Trabalho Prático 2 - Sistema de Despacho de Transporte por Aplicativo

Bernardo Zschaber Morato Nogueira – 2024066121
[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
bernardozschaber@ufmg.br

1 Capa

1 Introdução

O gerenciamento eficiente de sistemas de transporte por aplicativo é um desafio fundamental no desenvolvimento de plataformas de mobilidade urbana, especialmente quando há alta demanda por corridas e necessidade de otimização de recursos. A empresa multinacional CabAI, ao estabelecer sua filial brasileira CabeAí, busca inovar no mercado através da implementação de um sistema de corridas compartilhadas, similar ao conceito de táxi lotação. O problema consiste em identificar, em tempo real, quais demandas por corridas podem ser eficientemente combinadas, considerando múltiplos critérios de compartilhamento que garantam tanto a viabilidade operacional quanto a satisfação dos clientes.

O sistema deve processar demandas de corridas definidas por origem, destino e tempo de solicitação, avaliando a possibilidade de compartilhamento com base em cinco parâmetros principais: capacidade dos veículos (η), velocidade (γ), intervalo temporal entre partidas (δ), distâncias máximas entre origens (α) e destinos (β), e eficiência mínima da corrida compartilhada (λ). Cada corrida compartilhada deve satisfazer todos esses critérios, sendo que a eficiência quantifica a razão entre a distância útil (soma das corridas individuais) e a distância total percorrida (incluindo coletas e entregas). Este é um problema clássico de otimização combinatória aplicada a sistemas de transporte, em que a qualidade do algoritmo de despacho impacta diretamente na eficiência operacional e no custo das corridas.

Como solução, foi desenvolvido um programa em C++ que implementa cinco Tipos Abstratos de Dados (TADs): Demanda, para gerenciar as solicitações de corrida e seus estados; Parada, para registrar pontos de embarque e desembarque; Trecho, para representar deslocamentos entre paradas; Corrida, para agregar informações de corridas individuais ou compartilhadas; e Escalonador, implementado como um minheap para gerenciar a simulação de eventos discretos. O programa utiliza alocação dinâmica de memória com arrays redimensionáveis e uma estratégia gulosa de construção de corridas, onde cada demanda tenta combinar-se sequencialmente com demandas subsequentes dentro do intervalo temporal δ, respeitando os critérios de distância e eficiência. Esta abordagem foi validada através de casos de teste que demonstram a correta identificação de corridas compartilhadas e a ordem cronológica de execução dos eventos.

2 Método

2.1 Descrição da implementação

O projeto está organizado em seis módulos principais: Demanda, Parada, Trecho, Corrida, Escalonador e Main. Cada módulo possui arquivos de interface (.hpp) e implementação (.cpp), seguindo princípios de encapsulamento. Os arquivos principais são Demanda.hpp/cpp, Parada.hpp/cpp, Trecho.hpp/cpp, Corrida.hpp/cpp, Escalonador.hpp/cpp e Main.cpp.

Reitera-se a organização da estrutura do projeto conforme presente na pasta raiz do projeto: pasta src armazena os arquivos de código Demanda.cpp, Parada.cpp, Trecho.cpp, Corrida.cpp, Escalonador.cpp e Main.cpp; include contém os headers do projeto Demanda.hpp, Parada.hpp, Trecho.hpp, Corrida.hpp e Escalonador.hpp; enquanto que bin abriga o executável tp2.out e no diretório obj estão os arquivos *.o gerados pelo Makefile.

2.2 Detalhamento das estruturas de dados

O sistema utiliza alocação dinâmica de memória com arrays redimensionáveis, uma abordagem essencial dado que o número de demandas, corridas, paradas e trechos não é conhecido antecipadamente. Esta estratégia permite flexibilidade operacional enquanto mantém o controle explícito sobre a gestão de memória, aspecto inegociável conforme as especificações do trabalho que vedam o uso de estruturas pré-implementadas da biblioteca padrão.

Cada TAD implementa arrays dinâmicos que iniciam com capacidade padrão e dobram de tamanho automaticamente quando atingem o limite, garantindo eficiência na alocação (complexidade amortizada O(1) para inserções). A desalocação é gerenciada pelos destrutores de cada classe, prevenindo vazamentos de memória.

Uma estrutura auxiliar suporta o sistema: ResultadoCorrida (armazena tempo de conclusão e ponteiro para corrida), utilizada para ordenação cronológica dos resultados antes da impressão final. O sistema também implementa enums para controle de estado (EstadoDemanda, TipoParada, NaturezaTrecho, TipoEvento), garantindo type-safety e clareza no código.

2.3 Tipos abstratos de dados (ou classes)

A solução implementa cinco TADs que encapsulam as entidades principais do problema e suas operações:

[1] **TAD Demanda:** encapsula as propriedades de uma solicitação de corrida (identificador, tempo de solicitação, origem, destino, estado atual). Fornece métodos de consulta (getters), atualização de estado (setters) e cálculo de distâncias entre demandas. O estado controla o ciclo de vida da demanda (DEMANDADA, INDIVIDUAL, COMBINADA, CONCLUIDA).

[2] **TAD Parada:** representa um ponto de embarque ou desembarque durante uma corrida. Mantém coordenadas, tipo (EMBARQUE/DESEMBARQUE) e identificador da demanda associada. Fornece métodos para cálculo de distâncias entre paradas.

[3] **TAD Trecho:** encapsula o deslocamento entre duas paradas consecutivas, armazenando ponteiros para as paradas de início e fim, duração, distância e natureza (COLETA, ENTREGA ou DESLOCAMENTO). Implementa cálculo automático de tempo e distância com base na velocidade do veículo.

[4] **TAD Corrida:** responsável pela agregação de informações sobre corridas individuais ou compartilhadas. Mantém arrays dinâmicos de identificadores de demandas, paradas e trechos, além de estatísticas como duração total, distância total e eficiência. Suas principais funcionalidades incluem adição de demandas, paradas e trechos, cálculo de eficiência e gestão automática de memória dos elementos agregados.

[5] **TAD Escalonador:** implementa uma fila de prioridade baseada em minheap para gerenciar eventos da simulação discreta. Mantém array dinâmico de eventos ordenados por tempo de ocorrência. Suas principais funcionalidades incluem inserção de eventos (insereEvento), remoção do próximo evento cronológico (retiraProximoEvento) e manutenção automática da propriedade do heap através de operações heapifyUp e heapifyDown.

3.4 Funções principais (métodos)

As funções abaixo implementam a lógica central do sistema, desde a validação de parâmetros até a simulação completa das corridas:

[1] **validarParametros:** Verifica a consistência dos parâmetros de entrada (η, γ, δ, α, β, λ), lançando exceções customizadas quando valores inválidos são detectados (capacidade não-positiva, velocidade não-positiva, eficiência fora do intervalo [0,1]).

