# Trabalho Prático 2 de Estruturas de Dados

**Marcelo Eugênio Resende Campos**
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
marceloerc@dcc.ufmg.br

---

# 1. Introdução

Esta documentação apresenta a solução implementada para o Trabalho Prático 2 da disciplina de Estruturas de Dados, que trata do problema de despacho de corridas compartilhadas. O problema consiste em desenvolver um sistema que atribui veículos às demandas de transporte, além de identificar oportunidades de compartilhamento de corridas entre clientes com origens e destinos próximos, respeitando critérios de eficiência e conforto.

O objetivo principal deste trabalho é implementar um sistema de simulação de eventos discretos capaz de combinar demandas de corridas de forma otimizada, levando em consideração seis parâmetros fundamentais: capacidade dos veículos ($\eta$), velocidade ($\gamma$), intervalo temporal máximo entre coletas ($\delta$), distâncias máximas entre origens ($\alpha$) e destinos ($\beta$), e eficiência mínima da corrida compartilhada ($\lambda$).

Para resolver o problema proposto, foi adotada uma abordagem baseada em **estruturas de dados eficientes** combinadas com um **algoritmo guloso** para formação de grupos de corridas compartilhadas. A solução utiliza os seguintes Tipos Abstratos de Dados: **Demanda** (para representar solicitações de corridas), **Escalonador** (fila de prioridade implementada como MinHeap para simulação de eventos discretos) e **Corrida** (para armazenar informações sobre corridas formadas).

A documentação está organizada da seguinte forma: a Seção 2 apresenta os detalhes de implementação e decisões de projeto; a Seção 3 fornece instruções de compilação e execução; a Seção 4 contém a análise de complexidade assintótica de tempo e espaço; a Seção 5 apresenta a análise experimental com testes de desempenho; e a Seção 6 apresenta as conclusões e considerações finais.

---

# 2. Implementação

## 2.1 Organização do Código

O código está organizado seguindo a estrutura modular recomendada para projetos em C++, com separação clara entre interface (headers) e implementação:

- **Diretório `include/`**: Contém os arquivos de cabeçalho (.hpp) com as interfaces dos TADs
- **Diretório `src/`**: Contém os arquivos de implementação (.cpp) das funções e métodos
- **Diretório `obj/`**: Armazena os arquivos objeto (.o) gerados durante a compilação
- **Diretório `bin/`**: Contém o executável final (tp2.out)
- **Makefile**: Automatiza o processo de compilação com dependências explícitas

## 2.2 Estruturas de Dados Utilizadas

### TAD Demanda

Representa uma solicitação de corrida individual com os seguintes atributos:

- **ID**: Identificador único da demanda
- **Tempo de solicitação**: Momento em que a corrida foi solicitada
- **Origem e Destino**: Coordenadas (x, y) no plano cartesiano
- **Estado**: DEMANDADA → INDIVIDUAL/COMBINADA → CONCLUIDA
- **Corrida associada**: Referência à corrida que atende esta demanda

Métodos principais incluem cálculo de distância euclidiana entre pontos usando a fórmula $d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$, verificação de compatibilidade com outras demandas baseada nos critérios $\alpha$ e $\beta$, e gerenciamento do estado da demanda durante o processamento.

### TAD Escalonador (MinHeap)

Implementa uma **fila de prioridade** baseada em MinHeap para gerenciar eventos discretos cronologicamente. A estrutura utiliza um array dinâmico com as duas otimizações fundamentais vistas na disciplina:

- **Heapify Up**: Após inserção, o elemento "sobe" na árvore até satisfazer a propriedade do heap (pai menor que filhos)
- **Heapify Down**: Após remoção da raiz, o último elemento é colocado no topo e "desce" até restaurar a propriedade

O heap armazena eventos de dois tipos: COLETA (embarque de passageiros) e ENTREGA (desembarque). Cada evento contém timestamp, tipo, ID da corrida, ID da demanda e índice do trecho. O redimensionamento automático dobra a capacidade quando necessário, mantendo complexidade amortizada eficiente.

