

UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Trabalho Prático 2

Sistema de Despacho de Transporte por Aplicativo

Cristiano Martins Guimarães Grandi
Matrícula: 2025423440

Belo Horizonte
25 de novembro de 2025

1 Introdução

Com a expansão da empresa multinacional *CabAl* para o Brasil, sob a marca *Cabe Aí*, surge a necessidade de implementar um sistema de despacho eficiente que suporte a inovação de corridas compartilhadas. O contexto atual de mobilidade urbana exige soluções que não apenas transportem passageiros, mas que o façam otimizando recursos. A motivação principal para a adoção do sistema de compartilhamento (tipo táxi-lotação) reside na redução de custos tanto para o usuário final, que paga uma tarifa menor, quanto para a eficiência operacional da frota.

O problema central abordado neste trabalho é a alocação dinâmica de veículos para atender demandas de transporte, respeitando restrições rígidas de capacidade (η), tempo (δ), distância espacial (α, β) e eficiência operacional (λ). O desafio computacional é identificar, em tempo hábil, quais demandas podem ser combinadas sem violar essas restrições.

A solução empregada utiliza uma abordagem de **Simulação de Eventos Discretos (SED)**. Diferente de uma simulação contínua, o sistema modelado avança o tempo através de saltos discretos marcados pela ocorrência de eventos (solicitações de corrida, chegadas e partidas), gerenciados por uma estrutura de fila de prioridade (*MinHeap*). Isso permite processar um grande volume de demandas de forma cronológica e eficiente.

2 Método

A implementação foi realizada na linguagem C, estruturada de forma modular com separação clara entre tipos de dados (*types.h*), estruturas de dados (*minheap.h*), lógica de corridas (*ride.h*), funções auxiliares (*util.h*) e fluxo principal (*main.c*). O sistema baseia-se em quatro Tipos Abstratos de Dados (TADs) principais:

2.1 Tipos Abstratos de Dados

- **MinHeap (Escalonador):** Implementado em *minheap.h*, atua como o motor da simulação. É uma fila de prioridade que armazena eventos (paradas) ordenados pelo tempo. A operação *get_next* garante que o sistema sempre processe o evento mais iminente, fundamental para a corretude cronológica da SED.
 - *initialize*(MinHeap*, int): Aloca memória para o heap com tamanho especificado
 - *insert_new*(MinHeap*, RideStop*, double): Insere uma nova parada no heap, calculando seu tempo e mantendo a propriedade de heap mínimo através de *heapify*
 - *get_next*(MinHeap*): Remove e retorna a parada com menor tempo, restaurando a propriedade de heap através de *heapify*
 - *peek*(MinHeap*): Retorna a próxima parada sem removê-la
 - *finalize*(MinHeap*, double): Processa todos os eventos restantes, imprimindo os resultados das corridas
- **Ride (Corrida):** Definida em *types.h* e manipulada em *ride.h*, esta estrutura agrega as informações de uma viagem, contendo uma lista de passageiros (demandas)

e uma lista encadeada de paradas (`RideStop`). Ela gerencia o estado da corrida (individual ou compartilhada) e calcula métricas como a eficiência.

- `create_new_ride()`: Aloca e inicializa uma nova corrida vazia
 - `add_ride_demand(Ride*, Demand)`: Adiciona uma demanda à corrida, atualizando o status (`SINGLE` ou `COMBINED`)
 - `add_ride_stops(Ride*, Demand, double)`: Insere as paradas de origem e destino na lista encadeada de paradas, mantendo a ordem: todas as origens primeiro, seguidas de todos os destinos
 - `calculate_ride_efficiency(Ride)`: Calcula a eficiência como a razão entre a soma das distâncias diretas de cada passageiro e a distância total percorrida
 - `get_ride_total_distance(Ride)`: Percorre a lista de paradas calculando a distância total do trajeto
 - `remove_last_added_demand(Ride*)`: Remove a última demanda adicionada (usado quando uma tentativa de compartilhamento falha)
 - `remove_last_added_stops(Ride*)`: Remove as duas últimas paradas (origem e destino) da lista encadeada
- **RideStop (Parada)**: Representa uma parada (origem ou destino) no trajeto. Implementada como nó de lista duplamente encadeada, contém coordenadas (x, y) , tempo de chegada, distância acumulada, tipo (origem ou destino), ID da demanda associada e ponteiros para a corrida e paradas adjacentes.
 - **Demand (Demanda)**: Armazena informações de uma solicitação: ID, tempo de solicitação, status (demandada, individual, combinada ou finalizada) e estruturas `RideStop` para origem e destino.