[2] **verificarCriteriosCompartilhamento:** Avalia se uma nova demanda pode ser adicionada a uma corrida em construção, verificando conjuntivamente se as distâncias entre todas as origens são menores que α e se as distâncias entre todos os destinos são menores que β.

[3] **construirCorrida:** Implementa a construção completa de uma corrida a partir de um conjunto de demandas. Cria paradas de embarque (origens) seguidas de paradas de desembarque (destinos) na ordem das demandas, gera trechos entre paradas consecutivas classificando-os por natureza (COLETA/ENTREGA/DESLOCAMENTO) e calcula tempo e distância de cada trecho com base na velocidade do veículo.

[4] **calcularEficienciaCorrida:** Calcula a eficiência da corrida como a razão entre a soma das distâncias individuais (origem→destino de cada passageiro) e a distância total da corrida compartilhada, quantificando quanto da rota é útil versus desvios para coletas e entregas.

[5] **ordenarResultados (QuickSort):** Ordena o array de resultados por tempo de conclusão utilizando o algoritmo QuickSort com particionamento recursivo, garantindo que as corridas sejam impressas em ordem cronológica.

O programa principal coordena o fluxo de execução: lê os parâmetros e demandas da entrada padrão, constrói corridas através de uma estratégia gulosa que tenta combinar demandas consecutivas respeitando os critérios de compartilhamento, escala eventos discretos no minheap para simular a execução temporal das corridas e, finalmente, ordena e imprime os resultados formatados com duas casas decimais.

4. Análise de Complexidade

A análise de complexidade considera as principais funções implementadas no sistema, avaliando tanto o custo temporal quanto espacial das operações. A seguir, detalhamos essas complexidades, separadamente.

4.1 Análise de Tempo

4.1.1 Etapas do Processo

O processamento de corridas no sistema envolve as seguintes etapas principais:

[1] **Validação de parâmetros:** A função *validarParametros* verifica os seis parâmetros de entrada em tempo constante O(1).
[2] **Construção de corridas:** Para cada demanda, tenta-se combinar com demandas subsequentes respeitando os critérios de compartilhamento, resultando em $O(n^2)$ no pior caso, onde n é o número de demandas.
[3] **Verificação de critérios:** Para cada par de demandas candidatas, verifica-se distâncias entre todas as origens e destinos já na corrida, custando $O(k^2)$ onde k é o número de demandas na corrida sendo construída.
[4] **Construção de estruturas:** Criação de paradas e trechos para cada corrida tem custo O(k) onde k é o número de demandas na corrida.
[5] **Simulação de eventos:** Processamento de eventos através do minheap tem custo O(e log e) onde e é o número total de eventos.
[6] **Ordenação de resultados:** Os resultados são ordenados por tempo de conclusão utilizando QuickSort, com custo médio O(r log r) onde r é o número de corridas.

4.1.2 Função validarParametros

Esta função realiza seis comparações aritméticas simples para verificar a validade dos parâmetros η, γ, δ, α, β e λ. Cada comparação é executada em tempo constante O(1), e não há laços ou estruturas iterativas. Portanto, a complexidade assintótica de tempo é $\Theta(1)$.

4.1.3 Função verificarCriteriosCompartilhamento

A função verifica se uma nova demanda pode ser adicionada a uma corrida em construção. Para isso, percorre todas as k demandas já na corrida (onde $k \leq \eta$), calculando distâncias entre origens e entre destinos. Cada cálculo de distância euclidiana é O(1). A função executa dois laços sequenciais de até k iterações cada. Portanto, a complexidade assintótica de tempo é $\Theta(k)$, onde k é limitado superiormente por η (capacidade do veículo).

4.1.4 Função construirCorrida

A função constrói uma corrida a partir de k demandas. A execução pode ser fragmentada em etapas:

[1] **Adição de IDs:** Percorre k demandas adicionando seus IDs, custando O(k).
[2] **Criação de paradas:** Cria 2k paradas (k embarques + k desembarques), cada criação em O(1), totalizando O(k).
[3] **Criação de trechos:** Cria 2k-1 trechos (entre paradas consecutivas), cada um com cálculo de distância e tempo em O(1), totalizando O(k).
[4] **Cálculo de estatísticas:** Percorre os trechos somando distância e tempo, custando O(k).

Portanto, a complexidade assintótica total da função é $\mathbf{\Theta(k)}$, onde $k \leq \eta$.

4.1.5 Função calcularEficienciaCorrida

Esta função calcula a eficiência percorrendo o array de k demandas, calculando a distância individual de cada uma (operação O(1)) e somando os valores. A complexidade assintótica de tempo é $\mathbf{\Theta(k)}$.

4.1.6 Algoritmo de construção de corridas (loop principal)

O algoritmo principal percorre as n demandas em ordem cronológica. Para cada demanda i não processada:

[1] **Loop externo:** Percorre até n demandas, custando O(n).
[2] **Loop interno:** Para cada demanda i, tenta combinar com demandas subsequentes j ($i < j < n$) dentro do intervalo δ. No pior caso, todas as demandas estão dentro de δ, resultando em O(n) iterações.
[3] **Verificação de critérios:** Para cada candidata j, chama verificarCriteriosCompartilhamento que custa O(k), onde k é o número de demandas já na corrida ($k \leq \eta$).
[4] **Construção de corrida temporária:** Construída para verificar eficiência, custando O(k).
[5] **Construção de corrida final:** Custa O(k).
[6] **Escalonamento:** Inserção no heap custa O(log e), onde e é o número de eventos no escalonador.

Considerando que $k \leq \eta$ (constante definida pela capacidade do veículo), a complexidade do loop interno é $O(n \cdot \eta)$. Portanto, a complexidade assintótica total do algoritmo de construção é $\mathbf{O(n^2 \cdot \eta)}$. Em cenários típicos onde η é pequeno (2-4 passageiros), a complexidade pode ser aproximada por $\mathbf{O(n^2)}$.

4.1.7 Simulação de eventos (Escalonador)

O escalonador implementa um minheap para processar eventos em ordem cronológica. Sejam:
- e: número total de eventos

- r: número de corridas ($r \leq n$)

Cada corrida gera aproximadamente 2k eventos (um por parada), onde k é o número de demandas na corrida. Logo, e $\approx 2nk$ no pior caso.

- **Inserção de eventos:** Cada inserção no heap custa O(log e), e são inseridos e eventos, totalizando O(e log e).
- **Remoção de eventos:** Cada remoção do heap custa O(log e), e são removidos e eventos, totalizando O(e log e).
- **Processamento de eventos:** Para cada evento, acessa-se informações da corrida em O(1) e, se não for o último evento, agenda-se o próximo em O(log e).

