Trabalho Prático 2: Sistema de Despacho de Transporte por Aplicativo

Victor Murilo Kaizer Meireles
Matrícula: 2024096551

Novembro de 2025

1 Introdução

A otimização de recursos é um desafio fundamental na logística de transporte urbano. Para reduzir custos operacionais e melhorar a eficiência de serviços de mobilidade, torna-se essencial desenvolver algoritmos que maximizem a ocupação dos veículos. Nesse cenário, a empresa multinacional CabAI, através de sua filial brasileira "Cabe Aí" deseja implementar um sistema de despacho de corridas compartilhadas (*ride-sharing*). Este Documento apresenta o desenvolvimento de um simulador capaz de gerenciar essas demandas, utilizando a técnica de Simulação de Eventos Discretos (SED) para modelar a operação da frota ao longo do tempo.

O algoritmo desenvolvido neste trabalho tem como objetivo processar uma sequência cronológica de solicitações de transporte e decidir, em tempo real, se uma demanda deve ser atendida individualmente ou se pode ser agrupada com outras em uma única rota. O sistema lê os parâmetros de configuração da frota e a lista de solicitações dos clientes, verifica a compatibilidade entre passageiros e simula a execução das corridas, gerando ao final estatísticas detalhadas de desempenho, como tempo total e distância percorrida.

De forma sucinta, a solução implementada baseia-se em uma lógica de emparelhamento. A cada nova solicitação, o sistema investiga potenciais compartilhamentos dentro de uma janela de tempo pré-definida (δ). A viabilidade do agrupamento é determinada pela satisfação simultânea de vários critérios: a capacidade máxima do veículo (η), a proximidade geográfica entre as origens (α) e os destinos (β) dos passageiros, e a garantia de uma eficiência mínima (λ) para a rota combinada em comparação à corrida individual. Para o controle temporal da simulação, foi implementado um escalonador baseado em uma fila de prioridade (*MinHeap*), responsável por ordenar e executar cronologicamente os eventos de embarque e desembarque.

Esta documentação detalha a implementação técnica na seção Métodos, seguida pela avaliação de eficiência computacional na Análise de Complexidade. São apresentados também os mecanismos de segurança do código em Estratégias de Robustez. Posteriormente, discute-se o comportamento do sistema sob diferentes cenários na Análise Experimental, encerrando com as considerações finais na Conclusão.

2 Método

O sistema foi desenvolvido em C++, utilizando o paradigma de Programação Orientada a Objetos para modelar as entidades da corrida compartilhada e encapsular a lógica de simulação. A abordagem inicial baseia-se na leitura dos parâmetros globais de configuração definidos no arquivo de entrada: capacidade do veiculo (η), velocidade média (γ), janela temporal máxima (δ), distâncias máximas para agrupamento de origens (α) e destinos (β), e a eficiência mínima exigida (λ).

2.1 Tipos Abstratos de Dados

Para organizar os elementos da cena e os resultados do processamento, foram definidos Tipos Abstratos de Dados (TADs) específicos:

- **Demanda (Demand)**: Representa a solicitação de transporte de um cliente.
  - **Atributos**: Armazena o identificador único (**id**), o instante da solicitação (**requestTime**), as coordenadas geográficas de origem e destino (**Origin, Destination**), o estado atual da demanda (**REQUESTED, INDIVIDUAL, COMBINED e COMPLETED**) e um ponteiro para a corrida associada (**associatedRide**)
  - **Métodos**: Disponibiliza o construtor para inicialização dos dados brutos lidos do arquivo e métodos de acesso (*Getters*) para todos os atributos. Além disso, possui métodos modificadores (*Setters*) para alterar o estado da demanda (ex: de `REQUESTED` para `COMBINED`) e vincular a corrida responsável pelo atendimento.
- **Parada (Stop)**: Abstração de um ponto geográfico onde ocorre uma ação do veículo
  - **Atributos**: Contém as coordenadas geográficas (**Coordinates**), o tipo de parada (**PICKUP ou DROPOFF**) e um ponteiro para a demanda (**demand**) referente ao passageiro que está embarcando ou desembarcando.
  - **Métodos**: Possui um construtor que associa as coordenadas e o tipo à demanda, além de metodos de acesso (*Getters*) para a recuperação dessas informações durante a construção de trechos