2.2 Funções Auxiliares

O módulo `util.h` fornece funções de suporte:

- `get_distance(RideStop, RideStop)`: Calcula a distância euclidiana entre duas paradas usando `hypot()`
- `meets_distance_criteria(Ride*, Demand, double, double)`: Verifica se uma demanda satisfaz as restrições espaciais α e β em relação a todas as demandas já presentes na corrida

2.3 Algoritmo de Despacho e Compartilhamento

O fluxo principal, implementado em `main.c`, lê os parâmetros de configuração e as demandas, processando-as sequencialmente. Para cada nova demanda, o sistema:

1. Itera sobre as corridas ativas (ainda não finalizadas)
2. Verifica se a corrida tem capacidade disponível (η)
3. Verifica o critério temporal: se o tempo de solicitação da nova demanda está dentro da janela δ em relação ao início da corrida

4. Verifica as restrições espaciais usando `meets_distance_criteria()`, garantindo que as distâncias entre origens (α) e destinos (β) sejam respeitadas
5. Tenta adicionar a demanda à corrida usando `add_ride_demand()` e `add_ride_stops()`
6. Calcula a eficiência resultante com `calculate_ride_efficiency()`
7. Se a eficiência for maior ou igual a λ , aceita o compartilhamento; caso contrário, remove a demanda usando `remove_last_added_demand()` e `remove_last_added_stops()`
8. Se nenhuma corrida compatível for encontrada, cria uma nova corrida individual
9. Insere a primeira parada (origem) da corrida no MinHeap usando `insert_new()`

Após processar todas as demandas, o sistema chama `finalize()` no MinHeap, que processa os eventos cronologicamente, inserindo paradas subsequentes conforme as anteriores são processadas, até que todas as corridas sejam concluídas.

3 Análise de Complexidade

A eficiência do sistema depende majoritariamente das operações no MinHeap e da busca por corridas compartilháveis.

- **Espaço:** A complexidade de espaço é $O(N)$, onde N é o número de demandas, para armazenar as estruturas de dados de entrada. O Heap ocupa espaço proporcional ao número de eventos ativos simultâneos.
- **Tempo (Heap):** As operações de inserção (`insert_new`) e remoção (`get_next`) no MinHeap possuem complexidade $O(\log K)$, onde K é o número de eventos agendados.
- **Tempo (Simulação):** Para cada demanda, o algoritmo itera sobre as corridas ativas para tentar o compartilhamento. No pior caso, onde nenhuma combinação é possível, a complexidade aproxima-se de $O(N \cdot M)$, onde M é a janela de corridas ativas consideradas (limitadas por δ).

4 Estratégias de Robustez

Para garantir a estabilidade e confiabilidade do sistema, foram implementadas cinco estratégias principais de programação defensiva, organizadas em um módulo dedicado de validação e verificação:

4.1 Validação de Entrada

Implementada no módulo `validation.h`, esta estratégia valida todos os dados de entrada antes do processamento:

- **Parâmetros de Configuração:** A função `validate_config_params()` verifica se os parâmetros do sistema estão em intervalos válidos: capacidade $\eta > 0$ e $\eta \leq \text{MAX_CAPACITY}$, velocidade $\gamma > 0$, tempo máximo de espera $\delta \geq 0$, distâncias máximas $\alpha, \beta \geq 0$, e eficiência mínima $\lambda \in [0, 1]$. Valores fora desses intervalos causam encerramento controlado do programa com mensagem de erro descritiva.

- **Validação de Demandas:** A função `validate_demand()` verifica cada solicitação individualmente, garantindo que coordenadas sejam válidas, que origem e destino sejam diferentes (distância > 0.01), e que o tempo de solicitação seja não-negativo. Demandas inválidas são descartadas com aviso, permitindo que o sistema continue processando as demais.
- **Contagem de Demandas:** A função `validate_demand_count()` verifica se o número de demandas é positivo e emite aviso se exceder 100.000, indicando possível problema de desempenho.

4.2 Verificação de Alocação de Memória

Todas as chamadas a `malloc()` foram instrumentadas com verificação de retorno NULL:

- `create_new_ride()`: Verifica alocação de estrutura Ride
- `initialize()`: Verifica alocação do array de paradas do MinHeap
- `create_new_stop()`: Verifica alocação de cada parada individual

Em caso de falha de alocação, o sistema imprime mensagem de erro detalhada e encerra com `exit(1)`, prevenindo segmentation faults por acesso a ponteiros NULL.

4.3 Proteção contra Divisão por Zero

Operações aritméticas críticas foram protegidas contra divisão por zero:

- `calculate_stop_time()`: Verifica se velocidade > 0 antes de calcular tempo = distância/velocidade
- `calculate_ride_efficiency()`: Verifica se distância total > 0.0001 (tolerância para comparação de floats) antes de calcular eficiência = distância_original/distância_total

Estas verificações previnem resultados que corromperiam o estado do sistema.

4.4 Verificação de Limites de Arrays

Implementada proteção contra buffer overflow no array `demands[MAX_CAPACITY]`:

- `add_ride_demand()`: Verifica se `demand_number < MAX_CAPACITY` antes de adicionar
- `remove_last_added_demand()`: Verifica se `demand_number > 0` antes de remover
- `meets_distance_criteria()`: Verifica se `demand_number ≤ MAX_CAPACITY` antes de iterar

Violações de limites resultam em mensagem de erro e aborto da operação, prevenindo corrupção de memória.

4.5 Validação de Invariantes de Estruturas de Dados

Implementada no módulo `invariants.h`, esta estratégia verifica propriedades estruturais que devem sempre ser verdadeiras:

- **Invariantes Temporais:** A função `validate RideTemporalConsistency()` verifica que os tempos de paradas são monotonicamente crescentes ao longo da corrida, garantindo a consistência temporal do sistema.
- **Invariantes do Heap:** A função `validateHeapTimes()` verifica que todos os tempos no MinHeap são não-negativos, e `isValidMinHeap()` verifica a propriedade de heap mínimo (cada pai tem tempo menor ou igual aos filhos).