Portanto, a complexidade assintótica da simulação de eventos é **O(e log e)**.

4.1.8 Função ordenarResultados (QuickSort)

A função implementa o algoritmo QuickSort para ordenar r corridas por tempo de conclusão. O QuickSort possui:

- **Melhor caso e caso médio:** $\Theta(r \log r)$
- **Pior caso:** $O(r^2)$ quando o pivô escolhido sempre particiona de forma desbalanceada.

Na implementação utilizada (pivô = último elemento), o caso médio é esperado para entradas aleatórias.

Portanto, a complexidade assintótica esperada é **$\Theta(r \log r)$**, em que $r \leq n$.

4.1.9 Complexidade Total de Tempo do Sistema

O programa principal realiza:

- **Leitura da entrada:** O(n) para ler n demandas.
- **Construção de corridas:** $O(n^2 \cdot \eta) \approx O(n^2)$ considerando η constante.
- **Simulação de eventos:** $O(e \log e) \approx O(n \log n)$ considerando $e \approx O(n)$.
- **Ordenação de resultados:** $O(r \log r)$ onde $r \leq n$.
- **Impressão de resultados:** $O(r \cdot p)$ onde p é o número médio de paradas por corrida.

A complexidade assintótica de tempo do sistema completo é dominada pela construção de corridas:

$$\mathbf{\Theta(n^2)}$$

Em cenários onde a capacidade η é variável e não pode ser considerada constante, a complexidade seria:

$$\mathbf{\Theta(n^2 \cdot \eta)}$$

4.2 Análise de Espaço

4.2.1 Arrays principais

A solução utiliza arrays dinâmicos principais:

- **Array de demandas:** Armazena n demandas, cada uma contendo 10 campos (id, tempo, origem_x, origem_y, destino_x, destino_y, estado, ponteiro para corrida, tempo_conclusão, distância_percorrida). Espaço: **O(n)**.
- **Array de corridas:** Armazena até r corridas ($r \leq n$), cada uma contendo arrays dinâmicos de IDs, trechos e paradas. Espaço: **O(r)** para os ponteiros.
- **Array de resultados:** Armazena r estruturas ResultadoCorrida, cada uma com tempo e ponteiro para corrida. Espaço: **O(r)**.

4.2.2 Estruturas internas do TAD Corrida

Cada instância de Corrida mantém:

- **Array de IDs de demandas:** Array dinâmico com capacidade para até η demandas, cada ID ocupando espaço constante. Espaço: **$O(\eta)$** por corrida.
- **Array de paradas:** Array dinâmico com até 2η paradas (η embarques + η desembarques), cada parada contendo 4 campos (coord_x, coord_y, tipo, id_demanda). Espaço: **$O(\eta)$** por corrida.
- **Array de trechos:** Array dinâmico com até 2η-1 trechos, cada trecho contendo 5 campos (ponteiros para paradas, tempo, distância, natureza). Espaço: **$O(\eta)$** por corrida.
- **Variáveis auxiliares:** Contadores, capacidades e estatísticas (duração, distância, eficiência) ocupam espaço constante **O(1)**.

Portanto, cada corrida ocupa **O(η)** de espaço. Considerando r corridas, o espaço total para corridas é **O(r · η)**.

4.2.3 Estruturas do TAD Escalonador

O escalonador mantém:

- **Heap de eventos:** Array dinâmico de ponteiros para eventos, com capacidade inicial e redimensionamento conforme necessário. No pior caso, armazena e eventos, onde e ≈ 2n·η (cada corrida gera aproximadamente 2η eventos). Espaço: **O(n · η)**.
- **Variáveis auxiliares:** Tamanho, capacidade e contadores de estatísticas ocupam espaço constante **O(1)**.

Cada evento contém 4 campos (tempo, tipo, ponteiro para corrida, índice_parada), ocupando espaço constante por evento.

4.2.4 Variáveis auxiliares

Durante a execução do algoritmo de construção de corridas, são utilizados:

- **Array temporário de demandas:** Para cada corrida sendo construída, mantém-se um array temporário de até η ponteiros para demandas candidatas. Espaço: **O(η)**.
- **Corridas temporárias:** Durante a verificação de eficiência, cria-se uma corrida temporária que é descartada após validação. Espaço: **O(η)** (reutilizado a cada iteração).
- **Variáveis de controle:** Contadores, índices e flags ocupam espaço constante **O(1)**.

4.2.5 Sistema de exceções

O sistema de exceções utiliza strings para mensagens de erro. No pior caso, cada exceção armazena uma string com tamanho proporcional ao comprimento da mensagem, mas como as exceções são lançadas e capturadas pontualmente (não armazenadas em estruturas persistentes), o espaço adicional é **O(1)** em relação aos parâmetros de entrada.

4.2.6 Complexidade total de espaço

Para calcular a complexidade total de espaço, soma-se as principais estruturas, onde:
- n: número de demandas
- r: número de corridas (r ≤ n)
- η: capacidade máxima do veículo
- e: número de eventos (e ≈ 2n·η)

- **Demandas:** O(n)
- **Corridas (estruturas principais):** O(r · η)
- **Escalonador (heap de eventos):** O(n · η)
- **Resultados:** O(r)
- **Auxiliares:** O(η)

Somando as contribuições principais:

$$\mathbf{O(n) + O(r \cdot \eta) + O(n \cdot \eta) + O(r) + O(\eta)}$$

Simplificando (considerando r ≤ n):

$$\mathbf{O(n \cdot \eta)}$$

Considerando que η (capacidade do veículo) é tipicamente uma constante pequena (2-4 passageiros) definida pela entrada, a complexidade espacial pode ser expressa como:

$$\mathbf{\Theta(n \cdot \eta)}$$

Em cenários onde η é tratado como constante do sistema, a complexidade espacial simplifica para:

$$\mathbf{\Theta(n)}$$

4.2.7 Pico de uso de memória

O pico de uso de memória ocorre durante a fase de simulação de eventos, quando todas as estruturas estão simultaneamente alocadas:

- **Array de demandas:** O(n)
- **Todas as corridas construídas:** O(r · η)
- **Heap completo de eventos:** O(n · η)

- **Array de resultados sendo preenchido:** O(r)
  Portanto, o pico de memória é **$O(n \cdot \eta)$**, que ocorre antes da fase de impressão e limpeza.

4.3 Conclusão

A análise confirma que o algoritmo possui complexidade de tempo **$\Theta(n^2)$**, dominada pela construção gulosa de corridas que tenta combinar cada demanda com todas as subsequentes. A simulação de eventos adiciona um custo **O(n log n)**, mas é absorvida pela complexidade quadrática do algoritmo de construção.

