Trabalho Prático 2: Sistema de Despacho de Transporte por Aplicativo

Marney Santos de Melo
Matrícula: [Oculto para versão pública]

Novembro de 2025

Sumário

1 Introdução

No contexto moderno de mobilidade urbana, a eficiência dos serviços de transporte por aplicativo é crucial. Um dos maiores desafios enfrentados por empresas do setor é a otimização do despacho de veículos, especialmente quando se considera a modalidade de corridas compartilhadas. O problema consiste não apenas em alocar veículos para passageiros, mas em identificar quais solicitações podem ser agrupadas para maximizar a ocupação dos veículos sem violar restrições de tempo e distância.

Este trabalho aborda a implementação de um sistema de despacho e simulação de corridas compartilhadas para a empresa fictícia "Cabe Aí". O objetivo é desenvolver um simulador de eventos discretos que processe uma lista de demandas ordenada cronologicamente, aplicando um algoritmo guloso para combinar corridas com base em critérios de capacidade, proximidade espacial (origem e destino) e eficiência da rota.

A solução proposta utiliza Tipos Abstratos de Dados (TADs) para modelar as entidades do sistema e implementa uma estrutura de fila de prioridade (Min-Heap) para gerenciar o escalonamento dos eventos da simulação, garantindo a execução correta da cronologia das viagens.

2 Método

O sistema foi desenvolvido na linguagem C++ e compilado conforme o padrão C++11, utilizando o compilador G++, utilizando o Sistema Operacional Linux Ubuntu 22.04 LTS, respeitando as restrições imporstas no enunciado (ausência de STL como `std::vector` ou `std::priority_queue`). A arquitetura do sistema baseia-se na implementação de TADs, lógica de despacho e a simulação de eventos discretos.

2.1 Tipos Abstratos de Dados (TADs)

Foram implementadas classes fundamentais para representar os componentes para resolução do problema:

- **Classe `Demanda`:** Representa a solicitação unitária de um passageiro. Armazena o identificador único, o instante da solicitação, as coordenadas de origem e destino e o estado atual da demanda (`DEMANDADA`, `INDIVIDUAL`, `COMBINADA` ou `CONCLUIDA`), gerenciado através do `enum class Estado`.

- **Classe `Corrida`:** Abstração da viagem realizada por um veículo. Este TAD é responsável por agregar um conjunto de demandas e construir a rota sequencial de paradas. Calcula métricas essenciais como a distância total percorrida e a eficiência, definida pela razão entre a soma das distâncias euclidianas originais das demandas e a distância total da rota compartilhada.

- **Classe `Trecho`:** Modela a aresta de deslocamento entre dois nós (paradas) consecutivos. Calcula automaticamente a distância e o tempo de viagem com base na velocidade média do veículo (γ). Identifica a natureza do movimento: Coleta (entre embarques), Entrega (entre desembarques) ou Deslocamento (misto).

- **Classe `Parada`:** Define um ponto de parada na rota, associando uma coordenada geográfica e um tipo (Embarque/Desembarque) a demanda atendida.

- **Classe `Evento`:** Representa cada evento da simulação de enventos discretos (SED). Armazena o instante de ocorrência (`tempo`), o tipo do evento (`INICIO_CORRIDA`, `CHEGADA_PARADA`, `FIM_CORRIDA`) e os identificadores necessários para recuperar a qual corrida e parada o evento está associado (ID da corrida e índice da parada). É o objeto manipulado pelo Escalonador.
- **Classe `Escalonador`:** Implementa uma Fila de Prioridade utilizando um **Min-Heap** sobre um vetor estático de objetos `Evento`. É o motor da simulação temporal, garantindo que os eventos sejam processados em ordem estritamente cronológica. As operações de manutenção da propriedade do heap (*refazerSubindo* e *refazerDescendo*) garantem eficiência logarítmica nas inserções e remoções.
- **Class `Ponto`:** Classe/Struct auxiliar fundamental utilizada por todos os outros TADs para representar coordenadas cartesianas (x, y), para manter a consistência dos cálculos e faciliitar a manipulação de coordenadas.