Estas validações são chamadas em pontos críticos (antes de finalizar o heap) e ajudam a detectar bugs de lógica durante desenvolvimento e testes.

4.6 Consistência Geométrica

O cálculo de distâncias utiliza a função `hypot()` da biblioteca matemática padrão, garantindo precisão em ponto flutuante e tratamento adequado de casos extremos (coordenadas muito grandes ou muito pequenas), evitando erros de overflow/underflow que ocorreriam com cálculo manual de $\sqrt{x^2 + y^2}$.

5 Análise Experimental

Os experimentos visaram avaliar o impacto dos parâmetros de configuração no desempenho do sistema, conforme sugerido na especificação. Nos experimentos, fixamos os parâmetros base como $\eta = 3$, $\Delta = 30s$, $\lambda = 0.1$, $\alpha = 1500$, $\beta = 3000$. A partir deles, foram variados α , β e λ .

5.1 Configuração do Experimento e Geração de Dados

Para garantir a reprodutibilidade e testar o sistema sob condições de estresse controlado, não foram utilizados apenas dados aleatórios uniformes. Foi desenvolvido um script auxiliar em Python para gerar um conjunto de 1000 demandas sintéticas (`input_1000.txt`).

A geração dos dados seguiu uma lógica de agrupamento: foram definidos 9 pontos-base de coordenadas (ex: $(0, 0)$, $(100, 100)$, $(-100, 0)$), e as origens e destinos de cada demanda foram gerados aplicando uma distribuição normal com desvio padrão de 50 ao redor desses pontos.

Para os experimentos a seguir, fixou-se a capacidade do veículo ($\eta = 3$), a velocidade ($\gamma = 35$) e o tempo máximo de espera ($\delta = 30$), variando-se apenas as restrições espaciais.

5.1.1 Métricas Avaliadas

Para avaliar o desempenho do sistema de compartilhamento, foram coletadas três métricas principais em cada experimento:

1. **Número de Corridas:** Representa a quantidade total de veículos despachados para atender todas as demandas. Esta métrica é inversamente proporcional à eficiência do agrupamento: quanto menor o número de corridas para um conjunto fixo

de demandas, maior foi a taxa de compartilhamento. Um sistema ideal de compartilhamento de corridas minimiza esta métrica respeitando as restrições de qualidade de serviço.

2. **Distância Total Percorrida:** Soma das distâncias percorridas por todos os veículos da frota. Esta métrica reflete o custo operacional total do sistema, incluindo consumo de combustível, desgaste dos veículos e emissões. O compartilhamento eficiente tende a reduzir esta métrica, pois substitui múltiplas trajetórias individuais por trajetos otimizados que atendem múltiplos passageiros. No entanto, desvios excessivos para coleta/entrega podem aumentar a distância total, evidenciando o *trade-off* entre agrupamento e eficiência de rota.
3. **Proporção Corridas/Demandas:** Razão entre o número de corridas realizadas e o número total de demandas. Esta métrica normalizada varia entre 0 e 1, onde 1.0 indica ausência total de compartilhamento (cada demanda gerou uma corrida individual) e valores próximos de 0 indicam alto grau de agrupamento. Por exemplo, uma proporção de 0.5 significa que, em média, cada corrida atendeu 2 demandas. Esta métrica é particularmente útil para comparar o desempenho do sistema sob diferentes configurações de parâmetros, independentemente do tamanho do conjunto de dados.

5.2 Impacto da Variação da Distância Máxima entre Origens (α)

O parâmetro α define o raio máximo de desvio permitido para coletar passageiros adicionais em relação à origem da primeira demanda. No experimento, variou-se α no intervalo $[0, 500]$ metros. A proporção corridas/demandas é uma métrica direta da eficiência do agrupamento: um valor de 1.0 indica que cada demanda gerou uma corrida individual (sem compartilhamento), enquanto valores menores indicam agrupamento bem-sucedido.

A hipótese inicial é que valores muito baixos de α impedem o compartilhamento, pois exigem que os passageiros estejam praticamente no mesmo local, resultando em proporções próximas de 1.0.

