

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Trabalho Prático 2

Sistema de Despacho de Transporte por Aplicativo

Nome: Mateus Mendes Alves Cabral

E-mail: mttcabral@ufmg.br

Nº de matrícula: 2024088680

Disciplina: Estruturas de Dados

Professores: Anísio, Lucas, Wagner e Washington

Belo Horizonte
2025

Sumário

1	Introdução	2
2	Método	2
2.1	Estruturas de Dados	2
2.2	Tipos Abstratos de Dados (TADs)	2
2.3	Algoritmos e Lógica de Simulação	3
3	Análise de complexidade	3
3.1	Estruturas de Dados	3
3.2	Procedimentos da Simulação	4
3.3	Complexidade Espacial	4
4	Estratégias de Robustez	4
5	Análise Experimental	5
5.1	Metodologia	5
5.2	Resultados e Discussão	5
5.3	Conclusão da Análise	6
6	Conclusões	6
	Bibliografia	6

1 Introdução

O presente trabalho descreve a implementação de um sistema de despacho de corridas para a empresa CabeAí, uma filial da multinacional CabAI. O objetivo principal é simular e otimizar o transporte de passageiros, introduzindo a modalidade de corridas compartilhadas (ride-sharing). O problema consiste em receber um conjunto de demandas de transporte (origem, destino e horário) e alocar veículos para atendê-las, buscando agrupar passageiros com rotas compatíveis para aumentar a eficiência do sistema e reduzir custos.

A solução empregada utiliza uma abordagem de Simulação de Eventos Discretos (SED). O sistema processa as demandas em duas fases principais: primeiramente, uma estratégia gulosa (greedy) é aplicada para agrupar as solicitações em corridas, respeitando restrições de capacidade, tempo e distância. Em seguida, a execução dessas corridas é simulada cronologicamente utilizando uma fila de prioridade para gerenciar os eventos de deslocamento, embarque e desembarque. O simulador foi desenvolvido em C++, priorizando o gerenciamento manual de memória e a implementação de estruturas de dados fundamentais sem o uso da biblioteca padrão (STL), conforme os requisitos do projeto.

2 Método

A implementação do simulador foi estruturada em torno de Tipos Abstratos de Dados (TADs) que modelam as entidades do domínio e estruturas de dados dinâmicas para o gerenciamento das coleções.

2.1 Estruturas de Dados

Para atender à restrição de não utilizar a STL, foram implementadas duas estruturas genéricas fundamentais:

- **Vector**: Um vetor dinâmico (array redimensionável) que gerencia manualmente a alocação de memória. Ele suporta operações de inserção (`push_back`) com redimensionamento automático (estratégia de duplicação de capacidade) e acesso aleatório em tempo constante. É utilizado para armazenar as listas de requisições, corridas e segmentos.
- **MinHeap**: Uma fila de prioridade implementada sobre um **Vector**, organizada como uma árvore binária completa onde o menor elemento está sempre na raiz. Esta estrutura é essencial para o escalonador de eventos, garantindo que os eventos da simulação sejam processados na ordem cronológica correta.

2.2 Tipos Abstratos de Dados (TADs)

O sistema foi modelado com as seguintes classes principais:

- **Request (Demanda)**: Representa a solicitação de um passageiro, contendo ID, horário, origem, destino e estado atual (solicitada, individual, combinada, concluída).
- **Ride (Corrida)**: Agrupa uma ou mais demandas atendidas por um único veículo. Gerencia a rota, composta por uma sequência de segmentos, e calcula métricas como

distância total, duração e eficiência. A eficiência é definida pela razão entre a soma das distâncias diretas das demandas e a distância total percorrida pelo veículo.

- **Stop (Parada):** Representa um ponto geográfico na rota onde ocorre uma ação de embarque (`kPickup`) ou desembarque (`kDropoff`).
- **Segment (Trecho):** Modela a aresta direcionada entre duas paradas consecutivas, armazenando a distância, o tempo de deslocamento e o tipo do trecho (coleta, entrega ou deslocamento).