- **Trecho (Segment)**: Representa o deslocamento do veículo entre duas paradas consecutivas.
  - **Atributos**: Armazena ponteiros para a parada inicial e final (**startStop, endStop**), o tempo calculado para percorrer o trajeto (**time**), a distância euclidiana (**distance**) e a natureza do trecho, que pode ser de Coleta (`COLLECTION`), Entrega (`DELIVERY`) ou Deslocamento (`DISPLACEMENT`).
  - **Métodos**: Implementa um construtor que inicializa todos os dados calculados e métodos de acesso (*Getters*). Diferente de outras estruturas, este TAD não possui destrutor, delegando a limpeza de memória das paradas para a classe `Ride`.
- **Corrida (Ride)**: A estrutura central que agrupa passageiros e define a rota
  - **Atributos**: Gerencia dois vetores dinâmicos manuais: um para as demandas (**demands**) e outro para os trechos (**segments**). Mantém também contadores de capacidade e tamanho atual (**count, capacity**) para ambos os vetores, além de estatísticas acumuladas como distância total, duração total e eficiência (**totalDistance, totalDuration e efficiency**)
  - **Métodos**:
    - **Gerenciamento de Mémoria**: O construtor realiza a alocação dinâmica dos vetores com base no parâmetro η (capacidade). O destrutor é responsável por liberar a memória de todos os segmentos, das paradas associadas e dos próprios vetores de ponteiros.
    - **Operações**: Inclui `addDemand` e `addSegment` para inserção nos vetores manuais, além de `Getters` e `Setters` para as estatísticas da corrida.
- **Evento (Event)**: Unidade básica de simulação do sistema SED.
  - **Atributos**: Armazena o instante de ocorrência (**timestamp**), o tipo do evento (**type**), um ponteiro para a corrida afetada (**ride**) e o índice da parada a ser processada (**stopIndex**).
  - **Métodos**: Fornece acesso aos dados (`Getters`) para que o Escalonador (**Scheduler**) possa ordenar a execução e o motor de simulação possa processar a lógica de mudança de estado.
- **Escalonador (Scheduler)**: Estrutura de dados responsável pela ordem cronológica da simulação.
  - **Atributos**: Implementado sobre um vetor dinâmico de ponteiros para Eventos (**heapArray**) controlado por variáveis de capacidade e tamanho atual.
  - **Métodos**:
    - **Públicos**: `insert` (adiciona um evento) e `removeNext` (remove e retorna a raiz, ou seja, o evento de menor tempo).
    - **Privados** `siftup` e `siftDown`, algoritmos responsáveis por manter a propriedade de ordem do MinHeap após inserções e remoções, garantindo complexidade logarítmica.

2.2 Estruturas Auxiliares e Utilitários

Para manter o código modular e focado, funcionalidades de uso geral foram segregadas em componentes auxiliares:

- **Ponto (Point)**: Uma estrutura leve responsável apenas por armazenar pares de coordenadas cartesianas (x, y) e prover métodos de acesso. É utilizada como base para os atributos de localização em `Demand` e `Stop`.
- **Módulo de Utilidades (Utils)**: Um conjunto de funções estáticas desacopladas das classes principais. Este módulo centraliza a lógica matemática do sistema, contendo a implementação do cálculo de distância Euclidiana e a função complexa de cálculo de eficiência (λ), que simula rotas compartilhadas sem alterar o estado dos objetos. Essa separação facilita a manutenção e os testes unitários das regras de negócio matemáticas.

2.3 Algoritmos de Despacho e Simulação

A lógica de execução do sistema foi dividida em duas fases distintas: o processamento estático das demandas (Despacho) e a execução temporal dos eventos (Simulação).