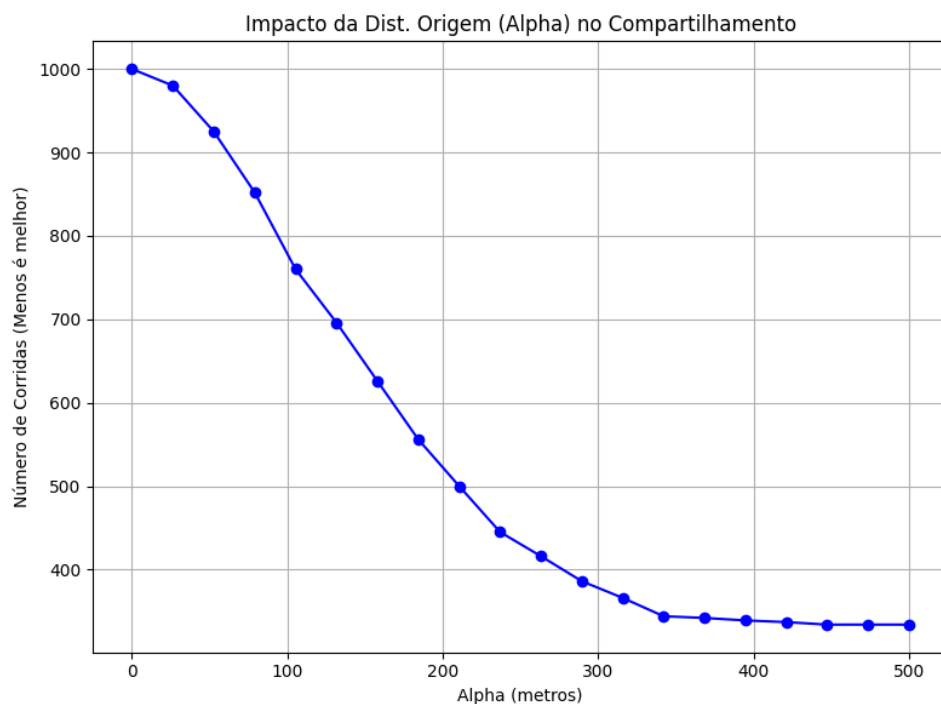


Figura 1: Impacto do relaxamento da restrição espacial (α) no número de corridas.

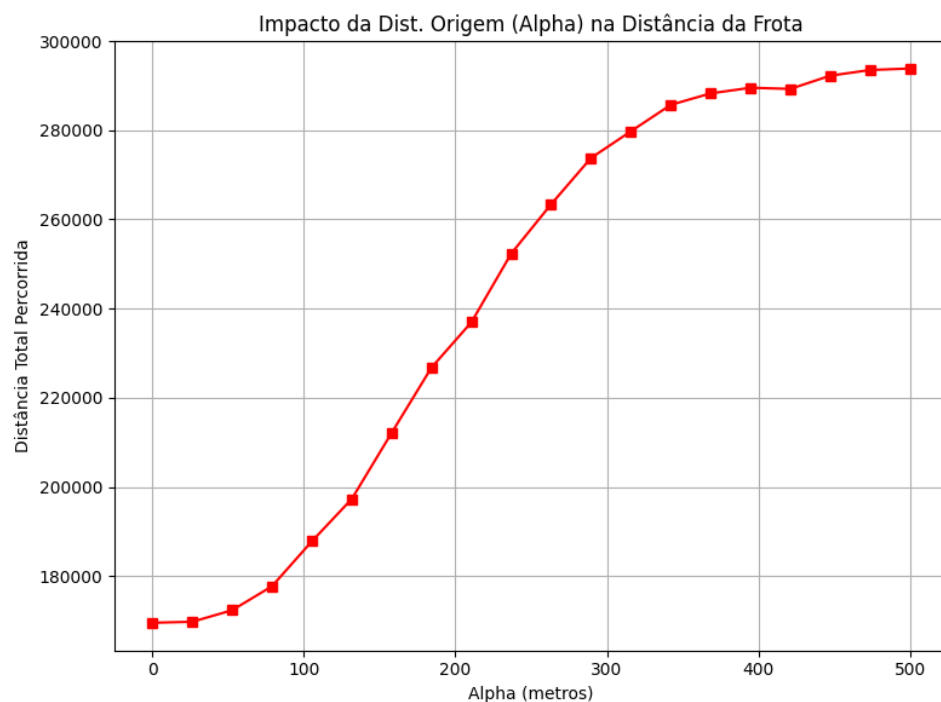


Figura 2: Impacto do relaxamento da restrição espacial (α) na distância total percorrida.

Conforme observado nos resultados obtidos, o relaxamento de α resultou em uma redução no número total de corridas. Isso indica que o algoritmo foi capaz de agrupar mais passageiros em um único veículo.

No entanto, observa-se um ponto de saturação. A partir de certo valor de α (aproximadamente 300), o ganho marginal de agrupamento diminui consideravelmente, pois o fator limitante passa a ser a capacidade do veículo (η), a eficiência (λ) ou o tempo de espera (δ), e não mais a distância física entre as origens.

