Análise de Complexidade: USEI14 - Maximum Throughput

1. Introdução Teórica

O objetivo desta User Story é calcular o fluxo máximo entre dois nós (Estações) num grafo direcionado com capacidades. O algoritmo selecionado foi o **Edmonds-Karp**, que é uma implementação específica do método de **Ford-Fulkerson**.

A diferença fundamental (e o motivo da escolha) é que o Edmonds-Karp utiliza uma pesquisa em largura (**BFS - Breadth-First Search**) para encontrar os caminhos de aumento (*augmenting paths*) no grafo residual. Isto garante que escolhemos sempre o caminho com o **menor número de arestas**, o que evita loops infinitos e garante uma complexidade temporal polinomial, independente do valor máximo do fluxo.

2. Análise de Complexidade Temporal

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

Onde:

- **V (Vertices):** Número de estações (|V|).
- **E (Edges):** Número de segmentos de linha (|E|).

2.1 Justificação Teórica

O algoritmo funciona em iterações (o loop `while`). Em cada iteração, ocorre:

1. Uma pesquisa BFS para encontrar o caminho mais curto no grafo residual.
2. Uma atualização do fluxo ao longo desse caminho.

Porque é O(V x E^2)**?**

- **Custo do BFS:** Usando Mapas de Adjacência (como implementado), um BFS percorre o grafo em **O(V + E)**, que simplificamos para **O(E)** num grafo conexo onde E ≥ V-1.
- **Número de Iterações (Augmentations):**

- Cada vez que aumentamos o fluxo, pelo menos uma aresta no grafo residual torna-se "saturada" (capacidade residual chega a 0) ou crítica.
- Teoricamente, prova-se que a distância (número de arestas) da fonte a qualquer nó no grafo residual nunca diminui.
- Cada aresta pode tornar-se crítica no máximo $|V|/2$vezes.
- Como existem |E| arestas, o número total de caminhos de aumento encontrados é limitado por **$O(V x E)**.

Cálculo Final:

$$\text{Total} = (\text{Custo do BFS}) \times (\text{Número Máximo de Augmentations})$$

$$\text{Total} = O(E) \times O(V \cdot E) = O(V \cdot E^2)$$

2.2 Comprovação com o Código (Code Mapping)

Abaixo mapeamos a teoria à implementação na classe `RailwayFlowService` (ou `RailwayNetworkService`).

A. Inicialização do Grafo Residual

```
// Complexidade: O(E)
// Iteramos sobre todos os segmentos para criar o mapa de capacidades
for (LineSegment seg : csvSegments) {
    // operações de put em HashMap são O(1) médio
    residualGraph.computeIfAbsent(...).put(...);
}
```

- **Análise:** A construção é linear em relação ao número de segmentos. $O(E)$.

B. O Ciclo Principal (Augmentations)

```
// Complexidade: O(V * E) iterações no pior caso
while (bfsAugmentingPath(residualGraph, sourceId, sinkId, parentMap)) {
    // 1. Encontrar o gargalo (bottleneck)
    // O loop percorre o caminho encontrado (no máximo V vértices)
    while (curr != sourceId) { ... } // O(V)

    // 2. Atualizar Grafo Residual
```

```
    // Percorre o mesmo caminho para atualizar fluxos forward/backward
    while (curr != sourceId) { ... } // O(V)
}
```

- **Análise:** O corpo do loop `while` (excluindo o BFS) é dominado pelo comprimento do caminho, que é no máximo $O(V)$. Contudo, o termo dominante dentro da condição do `while` é o BFS.

C. A Pesquisa BFS

```
// Complexidade: O(V + E) → O(E)
private boolean bfsAugmentingPath(...) {
    // ... setup da Queue ...
    while (!q.isEmpty()) { // Visita cada vértice no máximo uma vez: O(V)
        int u = q.poll();
        // Itera sobre as arestas adjacentes
        for (Map.Entry<Integer, Double> entry : neighbors.entrySet()) { // Total de iterações = O(E)
            // ... lógica de visita ...
        }
    }
}
```

Análise: O método `bfsAugmentingPath` implementa o BFS clássico. Com a representação `Map<Integer, Map<Integer, Double>>` (Lista de Adjacência), visitamos cada vértice e cada aresta no máximo uma vez. Logo, custa **$O(E)$**.

2.3 Resumo da Execução

Combinando as partes:
1. O loop `while` executa $O(V x E)$ vezes.
2. Dentro da condição do loop, chamamos `bfsAugmentingPath` que custa O(E).
3. As atualizações dentro do loop custam O(V), que é insignificante comparado com O(E) do BFS (pois V ≤ E em redes ferroviárias típicas).

$$\text{Complexidade} \approx O(V \cdot E) \times O(E) = O(V \cdot E^2)$$

3. Análise de Complexidade Espacial

A complexidade espacial refere-se à memória necessária para executar o algoritmo.

Complexidade: $O(V + E)$

Justificação:

1. **Grafo Residual (`residualGraph`):**
   - Armazenamos um `HashMap` onde as chaves são os Vértices (V).
   - Os valores são mapas de arestas. Para cada aresta original, criamos uma aresta *forward* e uma *backward*.
   - Espaço total: O(V + 2E) approx O(V + E).
2. **Estruturas Auxiliares do BFS:**
   - `Queue<Integer> q`: No pior caso armazena todos os vértices. O(V).
   - `Map<Integer, Integer> parentMap`: Armazena o caminho de reconstrução, uma entrada por vértice visitado. O(V).
   - `Set<Integer> visited`: Armazena vértices visitados. O(V).

Total:O(V + E) + O(V) + O(V) = O(V + E)

Como usamos **Mapas de Adjacência** (HashMaps) em vez de Matrizes de Adjacência (O(V^2)), a solução é eficiente em memória para grafos esparsos como redes ferroviárias (onde E \ll V^2).

4. Conclusão para o Projeto

No contexto da rede ferroviária belga (aprox. 559 estações e 691 linhas):

• $V \approx 559$

• $E \approx 691$

O algoritmo **Edmonds-Karp** é altamente adequado. Embora $O(V \cdot E^2)$ pareça pesado para grafos densos, em redes de transporte (grafos esparsos), o desempenho é excelente (ordem de milissegundos), como comprovado pelos testes unitários e logs de execução. A escolha de não usar Ford-Fulkerson com DFS (que poderia ter complexidade exponencial dependendo das capacidades) garante a estabilidade da solução.