2.3.1 Algoritmo de Despacho (Matching)

Esta fase implementa uma estratégia para o agrupamento de passageiros. O algorítmo itera sobre a lista de demandas ordenadas cronologicamente e, para cada solicitação pendente (c_0), tenta formar uma corrida compartilhada verificando candidatos subsequentes (c_j). A validação do agrupamento segue um fluxo de filtros pre-definidos:

1. **Verificação de Estado**: Ignora demandas que já foram alocadas em corridas anteriores.
2. **Capacidade (η)**: Garante que o número de passageiros não exceda o limite do veículo.
3. **Janela Temporal (δ)**: Interrompe a busca se a diferença de tempo entre a solicitação inicial e a candidata exceder o limite permitido.
4. **Proximidade Geográfica (α e β)**: Utiliza a distância euclidiana para verificar se a origem e o destino do candidato são próximos às origens e destinos de **todos** os passageiros já presentes na corrida. Esta verificação é conjuntiva.
5. **Eficiência (λ)**: Calcula a razão entre a soma das distâncias das corridas individuais e a distância total da rota compartilhada. Se a eficiência for inferior ao limiar λ, o candidato é descartado.

Após a definição do grupo de passageiros, a rota é construída seguindo a lógica FIFO (*First-In, First-Out*): o veículo realiza todas as coletas na ordem de solicitação e, em seguida, todas as entregas na mesma ordem. O primeiro evento da corrida (chegada para embarque do passageiro) é então inserido no Escalonador.

2.3.2 Motor de Simulação

A segunda fase consiste em um laço de repetição contínuo que consome eventos do Escalonador enquanto houver pendências. O processo segue os seguintes passos:

- **Avanço do Relógio**: O relógio lógico da simulação (simulationClock) é atualizado para o tempo (timestamp) do evento atual removido do MinHeap.
- **Processamento**: O sistema identifica a corrida e a parada associada ao evento.
- **Agendamento Dinâmico**: Se a parada atual não for a última da rota, o sistema calcula o tempo de chegada na próxima parada (baseado na distância do trecho e a velocidade γ) e insere um novo evento futuro no Escalonador.
- **Conclusão**: Caso seja a última parada, as estatísticas finais da corrida (tempo total, distância percorrida e sequência de coordenadas) são geradas e enviadas para a saída.

Esta separação entre a lógica de decisão (Algoritmo de Despacho) e a lógica temporal (Motor de Simulação) permite que o sistema processe regras complexas de agrupamento sem bloquear o fluxo da simulação.

3 Análise de Complexidade

A seguir, será apresentada a análise individual de cada etapa do algoritmo em relação à complexidade de tempo e espaço. A análise considera N como o número total de demandas de entrada e avalia sempre o pior caso de execução. Assume-se também que a capacidade do veículo (η) é uma constante pequena, de modo que operações internas que dependem dela são consideradas $O(1)$.

3.1 Análise de Tempo

A complexidade de tempo varia de acordo com a fase de execução do sistema: o Despacho (Matching) e a Simulação.

3.1.1 Algoritmo de Despacho

Esta etapa é responsável pelo agrupamento das demandas. A análise desta função é a mais crítica devido à sua estrutura de laços aninhados.

- A função possui um **laço externo principal** que itera sobre cada uma das N demandas (de $i = 0$ até $N - 1$) para definir o início de uma corrida
- Dentro deste laço, existe um **laço interno** que busca candidatos a compartilhamento nas demandas subsequentes (de $j = i + 1$ até $N - 1$) para definir o início de uma corrida.
- No pior caso (onde a janela de tempo δ é grande o suficiente para cobrir todas as requisições), o laço interno percorre todos os itens restantes para cada iteração do externo.
- As verificações internas (distância e eficiência) ocorrem em tempo constante $O(1)$, pois iteram apenas sobre a capacidade fixa do veículo (η).
- A complexidade total desta fase é dada pela soma das iterações do laço interno, que forma uma Progressão Aritmética:

$$\sum_{i=0}^{N-1}(N-1-i) = (N-1) + (N-2) + ... + 1 \approx \frac{N^2}{2}$$

- O termo dominante nesta série é quadrático. Portanto, a complexidade de tempo para a fase de Despacho é $O(N^2)$.