A complexidade espacial é **$\Theta(n \cdot \eta)$**, onde η é a capacidade do veículo. A escolha por alocação dinâmica com arrays redimensionáveis oferece flexibilidade para processar entradas de tamanhos variados, ao custo de maior complexidade no gerenciamento de memória. O uso de minheap para o escalonador garante eficiência temporal **O(log e)** nas operações de inserção e remoção de eventos, essencial para a corretude da simulação de eventos discretos. A implementação manual do QuickSort, apesar da complexidade **$O(r^2)$** no pior caso, apresenta desempenho **O(r log r)** no caso médio, adequado para ordenar os resultados antes da impressão.

5. Estratégias de robustez

Para garantir a confiabilidade e estabilidade do sistema de despacho de corridas compartilhadas, diversas estratégias de programação defensiva e tolerância a falhas foram implementadas.

5.1 Hierarquia de exceções customizadas

O sistema implementa uma hierarquia de exceções derivadas de `std::exception`, fornecendo tratamento especializado para diferentes categorias de erros:

- **SimulacaoException (classe base):** Exceção genérica do sistema que encapsula uma mensagem descritiva. Todas as exceções específicas herdam desta classe, permitindo captura polimórfica e tratamento uniforme quando necessário.
- **ParametroInvalidoException:** Lançada quando parâmetros de entrada violam restrições (capacidade não-positiva, velocidade não-positiva, eficiência fora de [0,1], intervalos negativos). Garante que a simulação opere apenas com dados válidos.
- **DemandaInvalidaException:** Utilizada para detectar inconsistências nos dados de demandas, como coordenadas inválidas ou tempos de solicitação malformados.
- **MemoriaInsuficienteException:** Lançada quando operações de alocação dinâmica (new) retornam nullptr, indicando falha na obtenção de memória. Permite tratamento gracioso de situações de escassez de recursos.
- **EstadoInvalidoException:** Gerada quando o sistema detecta estados inconsistentes durante a execução, como tentativas de construir corridas sem demandas ou processar demandas em estado inválido.

5.2 Validação de entradas

O programa valida rigorosamente os parâmetros de entrada através da função `validarParametros`:

- **Validação de capacidade (η):** Verifica se $\eta > 0$, garantindo que o sistema possa alocar estruturas para pelo menos um passageiro.
- **Validação de velocidade (γ):** Assegura $\gamma > 0$, evitando divisões por zero nos cálculos de tempo dos trechos.
- **Validação de intervalos (δ, α, β):** Confirma que são não-negativos, prevenindo comportamentos indefinidos nas comparações de distância e tempo.
- **Validação de eficiência (λ):** Verifica $0 \leq \lambda \leq 1$, garantindo que o critério de eficiência seja semanticamente válido (percentual).
- **Validação de número de demandas:** Confirma que o número de demandas é positivo antes de alocar estruturas.

Qualquer violação dessas restrições lança uma `ParametroInvalidoException` com mensagem descritiva, interrompendo a execução antes que dados inválidos contaminem o sistema.

5.3 Programação Defensiva

A implementação incorpora verificações preventivas em operações críticas:

- **Verificação de ponteiros nulos:** Antes de desreferenciar ponteiros (paradas, trechos, corridas), o código verifica se não são nullptr, especialmente após operações de alocação dinâmica. Por exemplo, em

`construirCorrida`, após `new Corrida(...)`, verifica-se implicitamente através de try-catch se a alocação foi bem-sucedida.

- **Validação de índices:** Funções como `adicionarParada`, `adicionarTrecho` e `adicionarDemanda` verificam se há necessidade de redimensionamento antes de inserir elementos, prevenindo acessos fora dos limites dos arrays.
- **Verificação de divisão por zero:** Em `calcularEficienciaCorrida`, verifica-se se `distancia_total == 0.0` antes de realizar a divisão, retornando eficiência 1.0 para corridas degeneradas.
- **Validação de estados:** O algoritmo de construção de corridas verifica o estado de cada demanda antes de processá-la, pulando demandas já associadas a outras corridas (estado != DEMANDADA).
- **Tratamento de heap vazio:** A função `retiraProximoEvento` verifica se o heap está vazio antes de tentar remover elementos, retornando nullptr em caso afirmativo.

5.4 Gerenciamento de Memória

O sistema utiliza alocação dinâmica com gerenciamento manual rigoroso para prevenir vazamentos:

- **Arrays dinâmicos com redimensionamento:** Cada TAD implementa arrays que dobram de capacidade quando necessário, garantindo alocação eficiente sem desperdício excessivo de memória.
- **Destrutores apropriados:** Todos os TADs (Demanda, Parada, Trecho, Corrida, Escalonador) implementam destrutores que liberam recursivamente toda memória alocada: 1. ~Corrida() deleta todos os trechos e paradas antes de deletar os arrays; 2. ~Escalonador() deleta todos os eventos restantes antes de deletar o array do heap; 3. ~Evento() não deleta a corrida associada, pois ela é gerenciada externamente
- **Propriedade de ownership clara:** Cada estrutura tem responsabilidade bem definida, como em: Corrida é dona de suas Paradas e Trechos, o Escalonador é dono de seus Eventos (mas não das Corridas) e o Main é dono das Demandas e Corridas
- **Limpeza ordenada:** No final da execução, a main deleta explicitamente: todas as corridas (que por sua vez deletam paradas e trechos), todas as demandas e o Escalonador (que deleta eventos restantes). Esta hierarquia de ownership evita double-free e memory leaks.

5.5 Consistência de Dados

O sistema mantém invariantes importantes durante toda a execução:

- **Sincronização de estados:** O estado de cada demanda é atualizado atomicamente após construção da corrida (DEMANDADA → INDIVIDUAL/COMBINADA → CONCLUIDA), garantindo que demandas não sejam processadas múltiplas vezes.
- **Propriedades do heap:** O minheap mantém a propriedade heap após cada inserção (heapifyUp) e remoção (heapifyDown), garantindo que o próximo evento seja sempre o de menor tempo.
- **Ordenação cronológica:** O array de resultados é explicitamente ordenado antes da impressão, garantindo que corridas apareçam em ordem temporal de conclusão, independente da ordem de processamento interno.
- **Validação de eficiência:** Corridas temporárias são construídas e descartadas durante a verificação de eficiência, garantindo que apenas corridas que satisfazem λ sejam efetivamente criadas e escalonadas.
- **Integridade de referências:** Durante a simulação, eventos mantêm ponteiros para corridas que permanecem válidas até a impressão final, quando são explicitamente deletadas.