2.3 Algoritmos e Lógica de Simulação

O fluxo principal do programa (`main.cc`) divide-se em três etapas:

1. **Agrupamento Guloso:** O algoritmo itera sobre as demandas ordenadas temporalmente. Para cada nova demanda, tenta-se criar uma nova corrida e, em seguida, agregar demandas subsequentes que satisfaçam as restrições de:
 - **Capacidade:** O número de passageiros não pode exceder o limite do veículo.
 - **Intervalo de Tempo (δ):** Apenas demandas próximas temporalmente são consideradas.
 - **Distância (α e β):** As origens e destinos devem estar dentro de um raio máximo de proximidade.
 - **Eficiência (λ):** A corrida combinada deve manter uma eficiência mínima.
 - **Atraso Máximo:** O tempo de espera não pode inviabilizar a viagem para o primeiro passageiro.
2. **Escalonamento Inicial:** Para cada corrida formada, um evento inicial é criado e inserido no `MinHeap`, marcado para o horário da primeira solicitação.
3. **Simulação de Eventos:** O laço principal consome eventos do `MinHeap`. Cada evento processado avança o relógio da simulação e, se houver mais paradas na rota da corrida, gera um novo evento futuro correspondente à chegada no próximo ponto, calculado com base na velocidade do veículo e distância do trecho.

3 Análise de complexidade

A análise de complexidade considera N como o número total de requisições e E como o número total de eventos gerados na simulação.

3.1 Estruturas de Dados

- **Vector:**
 - Acesso (`operator[]`): $O(1)$.
 - Inserção (`push_back`): Amortizado $O(1)$. No pior caso (redimensionamento), é $O(K)$, onde K é o tamanho atual, mas a estratégia de duplicação garante custo médio constante.

- **MinHeap:**

- Inserção (**push**): $O(\log M)$, onde M é o número de elementos na heap, devido à operação de **HeapifyUp**.
- Remoção (**pop**): $O(\log M)$, devido à operação de **HeapifyDown**.
- Consulta (**top**): $O(1)$.

3.2 Procedimentos da Simulação

- **Agrupamento Guloso:** No pior caso teórico, para cada requisição, o algoritmo verifica todas as requisições subsequentes para tentar combiná-las. Isso resultaria em uma complexidade de $O(N^2)$. No entanto, na prática, as restrições de capacidade (η) e intervalo de tempo (δ) limitam drasticamente o número de candidatos verificados, aproximando o comportamento de $O(N \cdot C)$, onde C é uma constante média de candidatos.
- **Simulação de Eventos:** O número total de eventos E é proporcional ao número de paradas, que por sua vez é linear em relação ao número de requisições ($E \approx 2N$). Cada evento envolve operações de heap ($O(\log E)$). Portanto, a complexidade da fase de simulação é $O(E \log E)$ ou $O(N \log N)$.

3.3 Complexidade Espacial

O espaço utilizado é dominado pelo armazenamento das requisições e das corridas formadas. Como cada requisição é armazenada uma única vez e pertence a uma única corrida, a complexidade espacial é $O(N)$.

4 Estratégias de Robustez

Para garantir a estabilidade e corretude do sistema, foram adotadas diversas práticas de programação defensiva e robustez:

- **Gerenciamento Manual de Memória:** Como o uso de `std::unique_ptr` ou `std::shared_ptr` não era permitido, foi implementado um controle de alocação e desalocação. Os destrutores das classes **Vector** e **Ride** garantem que a memória alocada para arrays internos e objetos compostos (como **Segment**) seja liberada corretamente, evitando vazamentos de memória (memory leaks).
- **Encapsulamento e Ocultamento de Informação:** Todas as classes utilizam modificadores de acesso (`private`, `public`) para proteger o estado interno. O acesso aos dados é feito exclusivamente através de métodos **Get** e **Set**, prevenindo modificações acidentais e garantindo a consistência dos dados.
- **Const Correctness:** O uso extensivo da palavra-chave `const` em métodos que não alteram o estado do objeto e em parâmetros de referência assegura que dados que não devem ser modificados permaneçam inalterados, permitindo que o compilador detecte erros de lógica em tempo de compilação.

- **Validação de Entrada:** O código verifica o sucesso das operações de leitura (`std::cin`) antes de prosseguir, evitando comportamentos indefinidos caso o arquivo de entrada esteja malformado ou incompleto.
- **Google C++ Style Guide:** O código foi escrito seguindo as práticas recomendadas pelo Google C++ Style Guide, que favorece a legibilidade, a manutenibilidade e a consistência do código.