3.1.2 Motor de Simulação

Este algoritmo processa a execução temporal utilizando o Escalonador (*MinHeap*).

- O número total de eventos gerados é proporcional ao número de paradas. Como cada demanda gera um embarque e um desembarque, temos $2N$ eventos, o que corresponde a uma ordem de crescimento linear, ou seja $O(N)$.
- O laço principal da simulação consome cada um desses eventos.
- Em cada iteração, realizam-se operações de remoção (`removeNext`) e inserção (`insert`) na fila de prioridade.
- As operações no *MinHeap* possuem custo logarítmico em relação ao número de elementos na fila, ou seja, $O(\log N)$
- **Conclusão**: A complexidade total deste algoritmo é o produto do número de eventos pelo custo das operações no Heap:

$$O(N) \times O(\log N) = O(N \log N)$$

3.1.3 Complexidade Total de Tempo

A complexidade total é a soma das etapas, dominada pelo termo de maior crescimento.

- **Tempo Total** = (Tempo de Despacho) + (Tempo de Simulação)
- **Tempo Total** = $O(N^2) + O(N \log N)$

Assim, a complexidade assintótica final do algoritmo é:

$$T(N) = O(N^2)$$

3.2 Análise de Espaço

A análise de espaço considera a memória dinâmica utilizada pelas estruturas de dados manuais, tratando N como o número de demandas.

3.2.1 Estrutura de Dados

As principais fontes de consumo de memória são:

- **Vetores de Entidades**: O vetor principal de demandas (`listOfAllDemands`) e o vetor de corridas (`R`) são arrays de ponteiros alocados dinamicamente com tamanho proporcional à entrada. Seu espaço combinado é $O(N)$.
- **Estrutura do Heap**: O vetor interno do Escalonador (`heapArray`) armazena os eventos agendados. No pior caso, o número de eventos simultâneos escala linearmente com o número de demandas ativas, consumindo memória auxiliar de $O(N)$.

3.2.2 Complexidade Total de Espaço

A complexidade total é a soma do espaço de armazenamento das entidades e do espaço de trabalho do simulador.

- **Espaço Total:** Composto pela soma das estruturas lineares.
- **Espaço Total:** $O(N) + O(N) = O(N)$

Assim, a complexidade total de espaço do algoritmo é:

$$E(N) = O(N)$$

3.3 Conclusão

A análise conclui que a complexidade temporal é dominada pela estratégia de emparelhamento ($O(N^2)$), que se sobrepõe ao custo logarítmico da simulação. Em contrapartida, a complexidade espacial mantém-se linear ($O(N)$), uma vez que as estruturas de dados manuais escalam de forma previsível com a quantidade de demandas, sem o overhead de estruturas recursivas profundas.

4 Estratégias de Robustez

Para garantir a estabilidade do simulador, foram implementadas estratégias focadas na validação de dados e na integridade da memória, compensando a ausência de contêineres automáticos da STL.

4.1 Validação de Entradas

- **Prevenção de Buffer Overflow:** Checagens de limite (*bounds checking*) são realizadas antes de qualquer inserção nos vetores manuais (Demandas, Segmentos e Heap). Se a capacidade for excedida, a operação é abortada com feedback de erro.
- **Validação de Fluxo:** O estado do `std::cin` é verificado após cada leitura. Entradas malformadas geram mensagens em `std::cerr` e encerram a execução de forma controlada, impedindo o processamento de dados corrompidos.

4.2 Gerenciamento de Memória e Exceções

- **Tratamento de Falhas de Alocação:** O bloco principal de alocação no `main` é protegido por `try-catch`. A captura da exceção `std::bad_alloc` evita o encerramento abrupto do programa em caso de falta de memória.
- **Hierarquia de Propriedade:** Para evitar erros de *double free*, a classe `Ride` foi definida como proprietária exclusiva dos recursos. Seu destrutor libera paradas e trechos em uma ordem específica, garantindo que nenhum ponteiro seja deletado duas vezes.