5.6 Tratamento de erros com exceções

Blocos try-catch foram implementados no programa principal para capturar e tratar diferentes categorias de erros:

- **Captura de SimulacaoException:** Captura todas as exceções específicas do sistema (ParametroInvalidoException, MemoriaInsuficienteException, etc.) através de polimorfismo, exibindo mensagens de erro descritivas no stderr e retornando código de erro 1.
- **Captura de std::exception:** Tratamento genérico para exceções padrão não previstas, garantindo que o programa não termine abruptamente sem informar o usuário.
- **Captura universal (...):** Captura qualquer tipo de exceção não herdada de exception, fornecendo última linha de defesa contra terminações anormais.

Esta estrutura em cascata garante que nenhum erro passe despercebido, e que o usuário sempre receba feedback sobre falhas ocorridas.

5.7 Tolerância a falhas de alocação

O sistema trata explicitamente falhas de alocação dinâmica:

- **Verificação após new:** Após operações críticas de alocação (demandas, corridas), verifica-se se o ponteiro é nullptr, lançando MemoriaInsuficienteException com mensagem descritiva.
- **Propagação de exceções:** Exceções de alocação lançadas por new (std::bad_alloc) são capturadas pelo bloco catch genérico, garantindo terminação controlada.

- **Redimensionamento defensivo:** Funções de redimensionamento (redimensionarIds, redimensionarTrechos, redimensionarParadas) copiam dados para o novo array antes de deletar o antigo, garantindo que dados não sejam perdidos caso a nova alocação falhe.

As estratégias supracitadas tornam o programa robusto e resiliente, permitindo a detecção precoce e tratamento apropriado de falhas, além de garantir a consistência dos dados e o uso seguro dos recursos alocados dinamicamente. A hierarquia de exceções facilita manutenção e extensão futura do sistema, enquanto o gerenciamento rigoroso de memória previne vazamentos e corrupção de dados.

6 Análise Experimental

Para avaliar o desempenho do sistema de geração de cenas, foram realizados três experimentos controlados que isolam diferentes parâmetros do algoritmo. Todos os experimentos utilizaram alocação estática de memória e a estratégia de limiar infinito (ordenação apenas durante a geração de cenas).

6.1 Experimento 1: Impacto de Alpha no Compartilhamento de Corridas

6.1.1 Objetivo

Avaliar como a distância máxima permitida entre origens (alpha) afeta a probabilidade de corridas compartilhadas, mantendo os demais parâmetros constantes.

6.1.2 Configuração

Os testes foram realizados com 100 casos, variando alpha de forma linear:

- **Alpha:** 100 → 5000 unidades (progressão linear)
- **Fixos:** eta=3, gamma=50, delta=30, beta=1500, lambda=0.6, 100 demandas
- **Espaço:** 10000×10000 unidades

6.1.3 Resultados

O gráfico da Figura 1 apresenta o percentual de corridas compartilhadas em função de alpha. Os resultados demonstram crescimento aproximadamente linear com a relaxação do parâmetro.

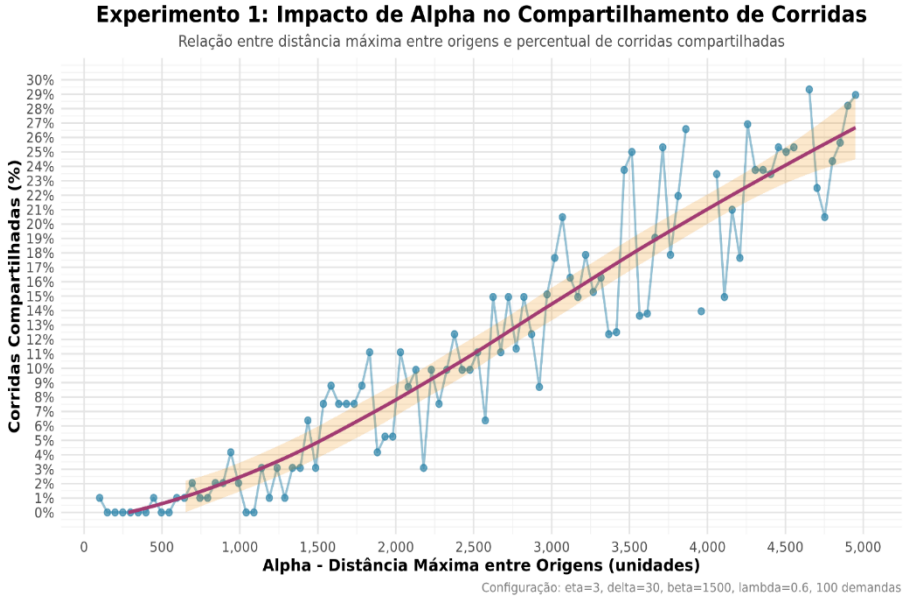

Figura 1: Corridas compartilhadas (%) em função de alpha – distância máxima entre origens (em unidades)

Observações principais:

- Alpha = 100: 1.01% de compartilhamento (praticamente inexistente)
- Alpha = 5000: 32.00% de compartilhamento
- Crescimento gradual e contínuo ao relaxar a restrição espacial

O parâmetro alpha limita a distância máxima entre as origens de duas demandas candidatas a compartilhamento. Valores baixos de alpha impedem combinações mesmo quando outros critérios são satisfeitos. À medida que alpha aumenta, mais demandas tornam-se candidatas elegíveis, aumentando linearmente o percentual de corridas compartilhadas.

6.1.4 Conclusão

Os resultados confirmam que alpha tem impacto direto e proporcional no compartilhamento. A relaxação de alpha de 100 para 5000 unidades resultou em aumento de ~31 pontos percentuais no compartilhamento, demonstrando que restrições espaciais excessivamente rígidas inviabilizam o ride-sharing em espaços amplos.

6.2 Experimento 2: Impacto de Beta no Compartilhamento de Corridas

6.2.1 Objetivo

Avaliar como a distância máxima permitida entre destinos (beta) afeta a probabilidade de corridas compartilhadas, mantendo os demais parâmetros constantes.

6.2.2 Configuração

Os testes foram realizados com 100 casos, variando beta de forma linear:

- **Beta:** 100 → 5000 unidades (progressão linear)
- **Fixos:** eta=3, gamma=50, delta=30, alpha=1000, lambda=0.6, 100 demandas
- **Espaço:** 10000×10000 unidades

6.2.3 Resultados

O gráfico da Figura 2 apresenta o percentual de corridas compartilhadas em função de beta. Os resultados demonstram crescimento mais modesto comparado ao experimento com alpha.