5 Análise Experimental

Nesta seção, investigamos como o parâmetro de janela de tempo (δ) influencia a capacidade do sistema de agrupar corridas e o impacto disso na eficiência operacional. O parâmetro δ define o tempo máximo que uma solicitação pode esperar para ser combinada com outra, sendo crucial para determinar o equilíbrio entre a taxa de ocupação dos veículos e o tempo de espera dos usuários.

5.1 Metodologia

Para realizar esta análise, variamos o valor de δ de 0 a 250, com incrementos de 10 unidades. Os testes foram conduzidos sobre três cenários distintos de demanda:

- **Horário de pico:** alta densidade de solicitações em um curto período, simulando horário de pico.
- **Área residencial:** solicitações espalhadas geograficamente e temporalmente, simulando áreas residenciais.
- **Cenário de controle:** um cenário de controle com distribuição aleatória uniforme.

As métricas avaliadas foram o **Número Total de Corridas** (quanto menor, maior o agrupamento) e a **Distância Total Percorrida** (custo operacional).

5.2 Resultados e Discussão

Os resultados demonstraram um comportamento consistente em todos os cenários: o aumento de δ levou a uma redução no número total de corridas, indicando que o sistema foi capaz de encontrar mais oportunidades de compartilhamento. No entanto, isso veio acompanhado de um aumento significativo na distância total percorrida.

No cenário *Horário de pico*, observamos uma redução de aproximadamente 11% no número de corridas (de 199 para 177) ao aumentar δ . Contudo, a distância total percorrida aumentou drasticamente, cerca de 140% (de 158.99 para 386.07). Isso sugere que, embora o sistema tenha conseguido agrupar passageiros, os veículos tiveram que realizar grandes desvios para atender às múltiplas solicitações. Esse comportamento é explicado pelos parâmetros permissivos de distância (α e β) e eficiência ($\lambda = 0.3$) utilizados nos testes, que permitiram combinações de rotas pouco eficientes geograficamente em troca de maior ocupação.

O cenário *Área residencial* apresentou a maior taxa de redução de corridas ($\approx 19\%$), caindo de 150 para 122. Similarmente, o custo em distância aumentou consideravelmente ($\approx 148\%$). Isso indica que em áreas de baixa densidade, a janela de tempo maior é essencial para permitir o compartilhamento, mas o custo de deslocamento para buscar passageiros distantes é alto.

5.3 Conclusão da Análise

Concluimos que o parâmetro δ é eficaz para promover o compartilhamento de corridas (ride-sharing). Entretanto, seu aumento indiscriminado, sem um ajuste rigoroso dos parâmetros de restrição espacial (α, β) e de eficiência mínima (λ), pode levar a rotas excessivamente longas. Para um sistema real, recomenda-se um valor de δ que equilibre a redução da frota ativa com um aumento aceitável na distância percorrida, garantindo que a economia de recursos não comprometa a qualidade do serviço prestado ao usuário.

6 Conclusões

O trabalho prático permitiu a implementação completa de um simulador de despacho de corridas com funcionalidade de compartilhamento, atendendo aos requisitos de eficiência e robustez.

Além disso, a aplicação da combinação de algoritmos gulosos para otimização local (agrupamento de corridas) com a estrutura de eventos para execução global demonstrou ser uma estratégia eficaz para resolver o problema proposto.

Bibliografia

GOOGLE. *Google C++ Style Guide*. 2025. Acesso em: 24 nov. 2025. Disponível em: <<https://google.github.io/styleguide/cppguide.html>>. Acesso em: 24 nov. 2025.

LACERDA, A.; SANTOS, M.; Meira Jr., W.; CUNHA, W. *Estruturas de Dados - Trabalho Prático 2: Sistema de Despacho de Transporte por Aplicativo*. [S.l.], 2025. Especificação do Trabalho Prático.

SAMBOL, M. *Heaps in 3 minutes — Intro*. 2025. Acesso em: 24 nov. 2025. Disponível em: <<https://youtu.be/0wPlzMU-k00>>. Acesso em: 24 nov. 2025.

WIKIPEDIA. *Algoritmo guloso*. 2025. Acesso em: 24 nov. 2025. Disponível em: <https://pt.wikipedia.org/wiki/Algoritmo_guloso>. Acesso em: 24 nov. 2025.