4.3 Programação Defensiva

- **Validação de Premissas:** O uso de `assert` em estruturas críticas, como o `Scheduler`, valida pré-condições (índices válidos e ponteiros não nulos) durante o desenvolvimento.
- **Interfaces Seguras:** O uso de `const` em todos os métodos de acesso (*Getters*) protege o estado interno dos objetos contra modificações acidentais.

4.4 Tratamento de Erros Lógicos

- **Operações Matemáticas Seguras:** O cálculo de eficiência (λ) possui verificação explícita contra divisão por zero.
- **Feedback de Erro:** Falhas lógicas são reportadas exclusivamente via `std::cerr`, mantendo a saída padrão (`std::cout`) limpa para os dados da simulação.

5 Análise Experimental

A avaliação empírica do sistema buscou quantificar o impacto das restrições operacionais tanto no desempenho computacional quanto na eficácia do serviço de transporte. O foco da investigação recaiu sobre o comportamento do algoritmo de despacho diante da variação de seus parâmetros críticos, visando identificar o ponto de equilíbrio entre o tempo de resposta do sistema e a taxa de ocupação da frota. Foram analisados dois eixos principais: a disponibilidade de **Candidatos a Combinação** e o **Custo de Coleta e Entrega**.

5.1 Metodologia

Utilizou-se um gerador de dados sintéticos com $N = 1000$ demandas distribuídas em clusters geográficos aleatórios para cada execução. Para cada experimento, definiu-se uma configuração base ($\eta = 4, \delta = 60s, \alpha = 2000m, \beta = 4000m, \lambda = 0.5$) e variou-se apenas a dimensão sob análise (*ceteris paribus*).

Os intervalos de teste foram definidos para cobrir desde cenários restritivos até situações de relaxamento total das regras:

- **Janela Temporal**: $\delta \in \{10s, ..., 480s\}$.
- **Capacidade**: $\eta \in \{1, ..., 10\}$.
- **Restrições Espaciais**: $\alpha \in \{100m, ..., 4000m\}$ e $\beta \in \{500m, ..., 5000m\}$.
- **Eficiência**: $\lambda \in \{0.1, ..., 0.9\}$.

5.2 Avaliação: Candidatos a Combinação

5.2.1 Impacto da Janela Temporal (δ)

A janela temporal determina o horizonte de busca por candidatos.

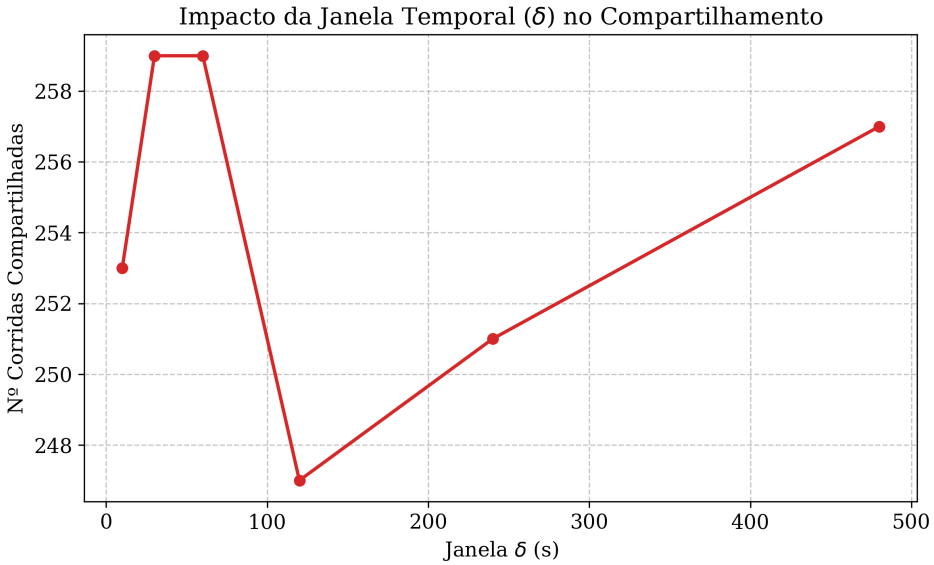

Figure 1: Comportamento do compartilhamento em função da janela temporal δ. A volatilidade da curva reflete a sensibilidade à topologia das demandas.