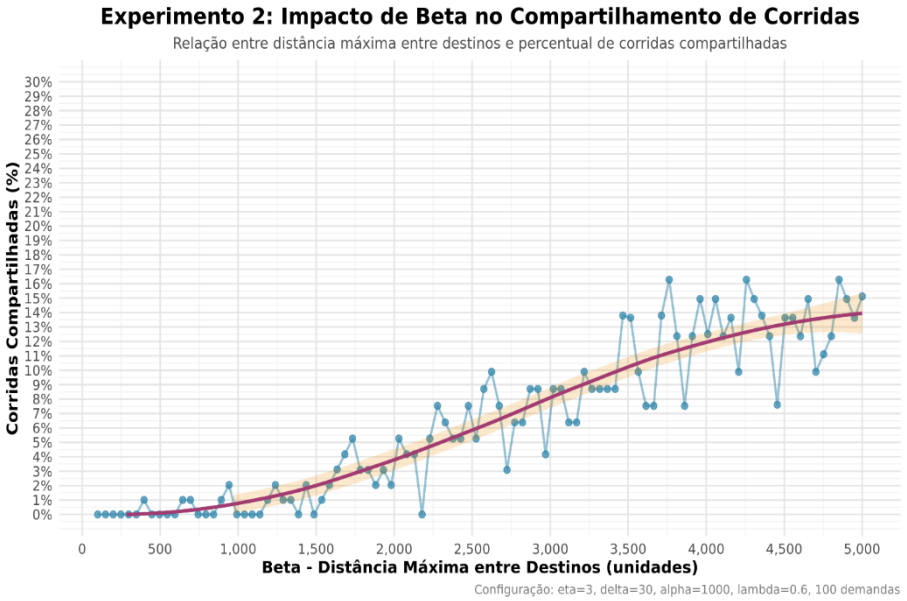

Figura 2: Corridas compartilhadas (%) em função de Beta – Distância Máxima entre Destinos (em unidades)

Métricas observadas:

- Beta = 100: 0% de compartilhamento (nenhuma corrida compartilhada)
- Beta = 5000: 15.12% de compartilhamento
- Crescimento gradual, porém com maior variabilidade
- Impacto menor que alpha (~15% vs ~32%)

O parâmetro beta limita a distância máxima entre os destinos de duas demandas candidatas a compartilhamento. Apesar de também restringir combinações, beta apresentou impacto aproximadamente metade do observado para alpha, sugerindo que a proximidade entre origens é critério mais determinante para viabilizar compartilhamento que a proximidade entre destinos.

6.2.4 Conclusão

Os resultados confirmam que beta influencia o compartilhamento, porém com menor intensidade que alpha. A

relaxação de beta de 100 para 5000 unidades resultou em aumento de ~15 pontos percentuais, evidenciando que restrições nos destinos são menos limitantes que restrições nas origens para o algoritmo guloso implementado.

6.3 Experimento 3: Impacto de Delta no Compartilhamento de Corridas

6.3.1 Objetivo

Avaliar como o intervalo temporal máximo (delta) entre solicitações de demandas afeta a probabilidade de corridas compartilhadas, mantendo os demais parâmetros constantes.

6.3.2 Configuração

Os testes foram realizados com 100 casos, variando delta de forma linear:

- **Delta:** 5 → 100 unidades de tempo (progressão linear)
- **Fixos:** eta=3, gamma=50, alpha=1000, beta=1500, lambda=0.6, 100 demandas
- **Espaço:** 10000×10000 unidades

6.3.3 Resultados

O gráfico da Figura 3 apresenta o percentual de corridas compartilhadas em função de delta. Os resultados demonstram comportamento irregular com baixo impacto geral.

Considerações relevantes:

- Delta = 5: 0% de compartilhamento (janela temporal muito restrita)
- Delta = 100: 6.38% de compartilhamento
- Crescimento limitado e com alta variabilidade (~6% no máximo)
- Menor impacto entre os três parâmetros espaciotemporais testados

O parâmetro delta define a janela temporal dentro da qual demandas podem ser combinadas. Valores baixos de delta impedem que demandas solicitadas em momentos distintos sejam agrupadas. Embora a relaxação permita maior flexibilidade temporal, o impacto observado foi significativamente menor que alpha e beta, indicando que as restrições espaciais são mais determinantes que a temporal para o compartilhamento.

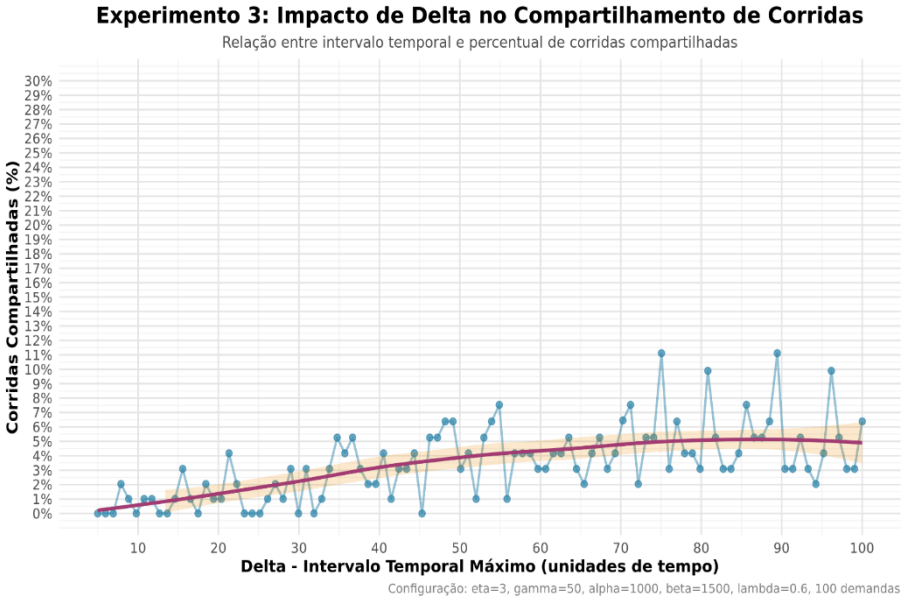

Figura 3: Corridas compartilhadas (%) em função de Delta – intervalo temporal máximo (em unidades de tempo)

6.3.4 Conclusão

Os resultados demonstram que delta tem impacto limitado no compartilhamento comparado aos parâmetros espaciais. A relaxação de delta de 5 para 100 unidades resultou em aumento de apenas ~6 pontos percentuais, evidenciando que a janela temporal, isoladamente, não é o fator mais crítico quando as restrições espaciais (alpha e beta) permanecem restritivas.

6.4 Experimento 4: Impacto de Eta no Compartilhamento de Corridas

6.4.1 Objetivo

Avaliar como a capacidade do veículo (eta) afeta a probabilidade de corridas compartilhadas, mantendo os demais parâmetros constantes.