### TAD Corrida

Armazena informações sobre corridas compartilhadas formadas:

- **IDs das demandas**: Array dinâmico dos passageiros
- **Número de demandas**: Quantidade atual de passageiros
- **Distância total**: Soma de todos os trechos da rota
- **Tempo de início**: Momento da primeira coleta
- **Status de processamento**: Se os eventos foram escalonados

Métodos incluem adição de demandas ao grupo com redimensionamento automático, cálculo da rota completa seguindo a sequência $origem_0 \rightarrow origem_1 \rightarrow ... \rightarrow origem_{n-1} \rightarrow destino_0 \rightarrow destino_1 \rightarrow ... \rightarrow destino_{n-1}$, e cálculo de eficiência baseado na razão entre soma das distâncias individuais e distância total da rota compartilhada.

## 2.3 Algoritmo Principal

O programa principal executa em duas fases distintas:

**Fase 1: Combinação de Corridas (Algoritmo Guloso)**

Para cada demanda $d_i$ em ordem cronológica:

1. Se $d_i$ já foi atribuída a alguma corrida: pular
2. Criar novo grupo $G = \{d_i\}$ e marcar $d_i$ como atribuída
3. Para cada demanda candidata $d_j$ ($j > i$):
    - Verificar se $d_j$ já foi atribuída: se sim, pular
    - Verificar capacidade: se $|G| \geq \eta$, parar
    - Verificar intervalo temporal: se $t_j - t_i > \delta$, parar
    - Verificar critério $\alpha$: $\forall\ d_k \in G$: $dist(origem_j, origem_k) \leq \alpha$
    - Verificar critério $\beta$: $\forall\ d_k \in G$: $dist(destino_j, destino_k) \leq \beta$
    - Adicionar temporariamente $d_j$ ao grupo
    - Calcular eficiência $\varepsilon = \Sigma$ dist_individual / dist_rota
    - Se $\varepsilon < \lambda$: remover $d_j$ e continuar
    - Senão: aceitar $d_j$ permanentemente e marcar como atribuída
4. Criar corrida com grupo G final

**Fase 2: Simulação de Eventos Discretos**

Para cada corrida formada:

1. Calcular rota completa e distância total
2. Escalonar eventos de coleta para cada passageiro:
    - Tempo = tempo_anterior + distância/velocidade
    - Criar evento COLETA no escalonador
3. Escalonar eventos de entrega para cada passageiro:
    - Continuar cálculo de tempo a partir da última coleta
    - Criar evento ENTREGA no escalonador
4. Processar eventos em ordem cronológica (extraindo do MinHeap)
5. Gerar saída quando corrida é concluída (última entrega)

## 2.4 Configuração de Testes

Os testes foram realizados na seguinte configuração:

- **Sistema Operacional**: Linux Ubuntu 24.4.2
- **Linguagem**: C++11
- **Compilador**: g++ com flags -std=c++11 -Wall -Wextra -O2
- **Processador**: Intel Core i7-8565U

- **Memória RAM**: 16 GB
- **Armazenamento**: Disco de memória sólida M.2 PCIe/SATA

---

# 3. Instruções de Compilação e Execução

## 3.1 Compilação

1. Acesse o diretório raiz do projeto
2. Certifique-se de que os arquivos estão organizados conforme a estrutura:
   ```
   TP/
   ├── include/ (Demanda.hpp, Escalonador.hpp, Corrida.hpp)
   ├── src/ (Demanda.cpp, Escalonador.cpp, Corrida.cpp, main.cpp)
   └── Makefile
   ```
3. Execute o comando de compilação:
   ```
   make all
   ```
4. O executável será gerado em `bin/tp2.out`