A Figura 1 apresenta um comportamento não-linear. Observa-se um pico inicial de compartilhamento para janelas curtas ($\delta \approx 50s$), seguido de uma queda abrupta e posterior recuperação. Essa oscilação evidencia que, para o algoritmo guloso, "esperar mais" nem sempre garante melhores resultados globais. Em janelas intermediárias ($\delta = 120s$), o algoritmo pode se comprometer prematuramente com combinações que, embora válidas temporalmente, são sub-ótimas geograficamente, impedindo agrupamentos mais eficientes que surgiriam posteriormente.

5.2.2 Impacto da Capacidade (η)

O parâmetro η define o limite físico de ocupação.

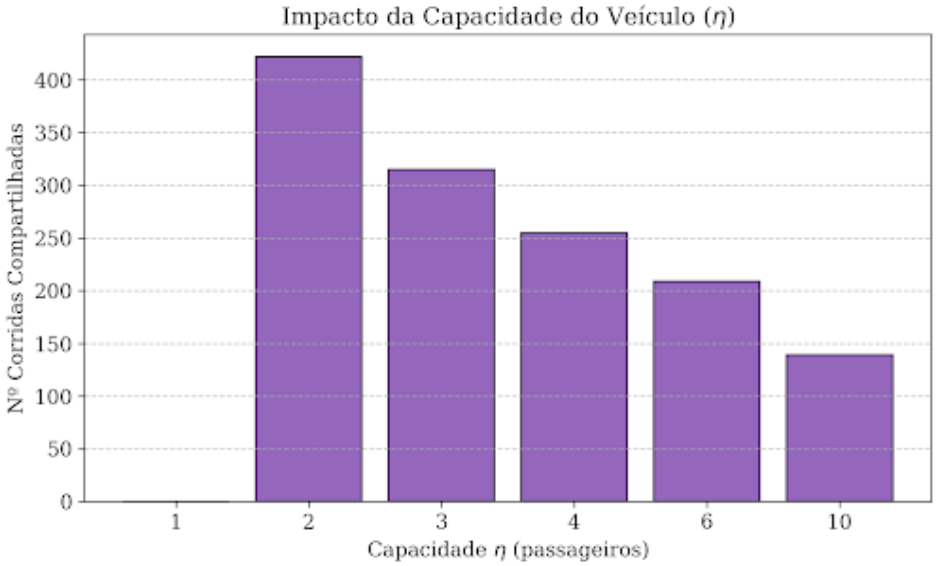

Figure 2: Influência da capacidade do veículo (η) no número de corridas compartilhadas.

A Figura 2 revela um resultado contraintuitivo e fundamental: o pico de compartilhamentos ocorreu com veículos de baixa capacidade ($\eta = 2$). À medida que a capacidade aumenta para $\eta = 10$, o número de corridas compartilhadas cai progressivamente. Isso sugere que a restrição dominante não é o espaço físico, mas sim a eficiência da rota (λ). É logisticamente inviável lotar um veículo grande sem que os desvios acumulados violem a tolerância de eficiência dos passageiros. O sistema tende naturalmente a formar "caronas duplas" otimizadas.

5.2.3 Restrições Espaciais (α e β)

Os parâmetros α (origem) e β (destino) definem o raio geográfico de busca. Para analisar seus impactos individualmente, fixou-se um parâmetro enquanto o outro variava.