2.2 Algoritmo de Despacho e Simulação

O fluxo de execução do programa, implementado na função `main`, segue duas etapas principais:

2.2.1 Fase 1: Agrupamento de Demandas (Algoritmo Guloso)

O sistema processa as demandas cronologicamente. Para cada demanda não atendida c_0, o algoritmo tenta formar uma corrida combinada verificando as demandas subsequentes c_i dentro de uma janela de tempo δ. Uma demanda candidata é adicionada à corrida se, e somente se, satisfizer conjuntivamente os critérios:

1. **Capacidade:** O número de passageiros não excede η.
2. **Proximidade:** As distâncias entre a nova origem/destino e as origens/destinos já existentes na rota respeitam os limites α e β.
3. **Eficiência:** A eficiência da nova rota proposta é maior ou igual ao limiar λ.

Caso um critério não seja satisfeito, a avaliação para aquela demanda candidata é interrompida imediatamente, caracterizando a abordagem gulosa.

2.2.2 Fase 2: Simulação de Eventos Discretos

Após a definição das corridas, o sistema inicializa o `Escalonador` com os eventos de chegada na primeira parada de cada corrida. O loop principal da simulação retira o evento de menor tempo do Min-Heap, processa a chegada na parada atual e, caso não seja o fim da corrida, agenda o evento da próxima parada somando o tempo de deslocamento do trecho correspondente.

3 Análise de Complexidade

A análise de complexidade foca nas duas estruturas principais: o algoritmo de despacho e as operações do escalonador. Seja N o número total de demandas e E o número total de eventos gerados.

3.1 Complexidade do Despacho

O algoritmo de agrupamento percorre as demandas em um loop principal. Para cada demanda i, ele verifica uma sub-lista de demandas subsequentes j. No pior caso, onde todas as demandas estão dentro do intervalo δ, teríamos um comportamento quadrático.

- A verificação de viabilidade (distância e eficiência) envolve iterar sobre as demandas já aceitas na corrida atual. Como a capacidade do veículo (η) é uma constante pequena e fixa (geralmente ≤ 4), essas operações são consideradas $O(1)$.
- Portanto, a complexidade do despacho é dominada pelo aninhamento dos loops de busca, resultando em $\mathbf{O(N^2)}$ no pior caso. Contudo, na prática, a janela de tempo δ limita o número de candidatos verificados, aproximando o desempenho de $O(N \cdot k)$, onde k é o número médio de demandas em δ.

3.2 Complexidade do Escalonador (Min-Heap)

O `Escalonador` foi implementado como um Heap Binário.

- **Inserir Evento:** A operação de inserção adiciona o elemento no final e realiza o *heapify up* (refazerSubindo). A altura da árvore é $\log E$, logo, a complexidade é $\mathbf{O(logE)}$.
- **Retirar Próximo Evento:** A remoção retira a raiz e realiza o *heapify down* (refazerDescendo). A complexidade também é $\mathbf{O(logE)}$.

Como cada corrida gera um número linear de eventos proporcional ao número de paradas (que é limitado por 2η), o número total de eventos E é proporcional a N. Assim, o custo total da simulação é $O(N \log N)$.

3.3 Complexidade de Espaço

O sistema utiliza alocação estática para garantir previsibilidade.

- Arrays de Demandas e Corridas: $O(N)$.
- Array do Heap (Escalonador): $O(E) \approx O(N)$.

A complexidade de espaço total é $\mathbf{O(N)}$.

4 Estratégias de Robustez

Para assegurar a estabilidade e a corretude do sistema, foram adotadas as seguintes estratégias de programação defensiva:

- **Precisão Numérica (Double vs Float):** Foi identificado que o uso de `float` causava erros de arredondamento cumulativos no cálculo de distâncias geográficas extensas, afetando a decisão de eficiência (λ). Todo o sistema foi implementado utilizando `double`, garantindo a precisão necessária para a corretude da simulação.

- **Alocação Estática de Memória:** Para eliminar riscos de vazamento de memória (*memory leaks*) e erros de segmentação comuns em alocação dinâmica manual, optou-se pelo uso de arrays estáticos com limites de segurança (`MAX_DEMANDAS`, `MAX_EVENTOS`). Isso simplifica o gerenciamento de recursos e aumenta a confiabilidade da execução.