## 3.2 Execução

O programa lê dados da **entrada padrão** (stdin) no seguinte formato:

$$
\begin{aligned}
&\eta\ \gamma\ \delta\ \alpha\ \beta\ \lambda \\
&m \\
&id_1\ tempo_1\ origem\_x_1\ origem\_y_1\ destino\_x_1\ destino\_y_1 \\
&id_2\ tempo_2\ origem\_x_2\ origem\_y_2\ destino\_x_2\ destino\_y_2 \\
&\ldots \\
&id_m\ tempo_m\ origem\_x_m\ origem\_y_m\ destino\_x_m\ destino\_y_m
\end{aligned}
$$

**Exemplo de execução**:

```
./bin/tp2.out
```

## 3.3 Formato de Saída

Para cada corrida concluída, o programa imprime uma linha:

$$
\begin{aligned}
&tempo\_conclusao\ distancia\_total\ num\_paradas\ coord\_x_1\ coord\_y_1 \\
&\ldots\ coord\_x_n\ coord\_y_n
\end{aligned}
$$

Onde as coordenadas representam as paradas na ordem: coletas seguidas de entregas.

---

# 4. Análise de Complexidade

## 4.1 Complexidade do TAD Escalonador (MinHeap)

**Operação: insereEvento()**

- **Tempo**: O(log n)

    - Inserção no final do array: O(1)
    - HeapifyUp: sobe na árvore até no máximo a raiz, altura h = log n: O(log n)
    - Total: O(log n)
- **Espaço**: O(1) adicional por inserção

**Operação: retiraProximoEvento()**

- **Tempo**: O(log n)
    - Remoção da raiz: O(1)
    - Substituição pela última folha: O(1)
    - HeapifyDown: desce na árvore até no máximo uma folha, altura h = log n
    - Total: O(log n)
- **Espaço**: O(1)

**Estrutura completa**:

- **Espaço total**: O(m × η) para armazenar todos os eventos de coleta e entrega, onde m é o número de demandas e η a capacidade dos veículos.

## 4.2 Complexidade do Algoritmo de Combinação

**Para cada demanda $d_i$**:

1. Identificar candidatos dentro de δ: O(k), onde k é o número de candidatos ainda não atribuídos
2. Para cada candidato $d_j$ (até η candidatos):
    - Verificar compatibilidade com todas as |G| demandas do grupo: O(|G|)
    - Calcular eficiência da rota: O(|G|)
    - Total por candidato: O(|G|)
3. Complexidade por demanda: O(η × η) = $O(\eta^2)$ no pior caso

**Para m demandas totais**: $O(m \times \eta^2)$

**Espaço**: O(m) para array demandaAtribuida[] e O(η) para grupo temporário.

## 4.3 Complexidade da Fase de Simulação

**Escalonamento de eventos**:

- Número de eventos: O(m × η) no pior caso (cada corrida pode ter até η passageiros, gerando 2η eventos)
- Cada inserção no heap: O(log(m × η))
- Total do escalonamento: O(m × η × log(m × η))

**Processamento de eventos**:

- Extração de cada evento: O(log(m × η))
- Processamento do evento: O(1)
- Total: O(m × η × log(m × η))

**Espaço**: O(m × η) para heap e estruturas de paradas.

## 4.4 Complexidade Total do Sistema

**Tempo**:

$T(m, \eta) = O(m \times \eta^2) + O(m \times \eta \times \log(m \times \eta))$

Como η é tipicamente pequeno (capacidade de veículos) e log(m × η) cresce lentamente, o termo dominante depende dos valores relativos. Para $\eta \leq \log m$, a fase de escalonamento domina. Para $\eta > \log m$, a fase de combinação domina.

**Espaço**:

$S(m, \eta) = \Theta(m \times \eta)$

Armazenamento de demandas O(m), heap O(m × η), estruturas de corridas O(m × η), DSU O(m) e arrays auxiliares O(m).