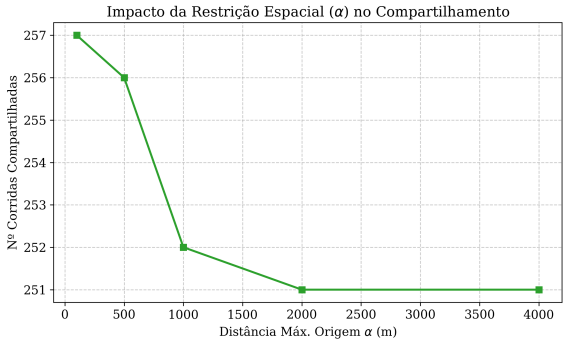

Figure 3: Impacto da distância máx. de coleta (α).

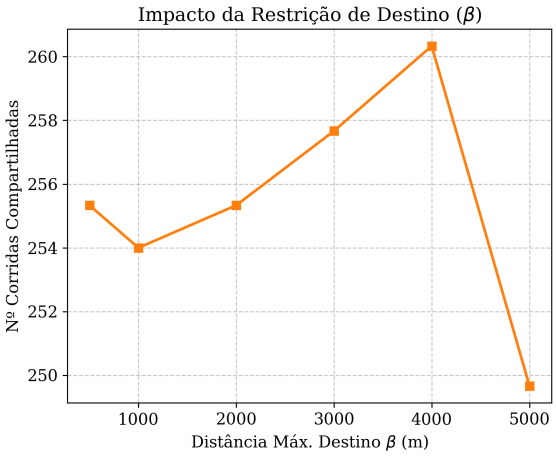

Figure 4: Impacto da distância máx. de entrega (β).

A análise comparativa das Figuras 3 e 4 revela comportamentos complementares.

- **Sensibilidade à Origem (α):** O sistema mostrou-se mais sensível a restrições na origem. Valores baixos de α derrubam drasticamente o compartilhamento, pois impedem o agrupamento inicial.
- **Tolerância no Destino (β):** O comportamento de β (Figura 4) tende a ser mais suave. Mesmo com restrições de destino mais rígidas, o sistema consegue formar parcerias se as origens forem próximas, desde que a rota final permaneça eficiente.

Ambos os gráficos apresentam um ponto de saturação a partir de 2000m, onde o aumento da tolerância geográfica deixa de gerar novas combinações válidas devido à barreira imposta pela eficiência (λ).

5.2.4 Limitação pela Eficiência (λ)

O parâmetro λ regula a qualidade da rota.

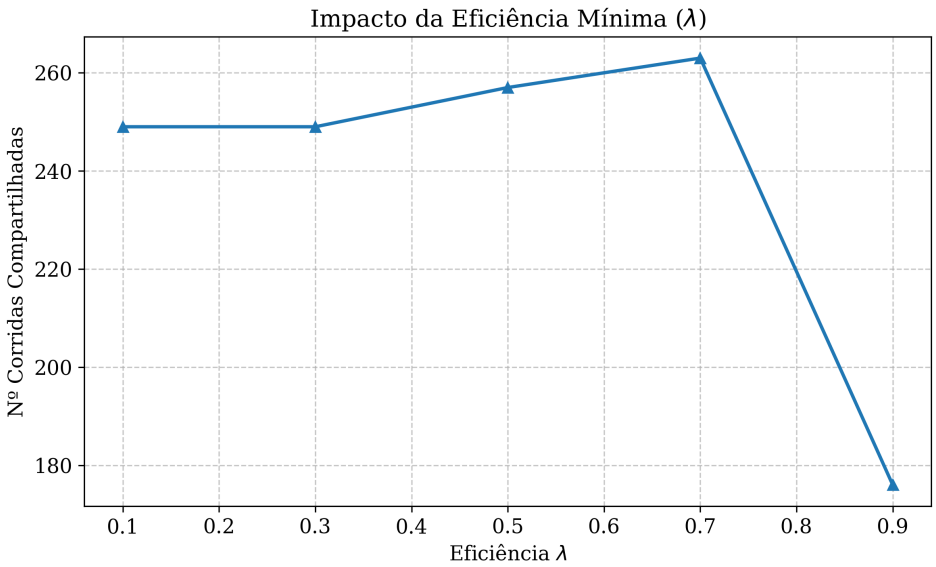

Figure 5: Estabilidade e ponto de ruptura na exigência de eficiência.