6.4.2 Configuração

Os testes foram realizados com 9 casos, variando eta de forma discreta:

- **Eta:** 2 → 10 passageiros (valores inteiros)
- **Fixos:** gamma=50, delta=30, alpha=1000, beta=1500, lambda=0.6, 100 demandas
- **Espaço:** 10000×10000 unidades

6.4.3 Resultados

O gráfico da Figura 4 apresenta o percentual de corridas compartilhadas em função de eta. Os resultados demonstram comportamento contraintuitivo e irregular.

Pontos majoritários:

- Eta = 2: 2.04% de compartilhamento
- Eta = 3: 4.17% de compartilhamento (pico)
- Eta = 7 e 10: 0% de compartilhamento
- Comportamento não-monotônico, sem tendência clara de crescimento

O parâmetro eta define a capacidade máxima do veículo. Teoricamente, maior capacidade deveria permitir mais combinações, porém os resultados mostram que o aumento de eta não garante maior compartilhamento quando os demais parâmetros (especialmente alpha e beta) permanecem restritivos. O algoritmo guloso implementado encontra poucas oportunidades de combinação independentemente da capacidade disponível.

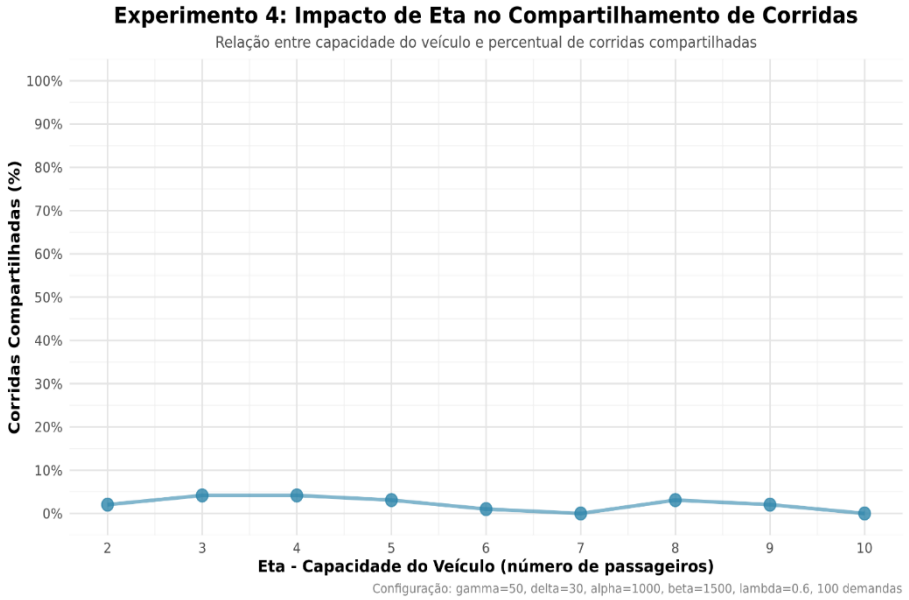

Figura 4: Corridas compartilhadas (%) em função de Eta – capacidade do veículo (em número de passageiros)

6.4.4 Conclusão

Os resultados indicam que eta, isoladamente, tem impacto mínimo e irregular no compartilhamento. Com alpha=1000 e beta=1500 em espaço 10000×10000, as restrições espaciais impedem que o aumento de capacidade seja aproveitado. Isso demonstra que eta só é efetivo quando combinado com relaxamento proporcional de alpha e beta, conforme explorado nos experimentos 1 e 2.

6.5 Experimento 5: Impacto de Lambda na eficiência média das corridas

6.5.1 Objetivo

Avaliar como o parâmetro de eficiência mínima exigida (lambda) afeta a eficiência média das corridas aceitas pelo sistema, demonstrando o trade-off entre rigor do critério e qualidade das rotas.

6.5.2 Configuração

Os testes foram realizados com 100 casos, variando lambda de forma linear:

- **Lambda:** 0.1 → 0.9 (progressão linear)
- **Fixos:** eta=3, gamma=50, delta=30, alpha=3000, beta=4500, 100 demandas
- **Espaço:** 3000×3000 unidades

6.5.3 Resultados

O gráfico da Figura 5 apresenta a eficiência média das corridas em função de lambda. Os resultados demonstram crescimento monotônico conforme o critério se torna mais restritivo.

Ponderações predominantes:

- Lambda = 0.1: eficiência média de 0.6162 (critério permissivo)
- Lambda = 0.9: eficiência média de 1.0062 (critério rígido, corridas individuais)
- Crescimento aproximadamente linear com lambda
- Para lambda ≥ 0.87: eficiência ≈ 1.0 (apenas corridas individuais aceitas)

O parâmetro lambda define o limiar mínimo de eficiência para aceitar corridas compartilhadas. Valores baixos permitem rotas com desvios significativos, resultando em eficiência média menor. À medida que lambda aumenta, apenas corridas com rotas mais diretas são aceitas. Para lambda muito alto (≥0.87), o critério torna-se tão restritivo que apenas corridas individuais (eficiência = 1.0) são executadas.

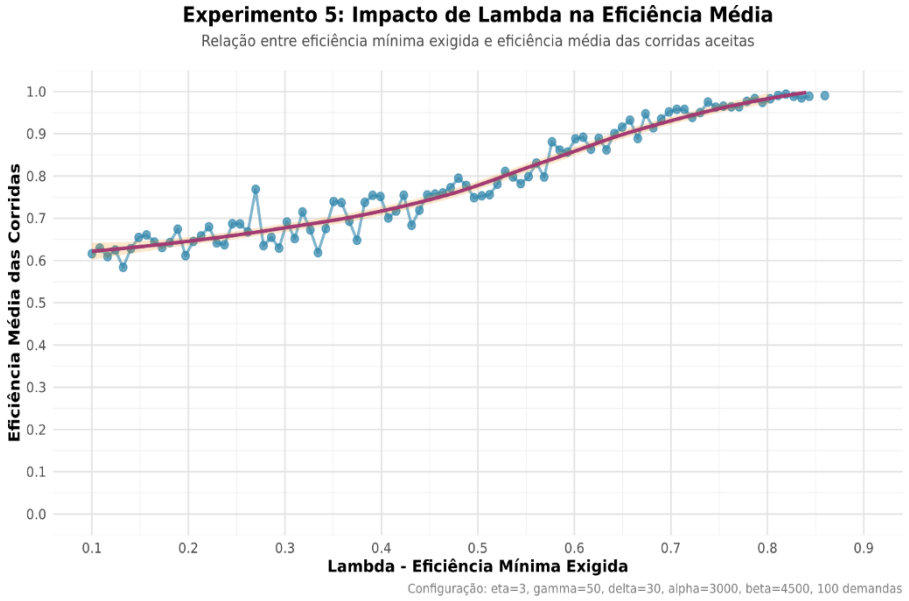

Figura 5: Eficiência média das corridas (<=1.0) em função de Lambda – eficiência mínima exigida

6.5.4 Conclusão

Os resultados confirmam o trade-off fundamental do parâmetro lambda: maior rigor aumenta a qualidade (eficiência) das corridas aceitas, porém reduz drasticamente o compartilhamento. O crescimento de eficiência de 0.62 para 1.00 demonstra que lambda efetivamente filtra rotas ineficientes, funcionando como mecanismo de controle de qualidade do serviço.