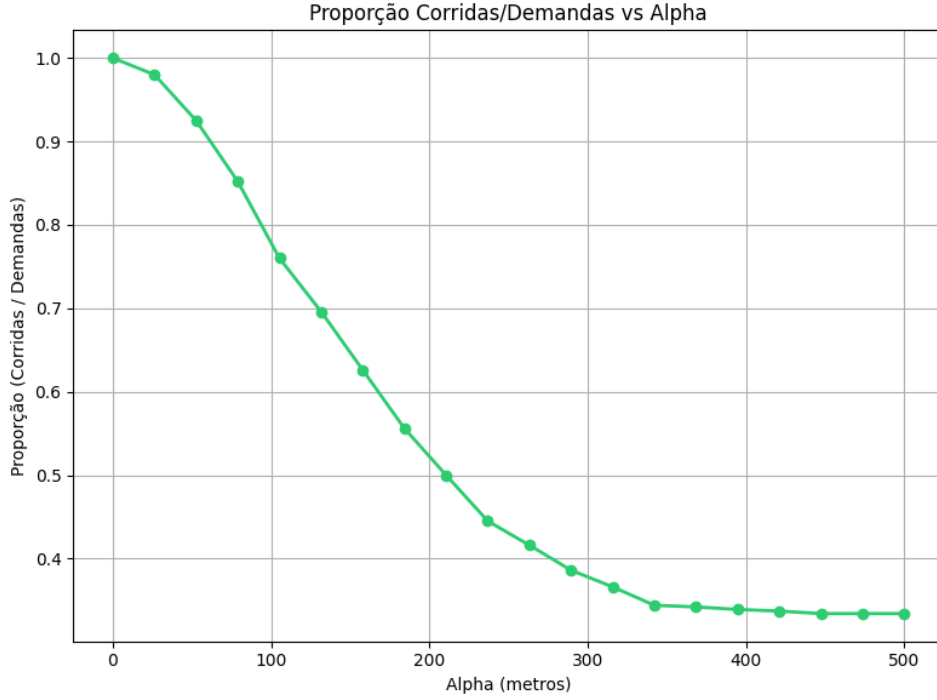


Figura 3: Proporção entre número de corridas e demandas em função de α .

Os resultados confirmam a hipótese: com α próximo de 0, a proporção se aproxima de 1.0, indicando ausência de compartilhamento. À medida que α aumenta, observa-se uma redução significativa na proporção, demonstrando que o relaxamento da restrição espacial permite agrupar mais demandas por corrida. A curva apresenta um ponto de saturação em torno de 300 metros, onde o ganho marginal diminui, sugerindo que outros parâmetros (β , η ou λ) passam a limitar o agrupamento.

5.3 Impacto da Variação da Distância Mínima entre Destinos (β)

O parâmetro β restringe a dispersão dos pontos de desembarque. O experimento variou β no intervalo $[0, 500]$ metros, mantendo α fixo em seu valor base. Este parâmetro atua como um filtro secundário crítico: mesmo que dois passageiros tenham origens próximas (satisfeito por α), eles só podem compartilhar a corrida se seus destinos também forem compatíveis. Com β restritivo, apenas passageiros com destinos muito próximos podem ser agrupados, resultando em proporções elevadas (próximas de 1.0).

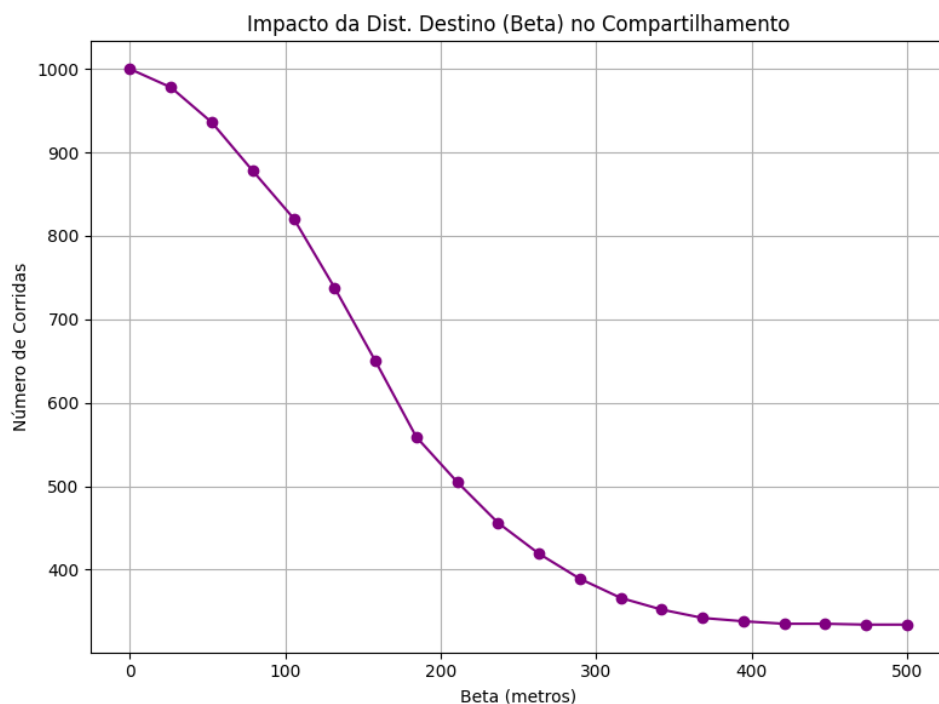


Figura 4: Impacto do relaxamento da restrição espacial (β) no número de corridas.