A Figura 5 demonstra a robustez do sistema. A taxa de compartilhamento mantém-se estável e alta (platô) para valores de λ entre 0.1 e 0.7. No entanto, observa-se um ponto de ruptura claro em $\lambda = 0.9$.

Ao exigir rotas quase diretas (90% de eficiência), o compartilhamento torna-se inviável para a vasta maioria das demandas, forçando uma reversão para o transporte individual. Isso confirma que existe uma margem operacional segura onde é possível otimizar a frota sem sacrificar excessivamente o tempo do passageiro.

5.3 Síntese da Análise Experimental

A bateria de testes permitiu traçar o perfil de comportamento do algoritmo guloso. Ficou evidenciado que o aumento indiscriminado da janela de busca (δ) ou do raio de coleta e de entrega (α, β) oferece retornos decrescentes: o custo computacional cresce quadraticamente, enquanto a qualidade da solução satura rapidamente devido às limitações geométricas das rotas.

O resultado mais significativo foi a identificação da eficiência mínima (λ) como o parâmetro preponderante na determinação da frota. Enquanto a capacidade física (η) mostrou ter impacto secundário — com veículos menores apresentando melhor desempenho relativo de ocupação — a exigência de eficiência atuou como o verdadeiro filtro de viabilidade. Conclui-se que, para este modelo de despacho, a otimização não é alcançada pelo aumento da capacidade dos veículos ou pelo tempo de espera, mas sim pelo ajuste fino da tolerância a desvios nas rotas individuais.

6 Conclusão

O presente trabalho consistiu no desenvolvimento completo de um sistema de despacho para corridas compartilhadas, aplicando a técnica de Simulação de Eventos Discretos (SED) para modelar a complexidade logística do transporte urbano. A implementação abrangeu desde a estruturação de Tipos Abstratos de Dados (TADs) para representar a frota e as demandas, até a construção de um motor de simulação temporal baseado em filas de prioridade.

Durante o desenvolvimento, foi possível consolidar o aprendizado sobre o gerenciamento manual de memória em C++. A ausência de bibliotecas de alto nível exigiu uma disciplina rigorosa na alocação e liberação de recursos, reforçando a importância de estratégias de robustez, como a definição clara de propriedade de ponteiros e o tratamento de exceções, para evitar vazamentos de memória e falhas de execução.

Do ponto de vista algorítmico, o projeto evidenciou na prática os compromissos (*trade-offs*) inerentes ao design de software. A análise experimental demonstrou que, embora estruturas eficientes como o *MinHeap* garantam a fluidez da simulação temporal ($O(N \log N)$), a lógica de negócios — o emparelhamento guloso das demandas — impõe um custo quadrático ($O(N^2)$) que se torna o gargalo do sistema. Aprendeu-se que a otimização de um serviço de transporte não depende apenas de algoritmos rápidos, mas do ajuste fino de parâmetros conflitantes, onde a exigência de eficiência (λ) e a tolerância temporal (δ) definem o limite entre um sistema coletivo viável e o transporte individual.

Em suma, o trabalho integrou conceitos fundamentais de Estruturas de Dados com práticas de engenharia de software, resultando em um simulador funcional capaz de oferecer *insights* quantitativos sobre mobilidade e eficiência operacional.

7 Bibliografia

References

[1] Lacerda, A.; Santos, M.; Meira Jr, W.; Cunha, W. *Especificação do Trabalho Prático 2: Sistema de Despacho de Transporte por Aplicativo*. Departamento de Ciência da Computação, UFMG. 2025/2.

[2] Lacerda, A.; Santos, M.; Meira Jr, W.; Cunha, W. *Estruturas de Dados - Material Didático (Slides)*. Disponível via Moodle. DCC/ICEx/UFMG, 2025.

[3] Chamowicz, L. *Programação e Desenvolvimento de Software II - Material Didático (Slides)*. Disponível via Moodle. DCC/ICEx/UFMG, 2025.

[4] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms*. 3rd Edition. MIT Press, 2009.