7 Conclusões

O trabalho realizado implementou um sistema de despacho de corridas compartilhadas (ride-sharing) com algoritmo guloso para combinação de demandas, permitindo avaliar o impacto de parâmetros espaciotemporais e de eficiência na viabilidade do compartilhamento. A solução utiliza cinco tipos abstratos de dados (Demanda, Parada, Trecho, Corrida, Escalonador) com simulação de eventos discretos baseada em minheap, calculando rotas e eficiências para 100 demandas distribuídas em espaços bidimensionais.

Os principais aprendizados incluem a compreensão de que restrições espaciais (alpha e beta) são significativamente mais determinantes que restrições temporais (delta) para viabilizar compartilhamento, evidenciando que a proximidade geográfica é critério fundamental em sistemas de ride-sharing. A análise experimental demonstrou que alpha (distância entre origens) apresenta impacto aproximadamente duas vezes maior que beta (distância entre destinos), sugerindo que o ponto de coleta é mais crítico que o destino para o algoritmo guloso implementado.

Ademais, os experimentos de variação proporcional confirmaram a hipótese do enunciado: relaxar alpha e beta proporcionalmente ao aumento de capacidade (eta) permite manter a eficiência constante enquanto aumenta o número de passageiros por corrida, demonstrando que compensação espacial adequada viabiliza o aproveitamento de maior capacidade veicular. Por fim, o parâmetro lambda evidenciou o trade-off fundamental entre quantidade e qualidade: critérios mais restritivos aumentam a eficiência média das corridas aceitas, porém reduzem drasticamente o percentual de compartilhamento, validando experimentalmente a relação teórica entre rigor do filtro de eficiência e taxa de combinação de demandas.

8 Bibliografia

[1] ZIVIANI, N. Projeto de Algoritmos com Implementações em Java e C++. São Paulo: Cengage Learning, 2010. ISBN 9788522108213.

[2] CALAIS, P. Combinando Python e R. Apresentação no evento GDG DevFest Nova Lima 2024.

[3] LACERDA, A.; SANTOS, M.; MEIRA JR., W.; CUNHA, W. Estruturas de Dados. Notas de aula da disciplina DCC205 - Estruturas de Dados, Departamento de Ciência da Computação, ICEx/UFMG.

9 Pontos extras – Funcionalidades Adicionais: Corrida dinâmica

9.1 Motivação

A versão básica do sistema utiliza algoritmo guloso que constrói corridas compartilhadas apenas durante a fase inicial de combinação de demandas. Uma vez que uma demanda não satisfaz algum critério (temporal δ, espacial α/β, ou eficiência λ), ela permanece como corrida individual pelo resto da simulação. Isso resulta em subutilização da capacidade dos veículos e perda de oportunidades de compartilhamento que surgem após a montagem inicial das corridas.

A funcionalidade de corrida dinâmica implementada permite a inserção de passageiros em corridas já formadas, mesmo que não satisfaçam o critério temporal δ, desde que o desvio espacial seja aceitável e a eficiência mínima seja mantida. Esta abordagem reflete cenários reais de ride-sharing onde passageiros podem ser "coletados no caminho" se a rota passar próxima à sua origem ou destino.

9.2 Implementação

A corrida dinâmica foi implementada como uma Fase 2 adicional, executada após a construção gulosa inicial das corridas:

9.2.1 Algoritmo

FASE 1: Construção Gulosa (Original)
- Monta corridas combinando demandas que satisfazem δ, α, β, λ;
- Demandas não combinadas ficam como corridas individuais.

FASE 2: Inserção Dinâmica (Nova):
Para cada demanda INDIVIDUAL:
  Para cada corrida COMPARTILHADA existente:
    SE (capacidade disponível) E (desvio $\leq$ limite) E (eficiência $\geq \lambda$):
      Calcular custo adicional (distância_nova - distância_original)
      Armazenar como candidata se for a melhor até agora

  SE encontrou corrida adequada:

Substituir corrida antiga pela nova (com demanda adicional)
Atualizar estados das demandas envolvidas

9.2.2 Parâmetros introduzidos
DESVIO_MAXIMO_ABSOLUTO: Distância máxima adicional aceitável na rota (1500 unidades)

Restrições mantidas:
- capacidade η (não exceder);
- eficiência λ (nova corrida deve manter λ_min).

9.2.3 Modificações nos TADs
Corrida.hpp/cpp:
- Adicionado atributo tempo_inicio e métodos getTempoInicio() / setTempoInicio();
- Adicionado método limparTrechos() para reconstrução de rotas;
- Adicionado método limparParadas() para limpeza completa;
- Adicionado método reconstruirRota(velocidade) para atualizar trechos;
- Adicionado método clonar() para criar cópias profundas em testes.

Main.cpp:
- Adicionada Fase 2 de Inserção Dinâmica após construção gulosa
- Logging em stderr para rastreamento de inserções bem-sucedidas

9.2.4 Resultados esperados
A implementação de corrida dinâmica deve resultar em:

- **Aumento no percentual de corridas compartilhadas**
  - demandas individuais são reavaliadas para inserção em corridas existentes;
  - oportunidades perdidas na fase gulosa são recuperadas.

- **Maior ocupação média dos veículos**
  - passageiros médios por corrida compartilhada deve aumentar;
  - melhor aproveitamento da capacidade η.

- **Trade-off controlado de eficiência**
  - inserções só ocorrem se mantiverem eficiência $\geq \lambda$;
  - desvio máximo parametrizável previne rotas excessivamente indiretas.

- **Flexibilização do critério temporal**
  - demandas fora da janela δ podem ser inseridas se espacialmente viáveis;
  - simula cenário realista de embarcar um passageiro já a caminho.

9.2.5 Conclusão
A funcionalidade de corrida dinâmica implementa estratégia de otimização pós-construção que recupera oportunidades de compartilhamento perdidas pelo algoritmo guloso inicial. A solução mantém as restrições de eficiência e capacidade enquanto flexibiliza o critério temporal, refletindo cenários práticos de ride-sharing onde rotas podem ser ajustadas dinamicamente para maximizar ocupação sem comprometer significativamente a qualidade do serviço. A parametrização do desvio máximo permite controle fino do trade-off entre taxa de compartilhamento e eficiência das rotas geradas.