- **Verificação de Limites:** Todas as inserções em arrays e no Heap possuem verificações de borda. Caso o número de demandas ou eventos exceda a capacidade pré-definida, o sistema emite um alerta via `std::cerr` e interrompe a operação de forma previsível, evitando comportamento indefinido por *overflow*.

- **Validação de Entrada:** O sistema verifica o sucesso da leitura dos parâmetros iniciais via `std::cin`. Caso a entrada esteja incompleta ou malformada, a execução é encerrada imediatamente com enissão de um alerta via std::cerr.

5 Análise Experimental

Conforme solicitado na especificação do trabalho, a análise experimental investigou o impacto dos parâmetros de despacho (δ, α, β e λ) no desempenho do sistema. O objetivo foi quantificar o *trade-off* entre o custo computacional (tempo de execução) e a qualidade do serviço (taxa de combinação).

Os experimentos foram conduzidos fixando a capacidade dos veículos ($\eta = 4$) e variando uma dimensão por vez, conforme detalhado nas seções a seguir.

5.1 Cenário 1: Impacto da Janela Temporal (δ)

Neste primeiro cenário, avaliou-se como o intervalo máximo de espera (δ) afeta a formação de corridas. Variamos δ de 10 a 60 unidades de tempo, mantendo os demais parâmetros fixos.

Análise: A observação dos dados revela dois comportamentos distintos:

- **Estagnação da Taxa de Combinação (Gráfico da Direita):** O aumento de δ não resultou em ganho significativo na taxa de combinação, que flutuou apenas entre 0.13 e 0.17. Isso indica que, para este conjunto de dados, o gargalo para o compartilhamento não é temporal. Simplesmente "esperar mais"não gera novas combinações se as origens e destinos não forem compatíveis espacialmente.

- **Instabilidade no Tempo de Execução (Gráfico da Esquerda):** O custo computacional apresentou um comportamento não linear. Destaca-se um pico severo em $\delta = 30$, onde o tempo médio saltou para $\approx 400.000\mu s$, enquanto nos demais casos manteve-se na faixa de $40.000 - 60.000\mu s$. Esse pico sugere que janelas de tempo intermediárias podem criar, ocasionalmente, arranjos de candidatos que forçam o algoritmo a realizar um número excessivo de verificações de compatibilidade.

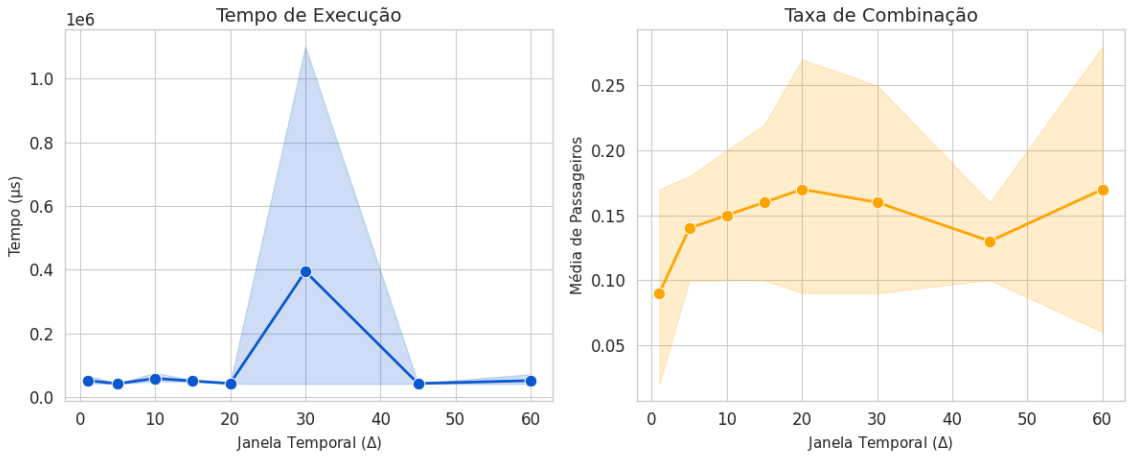

Figura 1: Impacto da variação de δ. À esquerda, nota-se a instabilidade do tempo de processamento (pico em 30); à direita, a estagnação da taxa de sucesso.