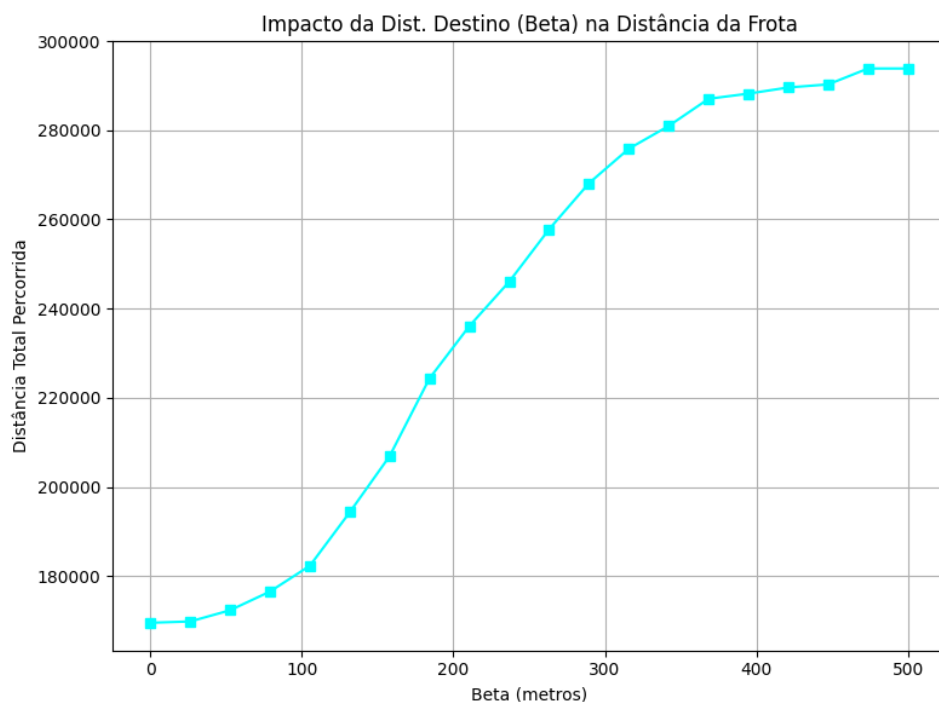


Figura 5: Impacto do relaxamento da restrição espacial (β) na distância total percorrida.

O gráfico de "Distância Total da Frota" em função de β revela a eficiência do sistema. Com β muito restrito (próximo de 0), o sistema opera quase como um táxi individual, resultando em uma distância total percorrida alta (soma de todas as trajetórias individuais). À medida que β aumenta, a distância total da frota tende a cair drasticamente, pois

múltiplos trajetos individuais são substituídos por trajetos otimizados compartilhados, validando a eficácia da lógica de compartilhamento de corridas implementada.

Neste experimento, fixamos os parâmetros base ($\eta = 3$, $\Delta = 30s$, $\lambda = 0.1$) e variamos a distância máxima entre destinos (β) de 0m a 500m.

Observa-se que, à medida que β aumenta, o sistema encontra mais oportunidades de compartilhamento, reduzindo o número total de veículos necessários. No entanto, o aumento excessivo de β pode degradar a eficiência individual de cada passageiro devido aos desvios maiores.

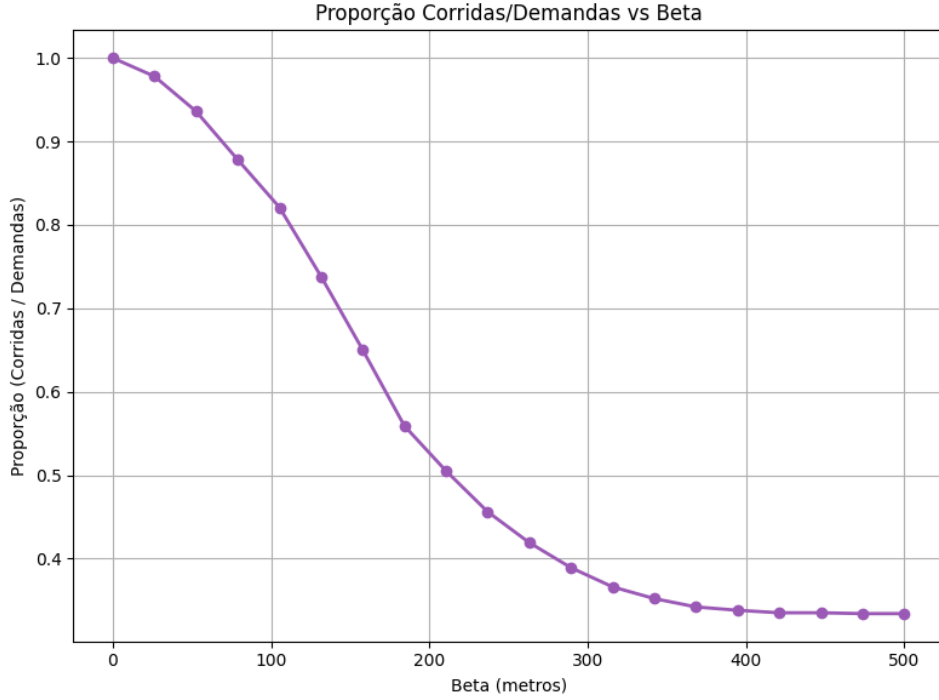


Figura 6: Proporção entre número de corridas e demandas em função de β .

O gráfico confirma o papel crítico de β no agrupamento. O relaxamento progressivo de β reduz a proporção corridas/demandas, permitindo que o sistema agrupe demandas com destinos mais dispersos. A curva demonstra que β é um gargalo significativo: mesmo com origens compatíveis, destinos incompatíveis impedem o compartilhamento. O comportamento é similar ao observado com α , indicando que ambos os parâmetros espaciais (origem e destino) devem ser ajustados em conjunto para otimizar o agrupamento no sistema.

5.4 Eficiência Mínima (Variação de λ)

O parâmetro λ representa a eficiência mínima aceitável para cada passageiro em uma corrida compartilhada, definida como a razão entre a distância direta (origem-destino) e a distância real percorrida. Valores baixos de λ (ex: 0.1) permitem desvios significativos, facilitando o agrupamento, enquanto valores próximos de 1.0 exigem trajetos quase diretos, inviabilizando compartilhamentos. Este parâmetro controla o trade-off entre eficiência operacional da frota (mais compartilhamento) e qualidade de serviço individual (menor tempo de viagem).