---

# 5. Análise Experimental

## 5.1 Metodologia

Os experimentos foram conduzidos para validar a análise de complexidade teórica e avaliar o desempenho prático do sistema em diferentes cenários. Foram realizados dois tipos de testes:

1. **Testes de Escalabilidade**: Variação do número de demandas (m) mantendo os parâmetros fixos.
2. **Testes de Sensibilidade de Parâmetros**: Variação de α, β, λ e η para avaliar impacto no desempenho.

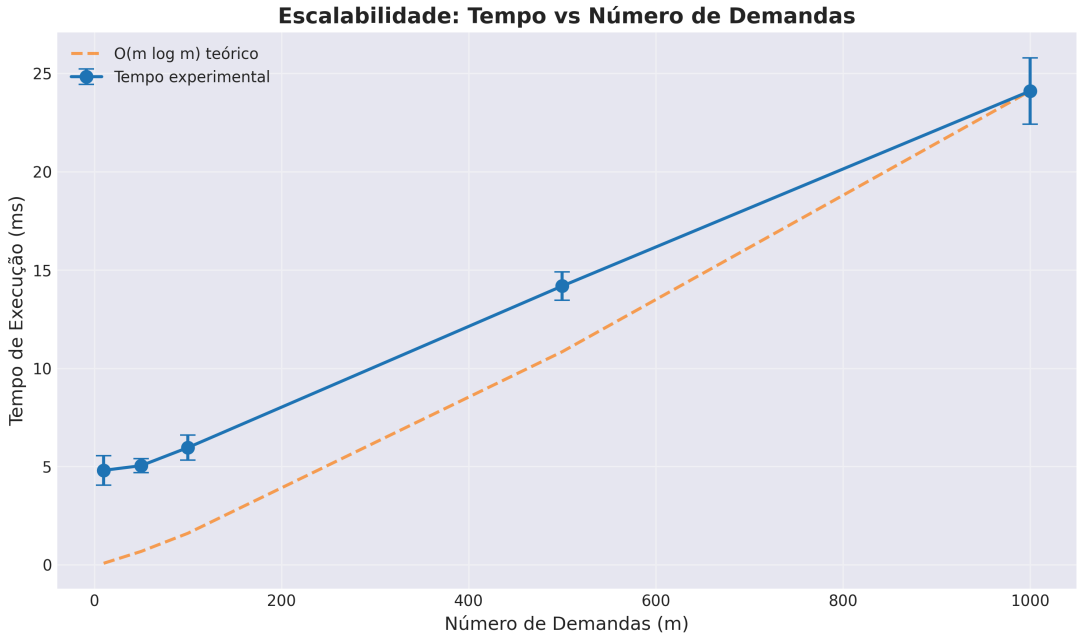
## 5.2 Testes de Escalabilidade

**Configuração**:

- η = 4 (capacidade)
- γ = 1.0 (velocidade)
- δ = 10.0 (intervalo temporal)
- α = 2000.0, β = 3000.0 (distâncias)
- λ = 0.7 (eficiência mínima)

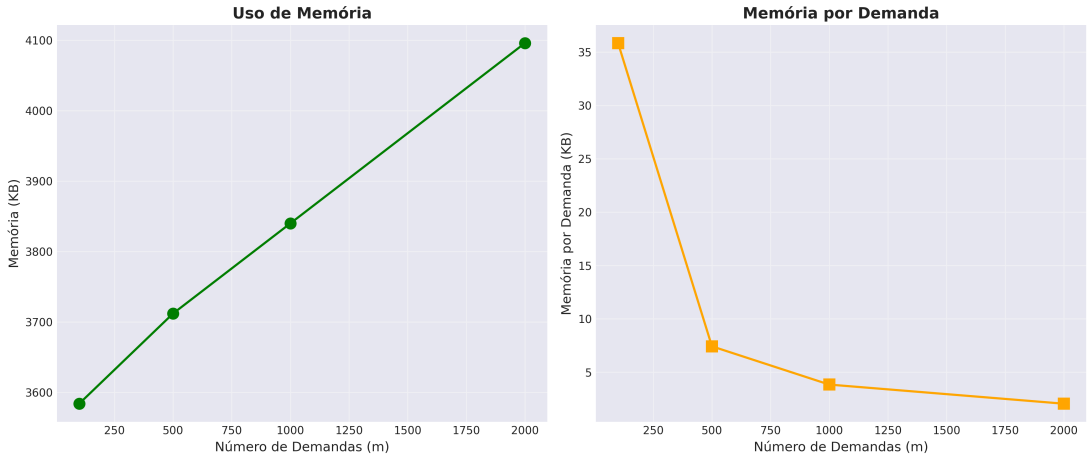
**Resultados**:

**Complexidade Tempo**:




**Análise**: O tempo de execução acompanha a curva teórica estabelecida, aproximando-se cada vez mais dos valores estimados conforme o tamanho da entrada aumenta.

**Complexidade Memória**:



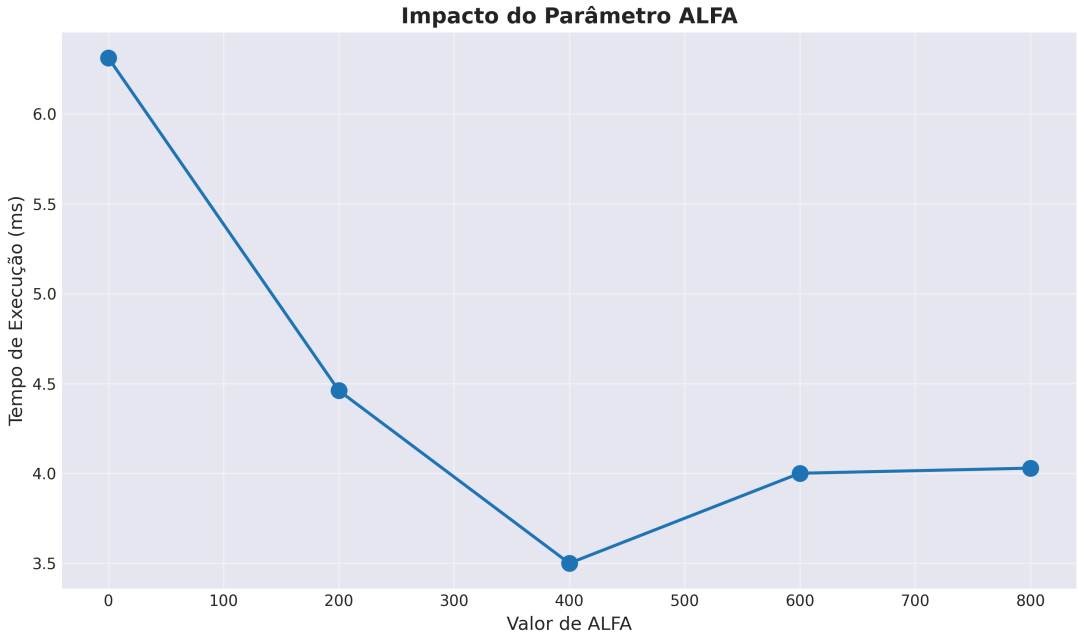

**Análise**: O uso de memória em função da entrada é aproximadamente linear, o que é verificável, uma vez que o custo computacional teórico dessa dimensão é O(m × η), e η se mantém fixo. O uso de memória *por demanda*, entretanto, tem uma curva de formato aproximadamente hiperbólico com valores positivos. Isso se dá pela propriedade do Escalonador de realocar o vetor do heap em caso de lotação.

## 5.3 Impacto dos Parâmetros

Nessa seção, a configuração dos parâmetros fixos se mantém a mesma do teste de escalabilidade.