5.2 Cenário 2: Impacto da Restrição Espacial (α)

Neste cenário, avaliamos a flexibilidade do desvio de rota variando o parâmetro α (distância máxima entre origens). Este parâmetro define o "quão longe"o veículo pode ir para buscar um passageiro extra.

Análise: Os resultados mostram que este é o fator mais crítico do sistema:

- **Crescimento Exponencial (Gráfico da Direita):** Diferente do cenário temporal, a curva de combinação aqui apresenta um crescimento agressivo. Para $\alpha \leq 1.0$ (desvios mínimos), a taxa é nula (0.0), indicando operação como táxi individual. A partir de $\alpha = 3.0$, a taxa sobe para 0.65 e atinge o pico de **4.33** em $\alpha = 5.0$. Este comportamento gráfico comprova que a restrição espacial é o fator determinante para o sucesso do compartilhamento.

- **Estabilidade de Desempenho (Gráfico da Esquerda):** O tempo de execução manteve-se estável e baixo ($\approx 43.000\mu s$) para quase todos os valores, indicando que a verificação espacial é computacionalmente barata.

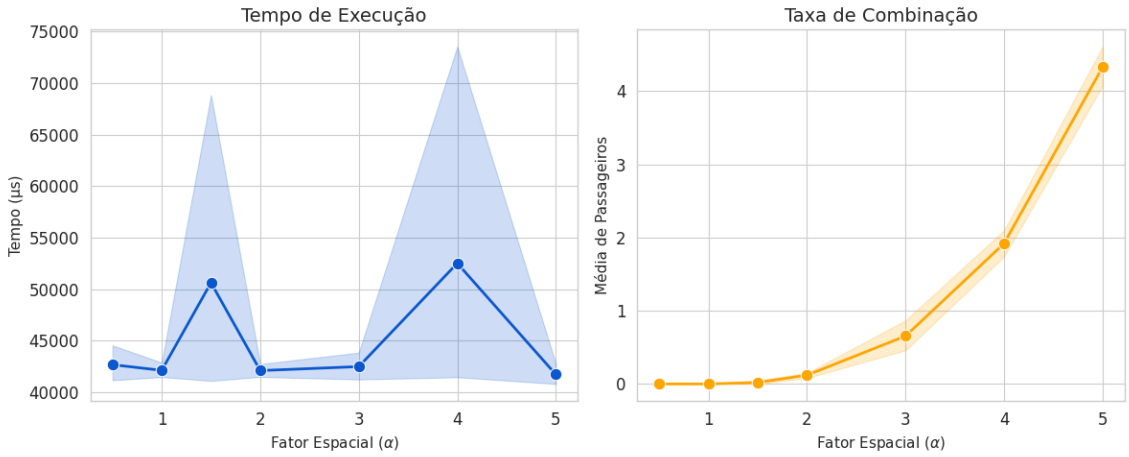

Figura 2: Impacto do parâmetro α. A curva da taxa de combinação (laranja) dispara conforme aumentamos a permissão de desvio, evidenciando o gargalo espacial.

5.3 Cenário 3: Eficiência da Rota (λ)

Por fim, analisamos o parâmetro λ, que define a eficiência mínima da rota.
Análise:

- **Tempo de Execução (Esquerda):** A linha praticamente reta indica que o custo para verificar a eficiência é constante. Exigir rotas mais eficientes não deixa o sistema mais lento nem mais rápido.
- **Taxa de Combinação (Direita):** Entre $\lambda = 0.3$ e $\lambda = 0.6$, a taxa manteve-se estável (0.19). Isso mostra que o sistema é robusto e encontra boas combinações naturalmente.