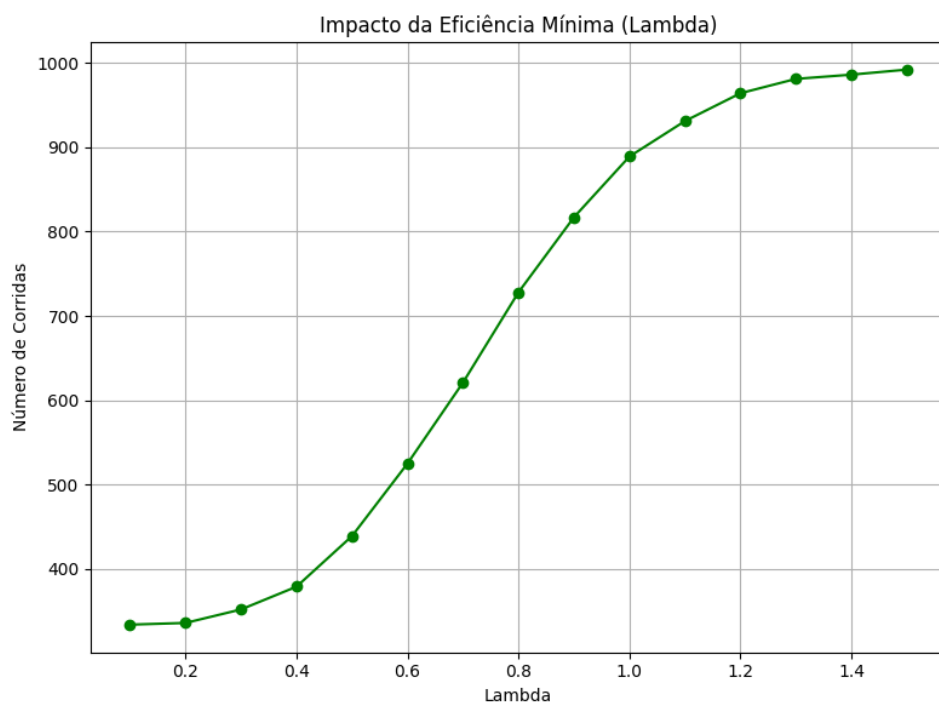


Figura 7: Relação entre exigência de eficiência (λ) e taxa de compartilhamento.

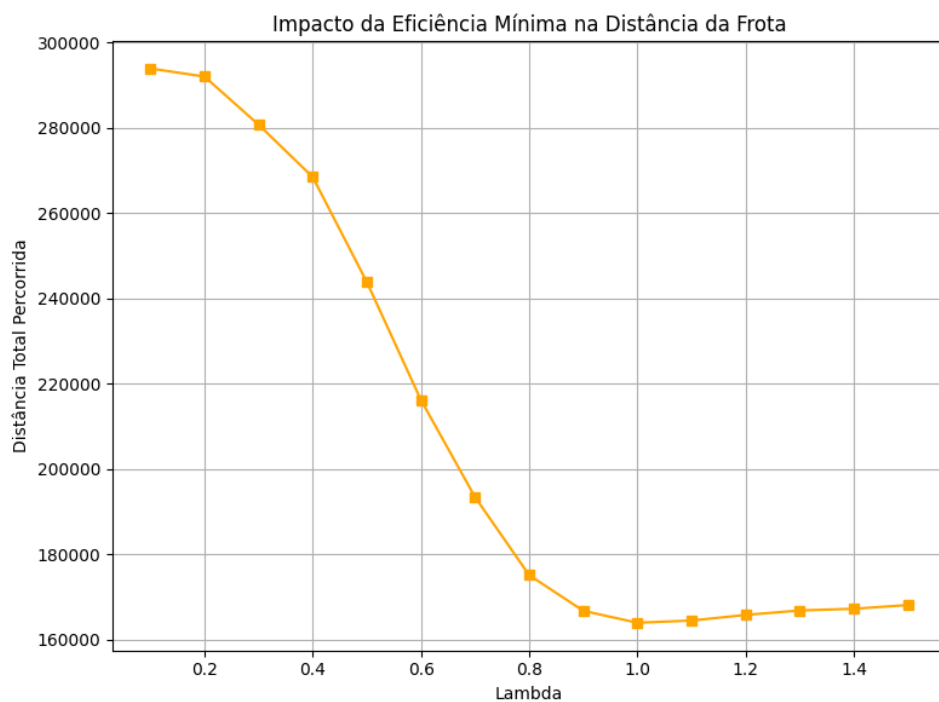


Figura 8: Relação entre exigência de eficiência (λ) e distância total percorrida.

Os gráficos revelam o comportamento esperado do sistema em relação à eficiência mínima. O número de corridas aumenta monotonicamente com λ , indicando que restrições mais rígidas de eficiência reduzem as oportunidades de agrupamento. Com λ próximo de 1.0, o sistema opera quase exclusivamente com corridas individuais, maximizando o

número de veículos necessários.

