaneamente.

- O resultado garante o custo mínimo total (Requirement: "total cost is minimum") respeitando a restrição de manter a rede conectada.