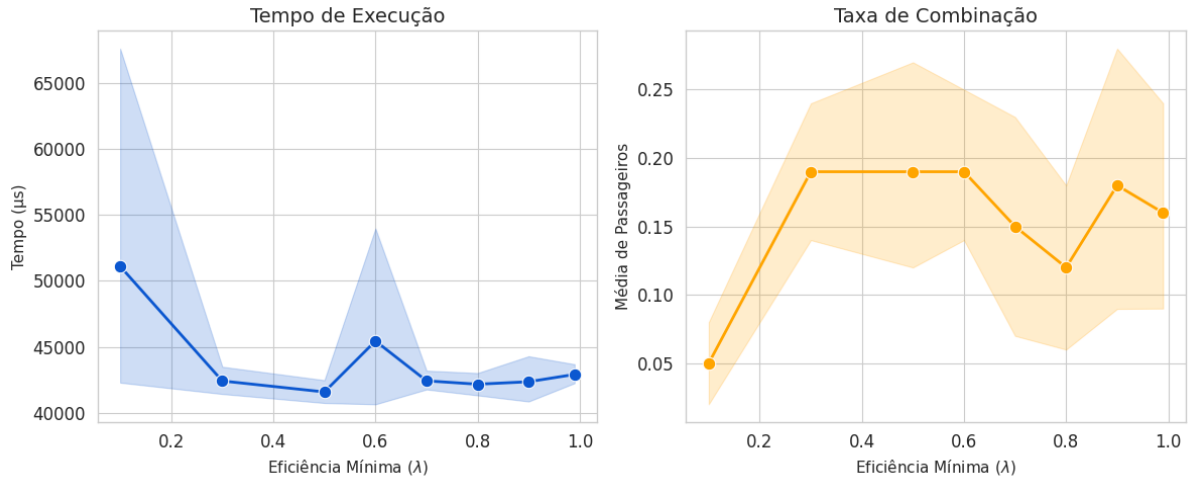

Figura 3: Cenário de Eficiência: A estabilidade em ambos os gráficos (linhas planas) confirma que o desempenho do sistema não depende drasticamente deste parâmetro.

5.4 Discussão dos Resultados

A análise comparativa revela que o gargalo do sistema é **espacial**, e não temporal. Enquanto o aumento da janela de espera (δ) elevou a instabilidade computacional sem ganhos práticos, o relaxamento do desvio de rota (α) provou ser o único fator capaz de escalar a eficiência das rotas.

Conclui-se, portanto, que a estratégia mais eficaz para maximizar o a eficiência é priorizar desvios maiores ($\alpha > 3.0$) em vez de forçar tempos de espera prolongados.

6 Conclusões

Neste trabalho, foi implementado um sistema de simulação de corridas compartilhadas para a empresa "Cabe Aí", utilizando a linguagem C++ e a técnica de Simulação de Eventos Discretos. O projeto envolveu a criação de Tipos Abstratos de Dados (TADs) para representar demandas, corridas e veículos, além da implementação de um escalonador baseado em Min-Heap para gerenciar a cronologia dos eventos.

Com o desenvolvimento, aprendi na prática a importância de escolher a estrutura de dados correta: o uso do Heap tornou o gerenciamento do tempo eficiente, algo que seria inviável com vetores simples. Também compreendi melhor o funcionamento de algoritmos gulosos, percebendo que eles são ótimos para decisões rápidas, mas dependem muito dos parâmetros definidos.

Pelos experimentos realizados, o aprendizado mais relevante foi perceber que o gargalo do sistema é espacial: permitir que o motorista faça desvios maiores traz muito mais resultados do que obrigar o passageiro a esperar mais tempo.

7 Bibliografia

Referências

[1] Anisio, Marcio, Wagner, Washington. (2025). *DCC205/DCC221 Estruturas de Dados: Trabalho Prático 2 - Sistema de Despacho de Transporte por Aplicativo*. DCC/ICEx/UFMG.

[2] Anisio, Marcio, Wagner, Washington, (2025). *Slides da Disciplina Estrutura de Dados* [Material Didático disponível via moodle]. DCC/ICEx/UFMG.

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

[4] Ziviani, N. (2011). *Projeto de Algoritmos com Implementações em Pascal e C* (3ª ed.). Cengage Learning. (Capítulo sobre Filas de Prioridade e Heaps).

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Capítulo 6: Heapsort).

[6] GeeksforGeeks. *Introduction to Min-Heap Data Structure*. GeeksforGeeks. Acessado em 20 de novembro de 2025. Disponível em: https://www.geeksforgeeks.org/dsa/introduction-to-min-heap-data-structure/

[7] Wikipédia. *Simulação de eventos discretos*. Wikipédia. Acessado em 18 de novembro de 2025. Disponível em: https://pt.wikipedia.org/wiki/Simulao_de_eventos_discretos