A distância total percorrida pela frota apresenta comportamento interessante: inicialmente, com λ baixo, a distância é relativamente alta devido aos desvios necessários para coletar e entregar múltiplos passageiros. À medida que λ aumenta, o sistema rejeita compartilhamentos com desvios excessivos, o que pode reduzir ligeiramente a distância total em alguns casos. No entanto, com λ muito alto, a distância total volta a aumentar, pois o sistema é forçado a realizar múltiplas corridas individuais ao invés de trajetos compartilhados otimizados. Este comportamento evidencia que existe um ponto ótimo de λ que equilibra compartilhamento e eficiência de rota.

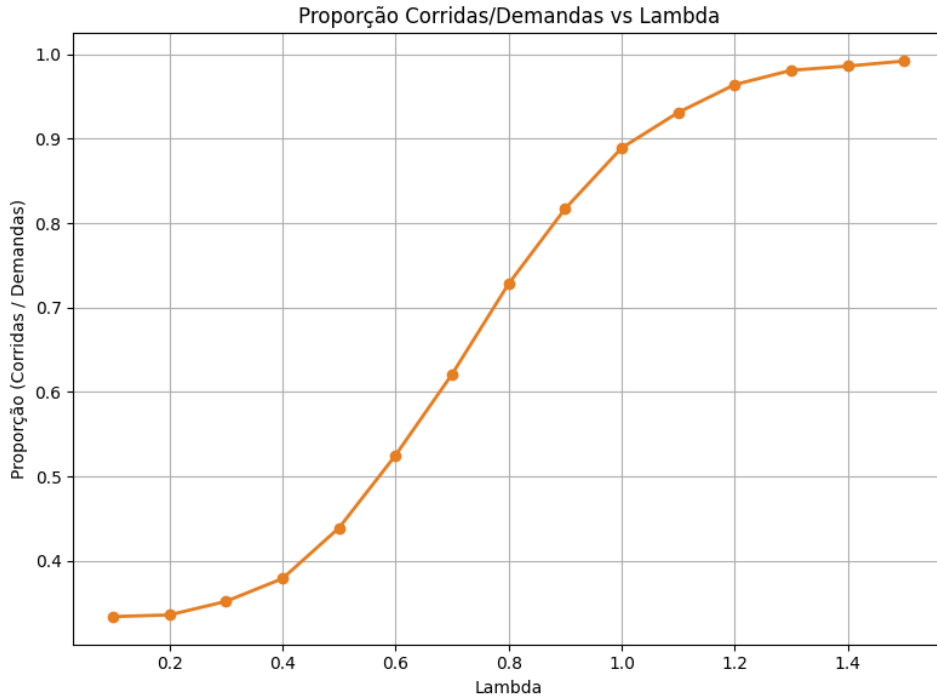


Figura 9: Proporção entre número de corridas e demandas em função de λ .

Os resultados demonstram claramente o impacto de λ no agrupamento. Com valores baixos (0.1 a 0.5), a proporção corridas/demandas permanece relativamente baixa, indicando alto grau de compartilhamento. À medida que λ aumenta, o sistema torna-se progressivamente mais restritivo, rejeitando compartilhamentos que impliquem desvios excessivos. Com λ próximo de 1.0, a proporção se aproxima de 1.0, indicando que o sistema opera quase exclusivamente com corridas individuais. Este comportamento evidencia que λ é o parâmetro de controle mais direto do trade-off entre custo operacional e qualidade de serviço.

6 Conclusões

Neste trabalho, foi implementado um simulador de despacho de corridas baseado em Simulação de Eventos Discretos (SED), capaz de avaliar e executar compartilhamento de veículos sob múltiplas restrições operacionais. A abordagem de SED mostrou-se adequada para modelar o sistema de transporte, onde o estado do sistema evolui através de eventos pontuais no tempo (solicitações de corrida, partidas e chegadas) ao invés de mudanças

contínuas. O uso da estrutura de dados *MinHeap* como escalonador de eventos provou-se essencial para gerenciar a ordem cronológica dos eventos de forma eficiente, garantindo que cada evento seja processado no momento correto da simulação.

A implementação da SED permitiu avaliar sistematicamente o impacto dos parâmetros de configuração no desempenho do sistema. Os experimentos demonstraram que a eficiência de um sistema de compartilhamento de corridas é um trade-off complexo entre múltiplas dimensões: restrições espaciais (α e β), temporais (δ) e de qualidade de serviço (λ). A análise experimental confirmou que os parâmetros espaciais devem ser ajustados em conjunto para otimizar o agrupamento, e que λ controla diretamente o equilíbrio entre eficiência operacional da frota e qualidade de serviço individual.

A metodologia de simulação adotada demonstrou ser uma ferramenta poderosa para análise de sistemas complexos de despacho, permitindo explorar cenários e configurações que seriam impraticáveis de testar em ambiente real. A modularidade do código e a separação clara entre estruturas de dados e lógica de simulação facilitam futuras extensões, como a incorporação de múltiplas classes de veículos, priorização dinâmica de demandas ou algoritmos de roteamento mais sofisticados.

7 Bibliografia

1. WIKIPEDIA. *Simulação de eventos discretos*. Disponível em: https://pt.wikipedia.org/wiki/Simulacao_de_eventos_discretos. Acesso em: nov. 2025.
2. Slides das aulas de Estrutura de Dados acerca de Heap. DCC/UFMG, 2025.