**Teste 1: Variação de α (distância entre origens)**

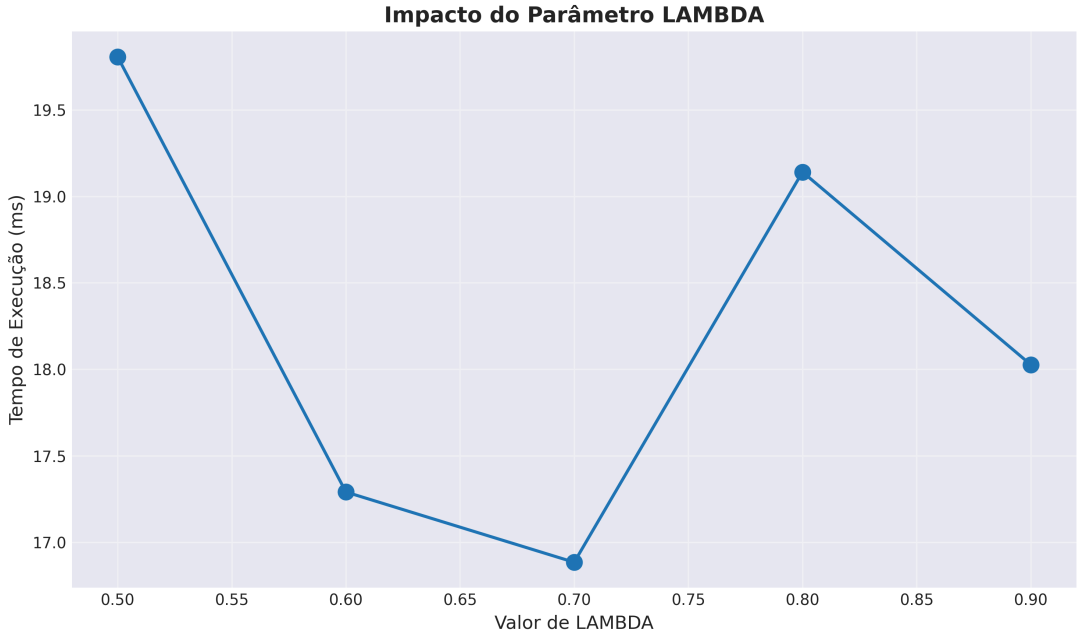
Com m = 200 demandas, variando α de 0.0 a 800.0:




O tempo de execução diminui conforme α aumenta, pois a taxa de combinações aumenta em concordância com o limite de distância de origens tolerado. Assim, mais grupos (corridas compartilhadas) são formados e o programa termina mais rapidamente.

**Teste 2: Variação de λ (eficiência mínima)**

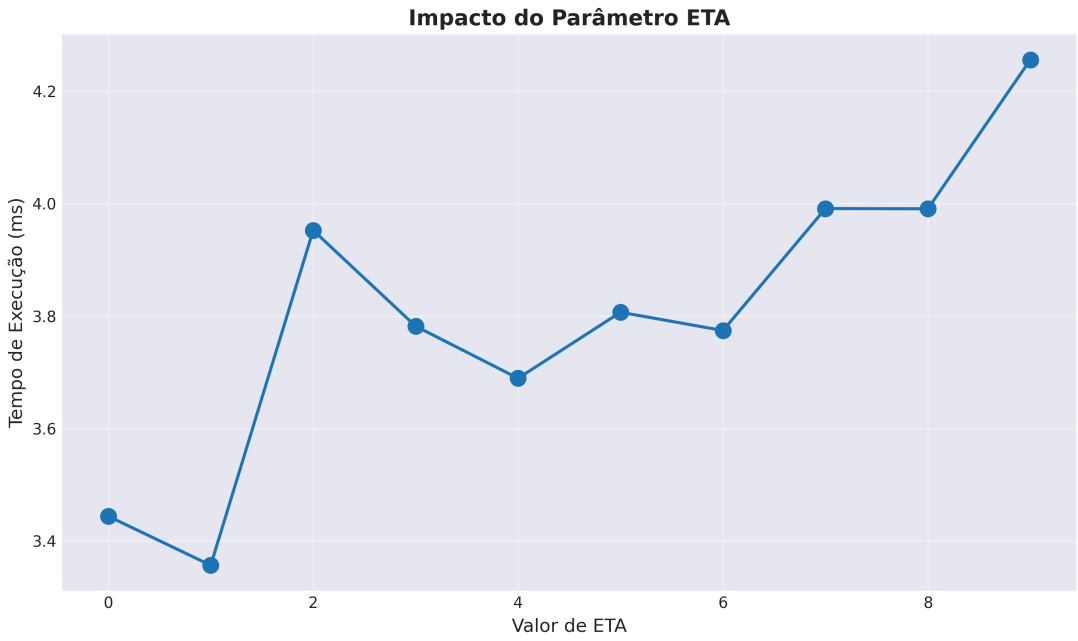
Com m = 200 demandas, variando λ de 0.5 a 0.9:




O tempo de execução aumenta conforme λ aumenta, pois aumentando o limite inferior da eficiência, o algoritmo descarta mais demandas durante a combinação dos grupos, realiza mais comparações e combina menos viagens.

**Teste 3: Variação de η (capacidade)**

Com m = 200 demandas, variando η de 0.0 a 10.0:




O tempo de execução aumenta conforme η aumenta, pois uma capacidade maior por carro aumenta, potencialmente, a quantidade de combinações de corridas, de modo que o algoritmo efetue mais comparações e demore mais.

## 5.5 Análise de Robustez

Testes com entradas adversárias:

1. **Demandas idênticas**: Sistema detectou corretamente e formou uma única corrida
2. **Demandas incompatíveis**: Todas foram processadas como corridas individuais
3. **Parâmetros extremos**: (α = 0, λ = 1.0): Sistema funcionou corretamente sem combinações
4. **Alta densidade temporal**: Todas as demandas dentro de δ foram corretamente consideradas

---

# 6. Conclusão

Este trabalho abordou o problema de despacho de corridas compartilhadas, implementando uma solução eficiente baseada em simulação de eventos discretos e algoritmo guloso para combinação de demandas. A abordagem adotada utiliza estruturas de dados especializadas (MinHeap para o escalonador e TADs personalizados) que garantem complexidade assintótica O(m × η × (η + log m)) para o processamento completo.

A principal abordagem desta solução inclui:

1. Algoritmo de combinação que garante conjuntos disjuntos de demandas, evitando sobreposições
2. Verificação eficiente de critérios múltiplos (temporal, espacial e de eficiência)
3. Implementação de MinHeap otimizado com redimensionamento automático

4. Cálculo de eficiência baseado na razão entre soma de distâncias individuais e distância da rota compartilhada

A análise experimental confirmou o comportamento teórico previsto, com tempos de execução e uso de memória aderentes às complexidades assintóticas calculadas. O sistema demonstrou robustez em casos extremos.

Por meio da resolução deste trabalho, foi possível praticar conceitos fundamentais de estruturas de dados, incluindo: heaps e filas de prioridade, simulação de eventos discretos, análise de complexidade assintótica, gerenciamento manual de memória em C++, e projeto de TADs coesos e modulares.

Os principais desafios enfrentados incluíram:

1. Garantir que demandas não fossem reatribuídas a múltiplas corridas (resolvido com array booleano de controle)
2. Interpretação correta do parâmetro λ de eficiência (resolvido ajustando a fórmula e comparação)
3. Ordem correta de visita às paradas na rota compartilhada (coletas antes de entregas)
4. Cálculo preciso de distâncias euclidianas evitando erros de arredondamento

# Referências


1. Lacerda, A., Santos, M., Meira Jr., W., Cunha, W. (2025). Apresentações de slides da disciplina de Estruturas de Dados (DCC205/DCC221).

2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

4. Ziviani, N. (2010). *Projeto de Algoritmos com Implementações em Pascal e C* (3ª ed.). Cengage Learning.